25th European Symposium on Algorithms

ESA 2017, September 4–6, 2017, Vienna, Austria

Edited by

Kirk Pruhs

Christian Sohler

Editors

Kirk Pruhs
Department of Computer Science
University of Pittsburgh, USA
krp2@pitt.edu

Christian Sohler
Department of Computer Science
Technische Universität Dortmund, Germany
christian.sohler@tu-dortmund.de

ACM Classification 1998
E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems, G.1.6 Optimization, G.2 Discrete Mathematics, G.4 Mathematical Software, I.1.2 Algorithms, I.2.8 Problem Solving, Control Methods, and Search, I.3.5 Computational Geometry and Object Modeling

ISBN 978-3-95977-049-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-049-1.

Publication date
September, 2017

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.ESA.2017.0

ISBN 978-3-95977-049-1 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

25th Annual European Symposium on Algorithms (ESA 2017).
Editors: Kirk Pruhs and Christian Sohler
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume contains the extended abstracts selected for presentation at ESA 2017, the 25th European Symposium on Algorithms, held in Vienna, Austria, on 4-6 September 2017, as part of ALGO 2017. ESA scope includes original research on both theoretical and applied algorithmics. Since 2002, it has had two tracks, the Design and Analysis Track (Track A), intended for papers on the design and mathematical analysis of algorithms, and the Engineering and Applications Track (Track B), for submissions dealing with real-world applications, engineering, and experimental analysis of algorithms. Information on past symposia, including locations and proceedings, is maintained at `http://esa-symposium.org`. In response to the call for papers for ESA 2017, 271 papers were submitted, 229 for Track A and 42 for Track B. Paper selection was based on originality, technical quality, interestingness, exposition quality, and relevance. Each paper received at least three reviews. After extensive discussions, the two program committees selected 69 papers for inclusion in the program, 58 from track A and 11 from track B. Thus the acceptance rate was about 25% for both tracks. The symposium featured two invited lectures: The first by David P. Woodruff (Carnegie Mellon University) and the second by David Mount (University of Maryland). The European Association for Theoretical Computer Science (EATCS) sponsored a best paper award and a best student paper award. A submission was eligible for the best student paper award if all authors were doctoral, master, or bachelor students at the time of submission. The best student paper award was given to Marc Roth for the paper "Counting restricted homomorphisms via Möbius inversion over matroid lattices".

The best paper award for track A was given to Marek Cygan, Lukasz Kowalik and Arkadiusz Socala for the paper "Improving TSP tours using dynamic programming over tree decompositions". The best paper award for track B was given to Hisao Tamaki for the paper "Positive-instance driven dynamic programming for treewidth".

We wish to thank all the authors who submitted papers for consideration, the invited speakers, the members of the Program Committees for their hard work, and all the external reviewers who assisted the Program Committees in the evaluation process. Special thanks go to the Local Organizing Committee, who helped us with the organization of the conference. Finally, we would like to thank Nicole Funk and Marvin Böcker for their valuable help in editing these proceedings.

Kirk Pruhs
Christian Sohler
July 2017

25th Annual European Symposium on Algorithms (ESA 2017).
Editors: Kirk Pruhs and Christian Sohler
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committees

Design and Analysis (Track A) Program Committee

Christian Sohler (chair)	Technische Universität Dortmund, Germany
Stephen Alstrup	University of Copenhagen, Denmark
Yossi Azar	Tel-Aviv University, Israel
Jaroslaw Byrka	University of Wroclaw, Poland
Amit Chakrabarti	Dartmouth College, USA
Vincent Cohen-Addad	University of Copenhagen, Denmark
Anne Driemel	TU Eindhoven, Netherlands
Alina Ene	Boston University, USA
Matthias Englert	University of Warwick, United Kingdom
Fedor Fomin	University of Bergen, Norway
Dimitris Fotakis	National Technical University of Athens, Greece
Shayan Oveis Gharan	University of Washington, USA
Fabrizio Grandoni	University of Lugano, Switzerland
Martin Grohe	RWTH Aachen University, Germany
Sudipto Guha	University of Pennsylvania, USA
Martin Hoefer	Goethe Universität Frankfurt, Germany
Jochen Koenemann	University of Waterloo, Canada
Robert Krauthgamer	Weizmann Institute of Science, Israel
Stefan Kratsch	Universität Bonn, Germany
Stefano Leonardi	Sapienza University of Rome, Italy
Edo Liberty	Amazon, USA
Wolfgang Mulzer	Freie Universität Berlin, Germany
Ian Munro	University of Waterloo, Canada
Alantha Newman	CNRS, Grenoble, France
Ilan Newman	University of Haifa, Israel
Evdokia Nikolova	University of Texas at Austin, USA
Erik Jan van Leeuwen	Max-Planck Institute for Informatics, Germany
Yuichi Yoshida	National Institute of Informatics, Japan

25th Annual European Symposium on Algorithms (ESA 2017).
Editors: Kirk Pruhs and Christian Sohler

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Engineering and Applications (Track B) Program Committee

Kirk Pruhs (chair)	University of Pittsburgh, USA
Kunal Agrawal	Washington University, St. Louis, USA
Eyjólfur Ingi Ásgeirsson	Reykjavík University, Iceland
Hannah Bast	Albert-Ludwigs Universität Freiburg, Germany
Carola Doerr	Université Pierre et Marie Curie – Paris 6, France
Kurt Mehlhorn	Max-Planck Institute for Informatics, Germany
Rolf Möhring	Beijing Institute for Scientific and Engineering Computing, China
Ben Moseley	Washington University, St. Louis, USA
Martin Nöllenburg	TU Wien, Austria
Jeff Phillips	University of Utah, USA
Rajeev Raman	University of Leicester, United Kingdom
Christian Schulz	Karlsruhe Institute of Technology, Germany
Frits Spieksma	KU Leuven, Belgium
Cliff Stein	Columbia University, USA
Sabine Storandt	Universität Würzburg, Germany

List of External Reviewers

Aaron Bernstein
Abhinav Srivastav
Adi Vardi
Adrian Vladu
Ágnes Cseh
Ahmad Abdi
Akanksha Agrawal
Alan Roytman
Alberto Marchetti-Spaccamela
Alek Vainshtein
Alexander May
Alexandr Andoni
Ali Khodabakhsh
Alkida Balliu
Allan Grønlund
Amin Gheibi
Amir Abboud
Amir Nayyeri
Amirali Abdullah
Anak Yodpinyanee
Anastasios Sidiropoulos
André Nichterlein
André van Renssen
Andreas Emil Feldmann
Andreas Galanis
Andreas Wiese
Andrew McGregor
Angelo Fanelli
Anil Maheshwari
Anna Adamaszek
Antonios Antoniadis
Antonis Thomas
Anupam Gupta
Aounon Kumar
Aparna Das
Arash Haddadan
Arindam Khan
Arnaud De Mesmay
Arne Schmidt
Arnold Filtser
Artur Kraska
Ashish Chiplunkar
Ashley Montanaro
Aurélien Ooms
Avinatan Hassidim
Ayumi Shinohara
Bahareh Banyassady
Bart De Keijzer
Bart M. P. Jansen
Ben Strasser
Benjamin Miller
Benjamin Raichel
Bernd Gärtner
Bingkai Lin
Birgit Vogtenhuber
Bojana Kodric
Boris Klemz
Brendan Lucier
Bruce Shepherd
Bundit Laekhanukit
Charilaos Efthymiou
Chen Attias
Chien-Chung Huang
Chinmay Hegde
Chris Schwiegelshohn
Christian Komusiewicz
Christian Scheffer
Christian Sommer
Christian Wulff-Nilsen
Christina Büsing
Christoph Berkholz
Christoph Dürr
Christopher Musco
Christos Tzamos
Claire Mathieu
Claudia Dieckmann
Clément Canonne
Corey Sinnamon
Corwin Sinnamon
Damian Straszak
Daniel Lokshtanov
Danny Hermelin
Danny Vainstein
David Adjiashvili
David Kirkpatrick
David Manlove
David Peleg
Debmalya Panigrahi
Dennis Olivetti
Dima Kogan

25th Annual European Symposium on Algorithms (ESA 2017).
Editors: Kirk Pruhs and Christian Sohler
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Dimitrios Skrepetos
Dimitris Achlioptas
Dimitris Tsipras
Dirk Oliver Theis
Dirk Sudholt
Don Sheehy
Dror Fried
Elmar Langetepe
Ely Porat
Emmanouil Pountourakis
Enrico Nardelli
Eric Allender
Erik D. Demaine
Erik Lindgren
Erin Chambers
Eva Rotenberg
Eva-Maria Hols
Evangelos Bampas
Evangelos Markakis
Evripidis Bampis
Fabian Stehn
Fahad Panolan
Felix Reidl
Francois Le Gall
Frank Hoffmann
Frederic Magniez
Friedrich Eisenbrand
Ge Nong
George Giakkoupis
Ger Yang
Gerard Tel
Giuseppe F. Italiano
Gonzalo Navarro
Gordon Wilfong
Graham Cormode
Gramoz Goranci
Greg Bodwin
Günter Rote
Guy Even
Haim Kaplan
Haitao Wang
Hang Zhou
Haris Angelidakis
Helmut Alt
Hendrik Molter
Herman Haverkort
Hicham El-Zein
Holger Dell
Hong Wei
Hossein Esfandiari
Hsien-Chih Chang
Huacheng Yu
Hubie Chen
Huy Nguyen
Ian Mertz
Ignasi Sau
Ilan Cohen
Ilya Razenshteyn
Inbal Rika
Inbal Talgam-Cohen
Ioannis Koutis
Ivkin Nikita
Jacob Holm
Jagadish M
Jakub Łącki
Jakub Radoszewski
Jan Marcinkowski
Jan Vondrak
Jannik Matuschke
Jayadev Acharya
Jean-Florent Raymond
Jeff Erickson
Jelani Nelson
Jesper Nederlof
Jesper Sindahl Nielsen
Jiří Sgall
Jittat Fakcharooenphol
Joachim Spoerhase
Joan Boyar
Joanna Ochremiak
Johannes Blömer
Johannes Fischer
Johannes Lengler
José A. Soto
José Verschae
Josh Alman
Julia Komjathy
Justin Ward
Kaiyu Wu
Kanstantsin Pashkovich
Karl Bringmann
Kasturi Varadarajan
Katarzyna Paluch
Katharina Klost
Kazuo Iwama
Keerti Choudhary

Kent Quanrud
Kevin Schewior
Kevin Verbeek
Kim-Manuel Klein
Kiril Solovey
Klaus Kriegel
Konrad Kazimierz Dabrowski
Konstantinos Mampentzidis
Konstantinos Panagiotou
Krzysztof Fleszar
Krzysztof Nowicki
Krzysztof Onak
Kunihiko Sadakane
Kyriakos Axiotis
Laszlo Vegh
Laura Sanita
Laurent Bulteau
Leah Epstein
Lena Schlipf
Lene Favrholdt
Liam Roditty
Lin Yang
Louxin Zhang
Lucas Pastor
Ludwig Schmidt
Łukasz Jeż
Lukasz Kowalik
Maarten Löffler
Magnus Bordewich
Magnus M. Halldorsson
Magnus Wahlström
Manoj Gopalkrishnan
Manoj Gupta
Manuel Penschuck
Marcel Roeloffzen
Marcin Bienkowski
Marcin Mucha
Marcin Pilipczuk
Marcin Wrochna
Marek Adamczyk
Markus Blaeser
Marthe Bonamy
Martin Böhm
Martin Dietzfelbinger
Martin Dyer
Martin Gairing
Masaki Yamamoto
Mateusz Lewandowski
Matias Korman
Matt Weinberg
Matthew Johnson
Matthias Mnich
Matúš Mihalák
Max Klimm
Max Willert
Maxim Sviridenko
Meirav Zehavi
Melanie Schmidt
Miao Qiao
Michael Goodrich
Michael Hamann
Michael Kerber
Michael Lampis
Michael Saks
Michael Walter
Michal Kotrbcik
Michał Pilipczuk
Michal Wlodarczyk
Michal Ziv-Ukelson
Mikkel Abrahamsen
Mohammad Ali Abam
Mohammad Salavatipour
Monika Henzinger
Moran Feldman
Mordecai J. Golin
Morgan Chopin
Moritz Baum
Moritz Muehlenthaler
Morten Stöckel
Nadja Scharf
Naonori Kakimura
Natan Rubin
Naveen Garg
Neal Young
Neil Olver
Nicolas Bousquet
Niklas Hjuler
Nikolai Gravin
Niv Buchbinder
Noah Stephens-Davidowitz
Nodari Vakhania
Norbert Zeh
Ofir Geri
Ohad Trabelsi
O-Joung Kwon
Ola Svensson

Oliver Schaudt
Olivier Devillers
Oren Weimann
Oswin Aichholzer
Pan Peng
Pankaj Agarwal
Panos Giannopoulos
Paresh Nakhe
Pascal Lenzner
Patrick K. Nicholson
Paul Duetting
Paul Wollan
Pavel Kolev
Pavel Veselý
Pawel Gawrychowski
Paweł Schmidt
Peter Jonsson
Petr Golovach
Petr Kolman
Philip Lazos
Philipp Kindermann
Philippe Gambette
Pinar Heggernes
Piotr Krysta
Pranabendu Misra
Prosenjit Bose
Rajesh Chitnis
Ramanujan M. S.
Rasmus Pagh
Ravi Boppana
Ravishankar Krishnaswamy
Rebecca Hoberg
Rémy Belmonte
Rephael Wenger
Reut Levi
Riccardo Colini Baldeschi
Robbie Weber
Robert Ganian
Robert Kleinberg
Roei Tov
Roland Glantz
Roman Rabinovich
Ronald de Wolf
Ruben Becker
S. Muthukrishnan
Sagar Kale
Sahil Singla
Saket Saurabh
Salvatore Ingala
Sam Buss
Samuel Taggart
Sara Cohen
Sariel Har-Peled
Sascha Witt
Satoru Iwata
Saurabh Ray
Sayan Bhattacharya
Sebastian Lamm
Sebastian Ordyniak
Sebastian Siebertz
Sepideh Mahabadi
Serge Gaspers
Seth Pettie
Shashwat Garg
Shay Mozes
Shay Solomon
Sheung-Hung Poon
Shmuel Onn
Shuichi Miyazaki
Simina Branzei
Simon Gog
Sivaramakrishnan Natarajan Ramamoorthy
Soeren Laue
Solomon Eyal Shimony
Søren Dahlgaard
Soumya Basu
Srinivasa Rao Satti
Stavros Kolliopoulos
Stefan Mengel
Stephan Friedrichs
Stephen Fenner
Subhas Nandy
Suguru Tamaki
Sumedha Uniyal
Sushmita Gupta
Takunari Miyazaki
Takuro Fukunaga
Tasuku Soma
Telikepalli Kavitha
Thanasis Lianeas
Thomas Bläsius
Thomas Dueholm Hansen
Thomas Erlebach
Thomas Kesselheim
Till Fluschnik
Till Tantau

Timo Kötzing
Tjark Vredeveld
Tobias Christiani
Tobias Harks
Tobias Maier
Tom van der Zanden
Tomas Balyo
Tomasz Kociumaka
Tomaž Hočevar
Travis Gagie
Troy Lee
Tsvi Kopelowitz
Ulrich Bauer
Ulrich Meyer
Valerie King
Vasileios-Orestis Papadigenopoulos
Vasilis Gkatzelis
Venkatesh Raman
Victor Verdugo
Virginia Vassilevska Williams
Viswanath Nagarajan
Waldo Gálvez
Wanchote Jiamjitrak
William Harvey
Xiaohui Bei
Xin Han
Yakov Nekrich
Yann Disser
Yasushi Kawase
Yoichi Iwata
Yutaro Yamaguchi
Yuval Emek
Yuval Filmus
Yuyi Wang
Zachary Frenette
Zachary Friggstad
Zahed Rahmati
Zohar Karnin

Sketching for Geometric Problems

David P. Woodruff

Carnegie Mellon University, Pittsburgh, PA, USA
dpwoodru@gmail.com

Abstract

In this invited talk at the European Symposium on Algorithms (ESA), 2017, I will discuss a tool called sketching, which is a form of data dimensionality reduction, and its applications to several problems in high dimensional geometry. In particular, I will show how to obtain the fastest possible algorithms for fundamental problems such as projection onto a flat, and also study generalizations of projection onto more complicated objects such as the union of flats or subspaces. Some of these problems are just least squares regression problems, with many applications in machine learning, numerical linear algebra, and optimization. I will also discuss low rank approximation, with applications to clustering. Finally I will mention a number of other applications of sketching in machine learning, numerical linear algebra, and optimization.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases dimensionality reduction, low rank approximation, projection, regression, sketching

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.1

Category Invited Talk

1 Projection

Formally, in the projection problem, we are given a point $b \in \mathbb{R}^n$ and a d-dimensional flat (affine subspace) H, and would like to compute the distance of b to H. In a typical setting, n is very large, and d, while much smaller than n, is also fairly large. Thus we cannot afford algorithms that say, are exponential in d. One way of being presented H is in its coordinate representation, so we can think of H as being the set of points y of the form $y = Ax + v$, where A is an $n \times d$ matrix and v is a point in $\mathbb{R}^n$, which we think of as an offset. Note that A is a tall and thin matrix. Letting $\text{dist}(b, H)$ denote the Euclidean distance of b to H, we have that $\text{dist}(b, H) = \text{dist}(b - v, H - v)$ by translation, where $H - v$ is the set of points y of the form $y = Ax$. Thus we can write $\text{dist}(b - v, H - v) = \min_{x \in \mathbb{R}^d} \|Ax - (b - v)\|_2$, which is just a regression problem. If A has linearly independent columns, i.e., represents a d-dimensional flat instead of a lower-dimensional flat, then the solution $x^* = (A^T A)^{-1} A^T (b - v)$. One can compute x^* in $O(nd^2)$ time, or faster by using fast matrix multiplication algorithms, but for large n and d this is too slow.

In the sketch and solve paradigm, one first relaxes the problem to a randomized approximation problem, instead allowing for one to output an $x' \in \mathbb{R}^d$ for which $\|Ax' - b\|_2 \leq (1 + \epsilon)\|Ax^* - b\|_2$ with large probability. We refer the reader to the survey [21] for more details and proofs of claims, but we describe the basic idea below. The crux of the sketch and solve paradigm is to first choose S from a random family of matrices, and many such families of matrices work, with the important property that S is wide and fat, that is, it has k rows and n columns for $k \ll n$. One then computes $S \cdot A$ and $S \cdot b$. Then one replaces the original regression problem with $\min_x \|(SA)x - (Sb)\|_2$. For small k, which we should

think of as being $\text{poly}(d/\epsilon)$, this problem does not even depend on the large dimension n. Therefore, one can now afford to compute the minimizer x' to this small regression problem using the closed form expression above, in only $\text{poly}(d/\epsilon)$ time. The goal is to choose S from an appropriate random family of matrices so that if one does this, then the minimizer x' is such that $\|Ax' - b\|_2 \leq (1+\epsilon)\|Ax^* - b\|_2$ with large probability.

It turns out that a number of families of random matrices work, such as a $k \times n$ matrix S of i.i.d. normal random variables, where $k = O(d/\epsilon^2)$, and the entries in S are scaled by $1/\sqrt{k}$. The main difficulty with such matrices is that computing $S \cdot A$ is slow. That is, S is a dense matrix, and computing $S \cdot A$ naïvely takes at least nd^2/ϵ^2 time, which is even slower than the exact algorithm for computing x^*, which just took nd^2 time. Note that for the exact algorithm, the bottleneck was in the computation of $A^T A$, and note that both algorithms can be sped up with fast matrix multiplication. While this is too slow for our purposes, in a very nice paper of Sárlos [19], he showed that one could choose S from a much more structured random family of matrices called Fast Johnson Lindenstrauss transforms. This reduces the time for computing $S \cdot A$ to $nd \log n$, and using the connection to regression described above, gives an overall algorithm in $nd \log n + \text{poly}(d/\epsilon)$ time for least squares regression. While this is optimal in the matrix dimensions, often A is itself a sparse matrix and one would like algorithms which run in time proportional to the number $\text{nnz}(A)$ of non-zero entries of A. In work with Clarkson [7] we show this is in fact possible by using the so-called CountSketch matrices from the data stream literature, where we achieve an overall running time of $O(\text{nnz}(A)) + \text{poly}(d/\epsilon)$ for regression. The key property of CountSketch matrices is that they are extremely sparse, having only a single non-zero entry per column. This enables the matrix-matrix product $S \cdot A$ to be computed in only $\text{nnz}(A)$ time. This is easily shown to be optimal, as any algorithm achieving relative error for general matrices A needs to read a constant fraction of the non-zero entries, as otherwise it might miss a very large entry. A number of interesting tradeoffs between the number of rows of S and its sparsity are possible, see also the followup works [15, 17].

In many settings one does not only want to project a point to a flat, but rather to a much more complicated object, such as the union of flats. A natural question is what properties of the object allow for sparse, low-dimensional sketching matrices S. A natural concept that arises is the sphereical mean width, or equivalently, the Gaussian mean width of the object. Intuitively this measures the average fatness of an object, over all directions on the unit sphere. While the sphere is very fat, a line is not. The less fat the object, the fewer dimensions one needs to preserve the norms of points in the object by a sketching matrix. In recent work of Bourgain, Dirksen, and Nelson, sparse sketching matrices for projecting onto general objects were developed [4]. One application of this is to tensor regression [14].

2 Low Rank Approximation

I will also discuss the low rank approximation problem, where the goal is to approximate a high rank matrix by a matrix of much lower rank. Low rank matrices have fewer parameters, and consequently can be stored much more efficiently in factored form and applied to vectors very quickly. Also, in many instances one has an underlying matrix which is of low rank, which then becomes high rank because of noise that was added. Hence in some settings, low rank approximation can also be viewed as a tool for noise removal.

Formally, one is given an $n \times d$ matrix A, and think of the n rows of A as being points in $\mathbb{R}^d$. The goal is to find a rank k matrix A' such that $\|A - A'\|_F \leq (1+\epsilon)\|A - A_k\|_F$, where for a matrix B, $\|B\|_F = \left(\sum_{i \in [n], j \in [d]} B_{i,j}^2\right)^{1/2}$ is the Frobenius norm, and A_k is the

best rank-k approximation to A under Frobenius norm. A natural way of solving low rank approximation is via the truncated singular value decomposition (SVD). Recalling that any matrix A can be expressed as $U\Sigma V^T$, where U and V have orthonormal columns, and Σ is a diagonal matrix with non-negative non-increasing values as one moves down the diagonal, we have that A_k is given by zero-ing out all but the top k diagonal entries of Σ, obtaining Σ_k. This effectively selects the k leftmost vectors of U and k uppermost vectors of V^T, which are also known as the principal components.

While the SVD gives an exact solution, it runs in time $\min(nd^2, dn^2)$, which can be sped up using fast matrix multiplication, but is still much slower than what we would like. As in the case of least squares regression, we can use sketching to obtain significantly faster algorithms if we allow randomization and approximation. Namely, if we allow for outputting a rank-k matrix A' for which $\|A - A'\|_F \leq (1+\epsilon)\|A - A_k\|_F$, then we can solve this problem in $\text{nnz}(A) + (n+d)\text{poly}(k/\epsilon)$ time [7]. To get some perspective on this, even when A is dense, the time, up to $\text{poly}(k/\epsilon)$ factors, is nd, which is significantly faster than what is achievable by the SVD. For sparse matrices, we obtain even larger speedups.

The basic idea behind using sketching for low rank approximation is to first compute $S \cdot A$, where S is one of the random matrices discussed above with a small number of rows, on the order of $\text{poly}(k/\epsilon)$. One then argues that there is a $(1+\epsilon)$-approximate rank-k solution in the span of the rows of SA. It follows that by projecting each of the rows of A onto the rowspan of SA, and then working in the coordinate representation of SA, one effectively reduces the dimension from d to $\text{poly}(k/\epsilon)$. Since the running time of the SVD is $O(nd^2)$, this smaller value of d allows one to now compute the SVD in only $n \cdot \text{poly}(k/\epsilon)$ time. One argues by the Pythagorean theorem that by first projecting the rows of A onto the rowspan of SA, and then performing an SVD, that one still obtains a $(1+\epsilon)$-approximation. Choosing S to be a CountSketch matrix, this whole procedure, except for the projection of the rows of A onto the rowspan of SA, can be executed in $\text{nnz}(A) + (n+d)\text{poly}(k/\epsilon)$ time. The bottleneck is the projection of the rows of A onto the rowspan of SA, but this can be done in $\text{nnz}(A) + (n+d)\text{poly}(k/\epsilon)$ time by using the approximate projection algorithms discussed above.

I will also discuss applications of low rank approximation to k-means clustering. Here the general idea is, if given n points in $\mathbb{R}^d$, to form an $n \times d$ matrix A and then compute a so-called projection-cost preserving sketch of A, which can then be used to prove a low rank approximation with certain strong properties [10, 11, 12]. One then replaces the original dimension d with a much smaller dimension depending on only k and $1/\epsilon$. Given such a small dimension, one then runs standard algorithms from the coreset literature to reduce the number n of points to $\text{poly}(k/\epsilon)$.

3 Additional Applications

Finally, I will conclude by mentioning a number of other problems sketching has been applied to, such as special kinds of low rank approximations callled CUR decompositions, in which the goal is to approximate a matrix A by a low rank matrix in which the factors of the low rank matrix consist of actual rows and columns of A. Thus, if A has sparse rows or columns, then so do its factors. Sketching has been applied successfully to obtain $\text{nnz}(A)$ time algorithms for CUR decompositions [5, 20].

Another interesting use of sketching is to high precision regression. One might complain that the natural sketch and solve algorithm producing a vector $x' \in \mathbb{R}^d$ for which $\|Ax' - b\|_2 \leq (1+\epsilon)\|Ax^* - b\|_2$ has running time $\text{nnz}(A) + \text{poly}(d/\epsilon)$ and is undesirable if ϵ is very small.

By using sketching it is possible to obtain algorithms running in roughly $\text{nnz}(A)\log(1/\epsilon)$ time [7]. The main idea is to use sketching to obtain an $O(1)$-approximate initialization to gradient descent as well as an $O(1)$-approximate preconditioner.

Other applications include robust low rank approximation [8, 20], kernelized problems [1], distributed and streaming computation [2, 3, 6, 13], tensor low rank approximation [20], weighted low rank approximation [18], structure-preseving low rank approximation [9, 16], etc. I refer the reader to my recent monograph for many of the details and additional applications of sketching [21]. While this accompanying article to my ESA talk is primarily focused on my own work, this is just due to the nature of the talk, and please see the above monograph for many other references on these and related topics.

References

1 Haim Avron, Kenneth L. Clarkson, and David P. Woodruff. Sharper bounds for regression and low-rank approximation with regularization. In *RANDOM*, 2017.

2 Maria-Florina Balcan, Vandana Kanchanapally, Yingyu Liang, and David Woodruff. Improved distributed principal component analysis. In *Advances in Neural Information Processing Systems (NIPS)*, 2014. URL: `https://arxiv.org/pdf/1408.5823`.

3 Maria-Florina Balcan, Yingyu Liang, Le Song, David Woodruff, and Bo Xie. Communication efficient distributed kernel principal component analysis. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 725–734. ACM, 2016. URL: `https://arxiv.org/pdf/1503.06858`.

4 Jean Bourgain, Sjoerd Dirksen, and Jelani Nelson. Toward a unified theory of sparse dimensionality reduction in euclidean space. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 499–508, 2015.

5 Christos Boutsidis and David P. Woodruff. Optimal CUR matrix decompositions. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 353–362. ACM, `https://arxiv.org/pdf/1405.7910`, 2014.

6 Christos Boutsidis, David P. Woodruff, and Peilin Zhong. Optimal principal component analysis in distributed and streaming models. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 236–249. ACM, 2016. URL: `https://arxiv.org/pdf/1504.06729`.

7 Kenneth L. Clarkson and David P. Woodruff. Low rank approximation and regression in input sparsity time. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 81–90, 2013. URL: `https://arxiv.org/pdf/1207.6365`.

8 Kenneth .L Clarkson and David P. Woodruff. Input sparsity and hardness for robust subspace approximation. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 310–329. IEEE, 2015. URL: `https://arxiv.org/pdf/1510.06073`.

9 Kenneth L. Clarkson and David P. Woodruff. Low-rank PSD approximation in input-sparsity time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2061–2072, 2017.

10 Michael B. Cohen, Sam Elder, Cameron Musco, Christopher Musco, and Madalina Persu. Dimensionality reduction for k-means clustering and low rank approximation. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOC)*, pages 163–172. ACM, 2015. URL: `https://arxiv.org/pdf/1410.6801`.

11 Michael B. Cohen, Jelani Nelson, and David P. Woodruff. Optimal approximate matrix product in terms of stable rank. In *Proceedings of the 43rd International Colloquium*

on Automata, Languages and Programming (ICALP), Rome, Italy, July 12-15, 2016, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. URL: `https://arxiv.org/pdf/1507.02268`, `doi:10.4230/LIPIcs.ICALP.2016.11`.

12 Dan Feldman, Melanie Schmidt, and Christian Sohler. Turning big data into tiny data: Constant-size coresets for k-means, PCA and projective clustering. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1434–1453, 2013.

13 Ravindran Kannan, Santosh S Vempala, and David P. Woodruff. Principal component analysis and higher correlations for distributed data. In *Proceedings of The 27th Conference on Learning Theory (COLT)*, pages 1040–1057, 2014.

14 Xingguo Li and David P. Woodruff. Near optimal sketching of low-rank tensor regression, 2017. Manuscript.

15 Xiangrui Meng and Michael W. Mahoney. Low-distortion subspace embeddings in input-sparsity time and applications to robust linear regression. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 91–100. ACM, 2013. URL: `https://arxiv.org/pdf/1210.3135`.

16 Cameron Musco and David P. Woodruff. Sublinear time low-rank approximation of positive semidefinite matrices. *CoRR*, abs/1704.03371, 2017.

17 Jelani Nelson and Huy L. Nguyên. OSNAP: Faster numerical linear algebra algorithms via sparser subspace embeddings. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 117–126. IEEE, 2013. URL: `https://arxiv.org/pdf/1211.1002`.

18 Ilya Razenshteyn, Zhao Song, and David P. Woodruff. Weighted low rank approximations with provable guarantees. In *Proceedings of the 48th Annual Symposium on the Theory of Computing (STOC)*, 2016.

19 Tamás Sarlós. Improved approximation algorithms for large matrices via random projections. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 143–152, 2006.

20 Zhao Song, David P. Woodruff, and Peilin Zhong. Low rank approximation with entrywise ℓ_1-norm error. In *Proceedings of the 49th Annual Symposium on the Theory of Computing (STOC)*. ACM, 2017. URL: `https://arxiv.org/pdf/1611.00898`.

21 David P. Woodruff. Sketching as a tool for numerical linear algebra. *Foundations and Trends in Theoretical Computer Science*, 10(1-2):1–157, 2014.

Permuting and Batched Geometric Lower Bounds in the I/O Model

Peyman Afshani[1] and Ingo van Duijn[2]

1 **MADALGO*, Department of Computer Science, Aarhus University, Aarhus, Denmark**
peyman@cs.au.dk

2 **MADALGO, Department of Computer Science, Aarhus University, Aarhus, Denmark**
ivd@cs.au.dk

Abstract

We study permuting and batched orthogonal geometric reporting problems in the External Memory Model (EM), assuming indivisibility of the input records. Our main results are two-fold. First, we prove a general simulation result that essentially shows that any permutation algorithm (resp. duplicate removal algorithm) that does $\alpha N/B$ I/Os (resp. to remove a fraction of the existing duplicates) can be simulated with an algorithm that does α phases where each phase reads and writes each element once, but using a factor α smaller block size.

Second, we prove two lower bounds for batched rectangle stabbing and batched orthogonal range reporting queries. Assuming a short cache, we prove very high lower bounds that currently are not possible with the existing techniques under the tall cache assumption.

1998 ACM Subject Classification F.2.2. Nonnumerical Algorithms and Problems, G.2.1. Combinatorics

Keywords and phrases I/O Model, Batched Geometric Queries, Lower Bounds, Permuting

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.2

1 Introduction

The I/O model [7] is the well-established model to design and analyze algorithms for massive data. In this model, the internal *memory* has size M and the input data is stored in a *disk* of infinite size that is divided into *blocks* of size B. The transfer of data between disk and the memory is done via *I/Os* where each I/O can read or write one block. We define $m = M/B$. All computation must take place in the internal memory. The goal is to minimize the total number of I/Os. This is an elegant model for problems where the size of the input data far exceeds the size of the available memory. Sometimes, algorithms require that $M \geq B^{1+\varepsilon}$ for a constant ε and this is known as the *tall cache* assumption (and the converse as the *short cache* assumption).

Batched Input with Constrained Output. The I/O model has been extensively studied [9, 8, 24]. In this paper, we will focus on proving lower bounds for batched geometric problems as well as engaging in a more in-depth study of the permutation algorithms. The two important batched problems that we study are the following.

* MADALGO Center for Massive Data Algorithmics is supported in part by the Danish National Research Foundation grant DNRF 84.

▶ **Problem 1** (Batched rectangle stabbing (BRS)). *The input comprises a set I of N axis-aligned rectangles and a query set Q of N points in $\mathbb{R}^d$.*

▶ **Problem 2** (Batched orthogonal range reporting (BORR)). *The input comprises a set P of N points and a query set R of N axis-aligned rectangles in $\mathbb{R}^d$.*

In a batched query problem, it is often required that the output should consist of all the pairs (e_i, q_j) where e_i is an input element that matches the query q_j. In this case, which we call the *paired output format*, the two problems stated above are equivalent; both output the set of incidence between an input set of points and rectangles.

In this paper, we consider a different query output format: for every query q_j, we require that all the input elements that match q_j must be placed consecutively in the output. In other words, the algorithm should list the answer to q_j fully before answering any other query. However, there is no restriction on the order in which the queries are answered nor on the order of elements reported for each query. We call this *query output format*. Thus, BRS and BORR are equivalent when we consider the paired output format but they could behave differently if we consider the query output format.

As we shall see shortly, a very connected research direction is in-depth study of algorithms that permute a given set of input elements in the I/O model. A major or interesting (in our opinion) rather open-ended unsolved questions are the following.

▶ **Question 3.** *Can one prove an $\omega(N/B)$ lower bound assuming $M > B^2$ for*
(i) *explicit permutations*
(ii) *or general permuting using any proof technique that is not based on counting?*

▶ **Question 4.** *Let $\mathcal{A}$ be an algorithm that can compute some permutation π of a given N input elements in $\alpha N/B$ I/Os, for some parameter α. Can we transform $\mathcal{A}$ into another algorithm $\mathcal{A}'$ that computes the same permutation π using $\mathcal{O}(\alpha N/B)$ I/Os, such that $\mathcal{A}'$ has a "usefully structured canonical" form, e.g., it uses simple permutation algorithms as building blocks?*

Previous work. Sorting and permuting are possibly the two most fundamental problems in the area of I/O algorithms, with permuting being one of the first problems studied in an I/O setting [19]. Sorting N elements requires $\mathcal{O}(\mathrm{Sort}(N)) = \mathcal{O}\big(\frac{N}{B}\log_m \frac{N}{B}\big)$ I/Os and this bound is tight [7]. The permutation problem is very similar to the sorting problem where the goal is to produce (possibly an implicitly defined) permutation of the input elements. It is also known that any permutation can be performed in $\mathcal{O}(\mathrm{Sort}(N))$ I/Os and there exists permutations that require asymptotically that many I/Os; however, the proof is existential and no such explicit permutation is known to this date [7]. This lower bound (as well as many of the lower bounds in the I/O model) are proved in the so-call *Indivisibility Model*: the data elements are assumed to be indivisible and atomic and each block can store B data elements and the only computation allowed on the atomic elements is to move, delete, or copy them (to or from memory). All other information or computation (unless explicitly mentioned) is free. In the rest of this article, we will only focus on algorithms that work in the indivisibility model. Within the context of permutations in the indivisibility model, there has been attempts to answer Question 3 (or alternatively, to study "easy" permutations) but all the known explicit permutations can be shown to be easier [7, 16, 21] and in particular, they all can be done in $\mathcal{O}(N/B)$ I/Os when we do not have a short cache.

Additionally, there has been a lot of interest in batched problems. For example, in a survey Vitter [24] cites 12 different problems that can be answered in $\mathcal{O}(\mathrm{Sort}(N) + K/B)$

I/Os where N is the total input size and K is the total output size. See also [10, 13, 17, 18, 20]. In particular Arge et al. [11] show that a slightly less restrictive version of Problem 2 can be solved in $\mathcal{O}\Big(N/B \log_m^{d-1} N/B + K/B\Big)$ I/Os. These results produce paired output format.

For the lower bounds, the permutation and sorting lower bounds as well as a problem known as "proximate neighbors" [15], provide a basis of $\Omega(\text{Sort}(N))$ lower bounds for a lot of problems, including problems with batches of N input elements and N queries. Showing a lower bound of roughly $\Omega(\text{Sort}(N))$ for smaller batches is more difficult but some such results are also known [4, 6] (although not explicitly stated in these papers). Lower bounds for dynamic batched queries have also been proved [5]. In general, $\Omega(\text{Sort}(N))$ is the only lower bound available for all of these problems, in particular because in the indivisibility model we can consider any algorithm that solves a batched problem as an algorithm that computes an implicitly defined permutation of the input elements (possibly with duplicates).

Our results. In relation to Question 4, we prove a simulation result that shows any algorithm in the indivisibility model that performs αn I/Os such that it reads and writes each element $\mathcal{O}(\alpha)$ times, can be "simplified" into an algorithm that performs $\mathcal{O}(\alpha)$ rounds where in each round each element is read and written once, using α factor smaller blocks.

In relation to the batched problems and assuming query output format, we prove that if a data structure answers BRS queries in $f(N) + c_0 K/B$ I/Os, for a constant c_0, then $f(N) = \frac{N}{\log B + \log\log \frac{N}{B}} \cdot \left(\frac{\log N}{m^{O(\alpha)}}\right)^{d-1}$, assuming $m \leq B^{\varepsilon}$ for a small enough constant ε. For the BORR problem, then we prove $f(N) = \Omega\Big(\frac{N}{B} \log_m^{d-1}(N)\Big)$. Interestingly, this might mean that BRS is a more difficult problem than BORR in the query output format.

1.1 Preliminaries

Technical barriers. The indivisibility model has been extremely successful in proving lower bounds for algorithmic and data structure problems. However, despite the considerable attention, there are still some very natural questions left open. For instance, we consider Question 3 as a major open question. The situation becomes more exasperating when one considers that the known existential proof in fact shows that almost all permutations should require $\Omega(\text{Sort}(N))$ I/Os to permute but yet, we do not know of a single permutation that even requires $\omega(N/B)$ I/Os. Furthermore, the existential proof (as well as the comparison-based lower bounds for sorting) only can show a $\Omega(\log_m(N!)) = \Omega(\text{Sort}(N))$ lower bound for *any* reasonably defined batched problem. For example, we can only obtain a $\Omega(\text{Sort}(N))$ lower bound for the d-dimensional BORR problem (for a constant d) since the total number of "combinatorially" different point sets of size N in $\mathbb{R}^d$ is at most $N!^d$ and $\log_m(N!^d) = \Theta(\text{Sort}(N))$ for a constant d. Obviously, it is extremely unlikely that this bound is tight and that the d-dimensional BORR problem can be solved in $\mathcal{O}(\text{Sort}(N))$ I/Os.

However, if we assume a short cache, then both of these obstacles go away: we can in fact show lower bounds for explicit permutations such as the matrix transpose permutation and using a different proof strategy [7]. So the natural question becomes, can we actually prove meaningful lower bounds for batched geometric queries under the short cache assumption? Apart from the above considerations, this is also motivated by the desire to understand the effects of short cache on the performance of the algorithms.

Hong-Kung's rounds. While trying to prove a lower bound for the complexity of fast Fourier transform, Hong and Kung [23] presented a general transformation of any I/O

algorithm into a more standard form that works in rounds. While their transformation is originally presented for $B = 1$, it is easily generalizable to larger block sizes. We can thus present their transformation as follow.

▶ **Theorem 5.** *An I/O algorithm $\mathcal{A}$ that runs in a machine with memory size M can be transformed into an equivalent algorithm $\mathcal{A}'$ with the same asymptotic running time on a machine with memory size $2M$ and the same block size such that $\mathcal{A}'$ runs in rounds and during each round, $\mathcal{A}'$ first reads $2M/B$ blocks, performs some computation and then writes $2M/B$ blocks and clears the memory.*

The increase in the block size of the machine in the above theorem is not consequential. It is easy to show that two machines where the block sizes and memory sizes differ only by a constant amount are equivalent, up to constant factors.

▶ **Corollary 6.** *Let $\mathcal{A}$ be an algorithm that works in Hong and Kung's rounds that creates a permutation π of a set of N input elements using $\alpha N/B$ I/Os. At least half of the elements are written at most $\mathcal{O}(\alpha)$ times. Therefore, every such element occurring in an output block can be traced back to one of $m^{\mathcal{O}(\alpha)}$ possible input blocks.*

Proof. By an averaging argument, not more than half of the elements can be written more than 2α times, thus at least half of the elements are written at most 2α times. Since $\mathcal{A}$ works in Hong-Kung rounds, we can trace the elements in an output block to $2m$ other blocks written previously by the algorithm. Those elements, subsequently can be traced back to $(2m)^2$ other blocks. For the elements that are written $\mathcal{O}(\alpha)$ times, the output block is traced back to $m^{O(\alpha)}$ input blocks. ◀

2 Universal External Permuting Algorithm

To study the hardness of permuting, we need to consider arbitrary algorithms that perform a specific permutation. That is, the hardness of a permutation is determined by the optimal algorithm performing it. Often, one admirable goal towards this end is to reduce any permuting algorithm into a "canonical" permuting algorithm that is simpler and easier to study. In fact, Hong and Kung's rounds is one such attempt. However, we would like to probe much deeper. Our basic building block is the following.

2.1 Blocked Shuffle Exchange

To simplify analysing external memory permuting lower bounds, we only consider a single type of algorithm that we call a Blocked Shuffle Exchange (BSE).

▶ **Definition 7.** In a machine with block size B and memory size M, *a blocked shuffle exchange with α phases* is an algorithm with the following structure.

(i) it runs in α phases

(ii) in each phase, the algorithm does the following until

all elements are read and written once: read at most $m = M/B$ blocks into the memory, write some permutation of the read elements to the disk, and then clear the memory.

The goal is to show that we can (partially) simulate any permuting algorithm with a BSE. In particular, the goal is to simulate an algorithm $\mathcal{A}$ that uses at most $\alpha N/B$ I/Os, with a BSE containing $O(\alpha)$ phases. We can in fact do this but under two caveats. The first caveat is that we only simulate the permutation of the elements that are written $O(\alpha)$

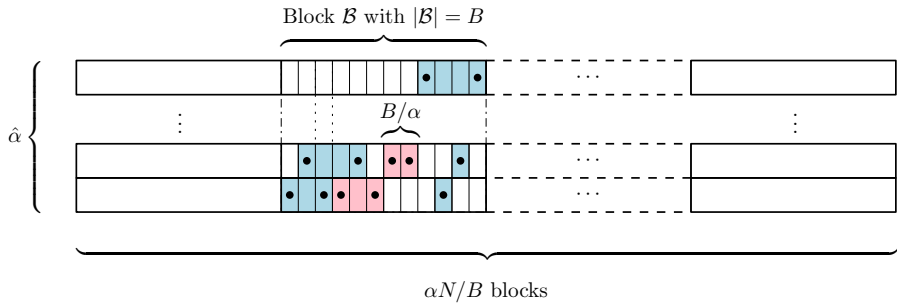


Figure 1 The write history of $\mathcal{A}$ consists of $\alpha N/B$ blocks of size B. Sparse layer blocks can be compactified using blocks of size B/α (2 in this example). Note that all columns contain at most one element.

times. This is necessary because of some (rather uninteresting) bad examples: an algorithm that sorts the first $O(N/\log N)$ elements of an input using N/B I/Os, obviously cannot be simulated with $O(1)$ phases of a BSE. However, the algorithm reads and writes a small portion of the elements many times while not touching the rest. So it is only meaningful to demand a simulation on the subset of the input elements that are not read or written many times. This is what we demand with the first caveat. For the second caveat (that we do not know if it is necessary or not) we define work as block size times number of I/Os performed; for an optimal simulation in terms of work, we need to run our simulation BSE on a smaller block size. The exact formulation of our result is the following.

▶ **Theorem 8.** *Let $\mathcal{A}$ be an algorithm that creates a permutation π of a set of N input elements using $\alpha N/B$ I/Os. Furthermore, we assume that $\mathcal{A}$ writes any element $O(\alpha)$ times.*

Then, we can create a BSE *that creates π using $O(\alpha)$ phases and* either *(i) uses $\alpha^2 N/B$ I/Os* or *(ii) uses the same amount of work but using blocks of size B/α .*

Proof. Observe that we can assume $\mathcal{A}$ writes every element exactly $\hat{\alpha} := c\alpha$ times for a constant c; if some elements are written fewer times, we can just read them and perform dummy writes.

To describe a BSE, we model the *sequential write history* of $\mathcal{A}$. That is, all the writes that $\mathcal{A}$ makes laid out sequentially in the order in which they are written. Now conceptually imagine having $\hat{\alpha}$ copies of this array stacked on top of each other, where each copy forms a *layer*. Every write performed by $\mathcal{A}$ thus corresponds to a column that is composed of $\hat{\alpha}$ *layer-blocks* stacked on top of each other. Assume the layer-blocks in one block are numbered from one to $\hat{\alpha}$, so that the ith block the kth column contains all elements written for the ith time at the kth write. Thus, simulating an I/O by algorithm $\mathcal{A}$ corresponds to reading or writing in the corresponding column.

The observation is that in the simulation, we can compute the $i+1$st layer by only reading from the ith layer. This follows from the fact that every read $\mathcal{A}$ makes is from a previously written block (or input block), and to produce all the $i+1$st writes only requires reading elements written i times. Thus, to compute the next layer, we run $\mathcal{A}$ but replace every read with a read to the corresponding block in the ith layer (and similar for writing to the $i+1$st layer). Since $\mathcal{A}$ uses $\alpha N/B$ I/Os, computing the next layer also takes at most that many I/Os. Since layer-blocks can be very *sparse*, this gives a work of $\hat{\alpha}\alpha N = \mathcal{O}(\alpha^2 N)$.

To achieve $\mathcal{O}(\alpha N)$ work, the layer blocks are tightly packed in smaller B/α-sized blocks. Every simulated I/O is now a sequence of densely filled B/α-sized blocks and one additional sparse block. Since every element occurs exactly once per layer, there are at most $N/(B/\alpha) =$

$\alpha N/B$ dense I/Os per layer. The same bound holds for sparse I/Os, since there are $\hat{\alpha}N/B$ columns, and at most one sparse I/O per column. Together, this yields $\hat{\alpha}\alpha N/B$ I/Os and $(\hat{\alpha}\alpha N/B)(B/\alpha) = \hat{\alpha}N$ work. ◀

The factor α reduction in block size in this result might not be optimal. For small block size, it might happen that $\alpha = \Omega(B)$, and thus the simulation essentially becomes an internal memory simulation. However, for simulations where α is a constant, the theorem is particularly useful.

▶ **Corollary 9.** *To prove that an explicit permutation π requires $\omega(N/B)$ I/Os (and thus $\omega(N)$ work), it is sufficient to prove that permuting π with a* BSE *requires $\omega(N)$ work.*

2.2 Abstract Duplicate Removal

As we show in Section 3, creating the output of a batched problem is not modelled as a permutation problem, but as a duplicate removal problem. Essentially, we can think of the algorithm as an algorithm that runs "backwards" and given the output of the batched problem, it is trying to remove all the duplicates and produce the input set of elements. Because of this, we prove a different simulation result that shows a duplicate removal algorithm can be manipulated to produce a particular permutation of a subset of the elements with some nice properties. Before stating our simulation result, we need to introduce some definitions pertaining to duplicate removal.

▶ **Definition 10.** Consider a set S of K atomic elements together with an equivalence relation $\equiv$ defined on S. An element e_1 is a *duplicate* of an element e_2 if and only if $e_1 \equiv e_2$. The duplicate removal problem is the problem of finding the quotient set or specifically, it is the problem of finding a subset $S' \subset S$ of N elements such that no two elements in S' are equivalent but for every element in S there is an equivalent element in S'.

The duplicate removal problem is trivial if the algorithm has full knowledge of which elements are duplicates and if we only care about the movement of the elements. However, such an algorithm is highly unrealistic. To tie up the algorithm into a more realistic behavior, we force the algorithm into *duplicate elimination framework (DEF).*

1: The algorithm starts with an input of K atomic elements, but with no knowledge of the equivalent relation $\equiv$.
2: At cost of one I/O, the algorithm can read or write a block.
3: The algorithm can move or delete elements in the main memory.
4: The algorithm works in the Hong-Kung's rounds.
5: The algorithm can detect all elements e_1, e_2 in the main memory s.t., $e_1 \equiv e_2$. From now on, the algorithm remembers this for free, for all copies of e_1 and e_2.

Crucially, an algorithm $\mathcal{A}$ can actually delete all copies of an element, if it detects that it is a duplicate of another element. This is a problem for showing lower bounds for the batched problem since this operation can shrink the input size of the duplicate removal algorithm, leaving an easier instance of the problem. In the following theorem, we overcome this difficulty.

▶ **Theorem 11.** *Consider an algorithm $\mathcal{A}$ that works in the DEF and given an input S of size K it detects a subset $S' \subset S$ of $K/2$ duplicate pairs in $\alpha K/B$ I/Os.*

Then, using $O(\alpha K/B)$ I/Os, and using a machine with $M' = M + B$ memory, we can create a permutation of a subset $S'' \subset S$ such that S'' contains $K/4$ pairs of elements (e, e') of S where e is a duplicate of e' and e and e' are placed in the same block.

Proof. Our overall proof strategy is as follows. We allocate a special buffer of size B in the memory where we collect pairs of elements (e, e') such that e is a duplicate of e'. Once the special buffer is full, we write them to the disk. To fill the special buffer we simulate $\mathcal{A}$ two times: once forward and once backwards. During the backwards execution of $\mathcal{A}$ we make some modifications where instead of writing an element e into a block $\mathcal{B}$, we may write an element e' instead. This means that, in the future, when we read the block $\mathcal{B}$, we will have the element e' instead of e. We continue the backwards execution of $\mathcal{A}$ while treating e' the same way e was treated; this is possible since $\mathcal{A}$ only moves or copies the element e and both can be applied obliviously to e' instead. It is important to observe that $\mathcal{A}$ might copy elements (consider it bookkeeping), even though it is a duplicate removal algorithm. Ultimately, what we want to show is that by using the sequence of I/Os that $\mathcal{A}$ performs, we can create an algorithm that produces a permutation of the input such that at least half of the elements reside in a block with at least one equivalent element. In order to show this, we define two notions.

Consider the original execution of $\mathcal{A}$. First, every element e defines a *copy tree* $C(e)$, which is a rooted tree, as follows. There is a node in $C(e)$ for every time e was loaded into a Hong-Kung round. The root of $C(e)$ is the first time the element e is loaded into memory. More than one copy of e could be in memory in a specific round. Therefore, pick one arbitrarily to be the representative that round. Two nodes u and v in $C(e)$ are connected if the block u is loaded from was written in the round where v was the representative. Note that this implies that $C(e)$ is a path if e is only ever moved around and never duplicated.

Second, for every equivalence class $\mathcal{E}$ we define an *equivalence tree* $T(\mathcal{E})$. Two elements e_i and e_j in $T(\mathcal{E})$ are connected if the algorithm discovered their equivalence. This implies in particular that (some copies of) e_i and e_j were in the memory at the same time. It is therefore easy to write all edges of $T(\mathcal{E})$ to a special buffer during the execution of $\mathcal{A}$. However, this would not be a permutation since every element of the equivalence tree is written as many times as its degree.

The basic idea is to output siblings of $T(\mathcal{E})$ in pairs, so that in the at least half of the elements of $T(\mathcal{E})$ are output as disjoint pairs. There are two obstacles with this approach. The most important is that siblings might not reside in memory at the same time. The other obstacle is that nodes might not have an even number of children.

First we show how to handle the second obstacle with the following grouping scheme (the algorithm does not actually perform these operations, but it is considered known by the algorithm). Consider the deepest internal node e. If it has an even number of children they are marked to belong to the same *group*, meaning that they will be paired up later. If e has an odd number of children, then e is grouped together with its children. In either case, e and its children are discarded and the scheme is repeated until there are no nodes left to group. Note that some elements are not grouped (i.e. those that had an even number of children), but at least half of the elements will be grouped. The goal now is to pair each element with exactly one member of its group.

We first run $\mathcal{A}$ with a memory of size $M + B$. The special buffer is used to write discovered pairs to disk, and it is written to the output section on disk when it is full. If an element meets a member of its group in the memory, we write the pair to the buffer. The two elements are now disregarded for the rest of the procedure, meaning they will not be paired up with other elements anymore. Note that all nodes that were grouped with their children have been written to disk. What is left is to show how to pair up the remaining unpaired siblings.

We do this by performing the I/Os of $\mathcal{A}$ backwards by considering writes as reads and vice versa. Elements that have already been paired up will not be used to form new pairs, but are still moved around in the backwards run. Consider the situation where the algorithm

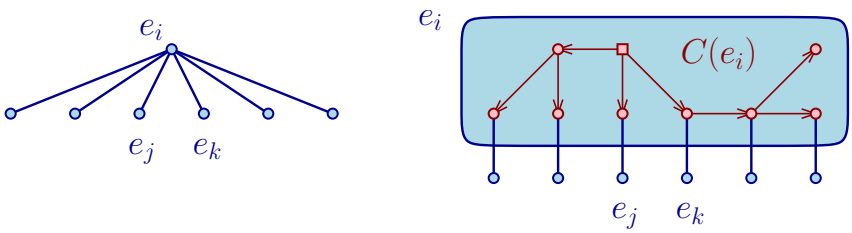


Figure 2 An element e_i and its children in the equivalence tree (left), and e_i as a supernode showing the underlying copy tree $C(e_i)$ (right).

"discovers" an equivalence between e_i and e_j, where e_i is the parent in the equivalence tree and e_j is still unpaired. Since e_i is either already paired in the forward run, or will not be paired at all, we can safely substitute it with e_j. That means that for the rest of the backwards run, every copy of this e_i upwards in the copy tree is now replaced by e_j. The claim is that all unpaired siblings will be paired up in this way.

Figure 2 depicts part of the equivalence tree of e_i. The two children e_j and e_k were never in memory at the same time. However, if we replace the copy of e_i (call it a) that discovers e_j by e_j (and similarly for the copy b that discovers e_k) the following happens. The substituted elements will end up in the memory at the same time, namely at the round where some copy of e_i wrote copies a and b to disk. At this point the elements are written to the buffer and ignored for the rest of the backwards run as usual. If more than two substituted elements meet in memory, then at most one (namely the one that was not written to the buffer) will propagate up the copy tree of e_i. By construction, an even number of siblings were left, so all of them will eventually be paired and written to the buffer.

Thus, all elements that were grouped are written to disk exactly once, and always paired. Since at least half of the elements were grouped, the proof is complete. ◀

3 Batched Lower Bounds for Short Cache

In this section, we describe our lower bounds for offline problems under the short cache assumption. As discussed earlier, a major open problem is to obtain some non-trivial lower bound of $\omega(\text{Sort}(N))$ for some offline problem without the short cache assumption and unfortunately, none of the known techniques seem capable of doing that.

In general, proving lower bounds for geometric problems involves first building a "difficult" input set and then proving that the input is indeed difficult. For our problems, this first part is now considered standard since there have been plenty of lower bounds that have been using similar set of basic constructions of points and rectangles [2, 3, 12, 14, 22].

These standard constructions have been summarized in the following theorems.

▶ **Theorem 12.** *[2, 3, 14] For any parameter n, we can place a set P of n points inside the unit cube $\mathcal{U}$ in $\mathbb{R}^d$ such that for any two points $p, q \in P$, the rectangle created by p and q has volume $\Omega(1/n)$. Furthermore, any rectangle of volume v contains $\Omega(vn - O(1))$ points.*

▶ **Theorem 13.** *[3, 14, 12] For any two parameters n and ℓ, $2 \leq \ell \leq n^{1/3}$, we can place a set R of n rectangles inside the unit cube $\mathcal{U}$ in $\mathbb{R}^d$ with the following properties. There are $t := c_t(\log_\ell n)^{d-1}$ types of rectangles, for a constant c_t, with each type having the dimensions $\left(\frac{1}{\ell}\right)^{i_1} \times \left(\frac{1}{\ell}\right)^{i_2} \times \cdots \left(\frac{1}{\ell}\right)^{i_{d-1}} \times \frac{t\ell^{i_1+i_2+\ldots i_{d-1}}}{n}$, for some integers $i_x \in \{0, \ldots, \log_\ell(n/t)\}$. The set R has the following properties:*

(i) *each rectangle has volume* $\frac{t}{n}$,
(ii) $\Theta(\frac{n}{t})$ *rectangles of each type are sufficient to tile* $\mathcal{U}$,
(iii) *every two rectangles of same type are disjoint, and*
(iv) r *rectangles that have distinct types intersect at a volume at most* $\frac{t}{n\ell^r}$.

3.1 Batched rectangle stabbing problem

In BRS we are given an input set I of N rectangles and a query set Q of N points. The goal is to find for every point q in Q, the set of rectangles that contain q. For every query point q, the algorithm is required to output the set of rectangles that contain q in contiguous blocks. However, the algorithm is given freedom to choose the order in which to report the queries, and within each query, the order of the rectangles that contain q.

▶ **Theorem 14.** *Let $\mathcal{A}$ be an algorithm that given the input sets I and Q for the BRS problem, answers the queries in query order format and in $f(N) + c_0K/B$ I/Os for a constant c_0. We prove that $f(N) = \frac{N}{\log B + \log\log n} \cdot \left(\frac{\log N}{m^{O(c_0)}}\right)^{d-1}$, assuming $B = \Omega(\log\log N)$.*

The first step is to construct the difficult input sets. First, we create a set Q of N points using *Theorem* 12. Then, using Theorem 13, we create a set I_1 of n initial rectangles for a parameter n. Next, we "clone" each initial rectangle β times, where β is a parameter. This is inspired by a data structure lower bound of [1]. Specifically, we create β copies of each initial rectangle and then place the copies in the input set I. Thus, we can construct a set I of $N = n\beta$ rectangles. One should think of the clones as slightly perturbed copies of the original rectangles, meaning, the cloned rectangles are distinct atomic rectangles. However, for simplicity we consider them to cover the same area. If an initial rectangle is stabbed by k query points, all its clones are said to have multiplicity k.

Thus, we have a set Q of query points, and a set I of rectangles. Assume that the algorithm decides to answer the set of queries in the order $< q_1, \cdots, q_N >$. For each q_i, let I_{q_i} refer to the subset of I that contains the point q_i. By Theorem 13 and because of our cloning, I_{q_i} contains $t\beta$ rectangles where $t = (\log_\ell n)^{d-1}$. The output of the algorithm can therefore be described as $O := I_{q_1}, ..., I_{q_N}$. Thus, O is a sequence of atomic elements, where each atomic element is a rectangle from I. Let K be the total length of O. With the input and output formalised, we have the necessary tools to prove the theorem.

Proof of Theorem ??. Consider the input (Q, I) and the output O as described above. O is generated from the sequence in which I is presented to the algorithm. Multiple query points might stab the same rectangle, so O can contain many duplicates. Since the operations of the algorithm are reversible in the indivisibility model, we can consider the algorithm in reverse. In this setting, the sequence O is the input and the goal is to remove duplicates. Observe that we have many duplicates; by Theorem 13, each rectangle $r \in I$ has volume t/n, and therefore by Theorem 12 it contains $\Theta(\frac{t}{n}N) = \Theta(t\beta)$ points. This means r appears $\Theta(t\beta)$ times among the query answers, and thus it is duplicated $\Theta(t\beta)$ times. By assumption the algorithm spends at most $f(N) + c_0K/B$ I/Os to remove all the duplicates. We claim it is enough to prove that this duplicate removal requires more than $(c_0 + 1)K/B$ I/Os. Most of this proof is devoted to proving this *main claim*, and in the end we show how it implies that $f(N) \geq K/B$.

By contradiction, assume the duplicate removal can be done in $\alpha K/B$ I/Os, for $\alpha = c_0 + 1$. If there are more than $K/10$ elements of O that are written more than 10α times, then it follows that the algorithm has spent more than $(10\alpha \cdot K/10)/B = \alpha K/B$ I/Os, which is not possible. Thus, let O' be the subset of O where each element of O' is written at most

10α times and now we know that O' contains at least $9K/10$ elements. Since O contains N unique elements, it follows that O' contains $9K/10 - N \geq 8K/10$ duplicates. Now ignore any element that is not in O' (or assume the algorithm can just remove the duplicates for free outside O'). This means, the algorithm has an input of size at least $9K/10$ and it remove at least $8K/10$ duplicates.

By Theorem 11, we can do a simulation of the duplicate removal with an α (i.e. constant) factor overhead on the number of I/Os. Let $\hat{O}$ be the sequence of elements produced by the simulation, and consider a block $\mathcal{B}$ in $\hat{O}$. By Theorem 11, we know $\mathcal{B}$ is filled with pairs of elements that are duplicates and by Corollary 6, $\mathcal{B}$ can be traced back to $w = m^{O(\alpha)}$ blocks of size B in the sequence O' and in particular to blocks in the sequence $I_{q_1}, ..., I_{q_N}$. Each block of size B can store answers for $\max\{1, \frac{B}{t\beta}\}$ queries and thus w blocks of size B correspond to $u = w \cdot \max\{1, \frac{B}{t\beta}\}$ queries. Since every rectangle is stored in the same block with at most β of its clones, there are at least $\frac{B}{c\beta}$ un-cloned (initial) rectangles in I_1 with multiplicity > 1. That means that there exists a subset $S \subseteq Q$ of queries (where $|S| \leq u$), so that in the set of rectangles in I_1 stabbed by S, there are at least $\frac{B}{c\beta}$ rectangles stabbed by at least two query points.

We show that for the right choice of parameters, this is impossible which would in turn prove our main claim above. To do this, it is enough to show that two query points q_i and q_j cannot stab $r = \frac{B}{c\beta u^2}$ common initial rectangles; if that holds, then the total of common initial rectangles over all u^2 pairs in S cannot amount to $\frac{B}{c\beta}$.

By an area argument, we show that two query points q_i and q_j cannot stab $r = \frac{B}{c\beta u^2} + 1$ initial rectangles. If the area of the intersection of r initial rectangles, which is $\frac{t}{n\ell^{r-1}}$, is smaller than the area spanned by q_i and q_j, which is at least $\Omega(\frac{1}{N}) = \Omega(\frac{1}{n\beta})$ by Theorem 12, then we are done. Thus, we must ensure that $\frac{t}{n\ell^{r-1}} < \frac{1}{c'n\beta}$, (i.e. $c'\beta t < \ell^{r-1}$) for some constant c'. By substituting $t = c_t(\log_\ell n)^{d-1}$ and r we get:

$$c'c_t\beta(\log_\ell n)^{d-1} < \ell^{\frac{B}{c\beta u^2}} \tag{1}$$

$$(d-1)\log\log_\ell n + \log\beta + O(1) < \frac{B\log\ell}{c\beta u^2} \tag{2}$$

$$\beta((d-1)\log\log_\ell n + \log\beta + O(1)) < \frac{B\log\ell}{cu^2} \tag{3}$$

We set $\beta = B/(\log B + \log\log n)$ and thus we get $u = w$ since $\beta t \geq B$. We assume $B = \Omega(\log\log N)$ and thus this implies $\beta \geq 1$ and thus it is a valid choice for β. Then, we observe that setting the value $\log\ell = m^{O(\alpha)}$ satisfies the inequality 3. We thus we obtain a lower bound of $f(N) = \Omega(K/B)$. Since each query point hits exactly $t\beta$ rectangles, the output size is $K = Nt\beta$. For our choice of ℓ, we have $t = \left(\frac{\log N}{m^{O(\alpha)}}\right)^{d-1}$. Using the notation $f \gg g$ for $f = \Omega(g)$, we get

$$f(N) \gg \frac{K}{B} = N \cdot \left(\frac{\log N}{m^{O(\alpha)}}\right)^{d-1} \frac{1}{\log B + \log\log n} \tag{4}$$

◀

This lower bound is higher than the upper bound shown in [11] for the paired output format. One trivial approach to achieve the query output format is sorting the paired output format. This yields $\mathcal{O}\Big(N/B\log_m^{d-1} N/B + K/B\log_m K/B\Big)$ I/Os, which of course does not match our lower bound. Besides, our theorem only applies to algorithms that use $\mathcal{O}(f(N) + \alpha K/B)$ I/Os. It would be interesting to see if our lower bounds can be matched by a specialised algorithm tailored for the query output format.

3.2 Batched Orthogonal Range Reporting

▶ **Theorem 15.** *Let $\mathcal{A}$ be an algorithm that given the input sets P and R for the BORR problem, answers the queries in $f(N)+c_0K/B$ I/Os. And assume $B^\varepsilon > m^{c_0}$ for some small enough constant ε. We prove that*

$$f(N) = \Omega\left(\frac{N}{B}\log_m^{d-1}(N) + K/B\right).$$

Proof. The proof of this theorem follows the same reasoning as the proof of Theorem 14, but the input objects (points) are not cloned. Consider an input (P, R) where P is a set of N points as in Theorem 12 and R is set of n query ranges as in Theorem 13. The value of n is determined by a parameter β so that $\beta = N/n$. Note that we create fewer rectangles than points.

As before, we look at the problem as a duplicate removal problem, define the sequence O of size K and observe that it is sufficient to prove that removal of duplicates from O requires at least $\alpha K/B$ I/Os where $\alpha = c_0 + 1$. As before, we assume otherwise, meaning, we assume that the algorithm can remove duplicates in $\alpha K/B$ I/Os. We then define the sequence O' and use Theorem 11 to define the sequence $\hat{O}$. We know that every block in $\hat{O}$ contains B/c duplicates for some constant c. However, here the role of the rectangles and points are swapped and proofs start to diverge.

The volume of a rectangle is $\frac{t}{n}$, so by Theorem 12 it contains $\Theta(N\frac{t}{n}) = \Theta(t\beta)$ points. Since the points contained in a rectangle are reported consecutively, a block of size B can store answers to $\max\{1, \frac{B}{t\beta}\}$ queries. Thus, setting β to be $\max\{1, \Theta(\frac{B}{t})\}$, every block can store the answers to $\mathcal{O}(1)$ queries. As before, every block in $\hat{O}$ can be traced back to only $m^{O(\alpha)}$ blocks in the sequence O and since very block in the sequence O stores answers of at most $O(1)$ queries, it follows that every block in $\hat{O}$ can be traced back to $w = m^{O(\alpha)}$ rectangles that contains B/c points that are contained in at least two of the rectangles. This means, for some pair of rectangles q_1, q_2, we must have $B/(cu^2)$ common points. Observe that the area of $q_1 \cap q_2$ is at most $O(\frac{t}{n\ell})$ and thus by Theorem 12, $q_1 \cap q_2$ can contain at most $1 + N \cdot O(\frac{t}{n\ell}) = 1 + c'\frac{t\beta}{\ell}$ points, for some constant c'.

Thus, we can get a contradiction by satisfying the inequality

$$1 + c'\frac{t\beta}{\ell} < \frac{B}{w^2}.$$

Observe that if $\frac{B}{w^2} > 1$, then we can pick ℓ large enough such that it satisfies the inequality. In particular, we set $\ell = \Omega(w^2)$. The assumption of $\frac{B}{w^2} > 1$ translates to $B^\varepsilon > m^{c_0}$ which is satisfied by our short cache assumption. Thus, we get a lower bound

$$f(N) = \Omega\left(\frac{K}{B}\right) = \Omega\left(\frac{t\beta n}{B}\right) = \Omega\left(\frac{tN}{B}\right) = \Omega\left(\frac{N\log_{w^2}^{d-1} N}{B}\right) = \Omega\left(\frac{N}{B}\log_m^{d-1} N\right). \quad ◀$$

References

1. Peyman Afshani. Improved pointer machine and I/O lower bounds for simplex range reporting and related problems. In *Symposium on Computational Geometry (SoCG)*, pages 339–346, 2012.
2. Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting in three and higher dimensions. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 149–158, 2009.

3 Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting: query lower bounds, optimal structures in 3-d, and higher-dimensional improvements. In *Symposium on Computational Geometry (SoCG)*, pages 240–246, 2010.

4 Peyman Afshani, Gerth Stolting Brodal, and Norbert Zeh. Ordered and unordered top-k range reporting in large data sets. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 390–400, 2011.

5 Peyman Afshani and Nodari Sitchinava. I/O-efficient range minima queries. In *Scandinavian Workshop on Algorithms Theory*, pages 1–12, 2014.

6 Peyman Afshani and Norbert Zeh. Lower bounds for sorted geometric queries in the I/O model. In *ESA 12: Proceedings of the 20th Annual European Symposium*, pages 48–59, 2012.

7 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

8 L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

9 Lars Arge. *Efficient external-memory data structures and applications.* PhD thesis, Aarhus University, 1996.

10 Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

11 Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Theory and practice of I/O efficient algorithms for multidimensional batched searching problems. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 685–694, 1998.

12 Lars Arge, Vasilis Samoladas, and Ke Yi. Optimal external-memory planar point enclosure. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 40–52, 2004.

13 Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 295–310. Springer, 1995.

14 Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM (JACM)*, 37(2):200–212, 1990.

15 Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 139–149, 1995.

16 T. H. Cormen. Fast permuting on disk arrays. *Journal of Parallel and Distributed Computing*, 17(1):41–57, 1993.

17 Andreas Crauser, Paolo Ferragina, Kurt Mehlhorn, Ulrich Meyer, and Edgar A Ramos. I/O-optimal computation of segment intersections. *External Memory Algorithms and Visualization*, pages 131–138, 1999.

18 Andreas Crauser, Paolo Ferragina, Kurt Mehlhorn, Ulrich Meyer, and Edgar A. Ramos. Randomized external-memory algorithms for line segment intersection and other geometric problems. *International Journal of Computational Geometry & Applications*, 11(03):305–337, 2001.

19 Robert W. Floyd. Permuting information in idealized two-level storage. In *Complexity of computer computations*, pages 105–109. Springer, 1972.

20 Michael T. Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 714–723, 1993.

21 Gero Griener. *Sparse Matrix Computations and their I/O Complexity.* PhD thesis, Technische Universität München, 2012.

22 Joseph M. Hellerstein, Elias Koutsoupias, Daniel P. Miranker, Christos H. Papadimitriou, and Vasilis Samoladas. On a model of indexability and its bounds for range queries. *Journal of the ACM (JACM)*, 49(1):35–55, 2002.

23 Hong T. Kung. Computational models for parallel computers. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 326(1591):357–371, 1988.

24 J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2008. doi:10.1561/0400000014.

Independent Range Sampling, Revisited*†

Peyman Afshani[1] and Zhewei Wei‡[2]

1 **MADALGO§, Department of Computer Science, Aarhus University, Aarhus, Denmark**
peyman@cs.au.dk

2 **School of Information, Renmin University of China, Beijing, China**
zhewei@ruc.edu.cn

Abstract

In the independent range sampling (IRS) problem, given an input set P of n points in $\mathbb{R}^d$, the task is to build a data structure, such that given a range R and an integer $t \geq 1$, it returns t points that are uniformly and independently drawn from $P \cap R$. The samples must satisfy *inter-query independence*, that is, the samples returned by every query must be independent of the samples returned by all the previous queries. This problem was first tackled by Hu *et al.* [15], who proposed optimal structures for one-dimensional dynamic IRS problem in internal memory and one-dimensional static IRS problem in external memory.

In this paper, we study two natural extensions of the independent range sampling problem. In the first extension, we consider the static IRS problem in two and three dimensions in internal memory. We obtain data structures with optimal space-query tradeoffs for 3D halfspace, 3D dominance, and 2D three-sided queries. The second extension considers weighted IRS problem. Each point is associated with a real-valued weight, and given a query range R, a sample is drawn independently such that each point in $P \cap R$ is selected with probability proportional to its weight. Walker's alias method is a classic solution to this problem when no query range is specified. We obtain optimal data structure for one dimensional weighted range sampling problem, thereby extending the alias method to allow range queries.

1998 ACM Subject Classification F.2.2 [Nonnumerical Algorithms and Problems] Geometrical Problems and Computations

Keywords and phrases data structures, range searching, range sampling, random sampling

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.3

1 Introduction

Range searching is a fundamental problems in computational geometry.. The input is a set P of n data points in d-dimensional real space, $\mathbb{R}^d$ (possibly weighted). The goal is to preprocess the points into a data structure, s.t., given a query range R, the points in $P \cap R$ can be counted or reported efficiently. Range searching has been studied extensively and we refer the reader to the survey by Agarwal and Erickson [5] for a broad overview of the area.

Sampling is one of the most natural operations to deal with large data, making efficient and robust sampling vital in many applications. Here, we consider the range sampling

* A full version of the paper is available at http://weizhewei.com/papers/esa17-full.pdf.
† This work was partly supported by the Partially supported by the National Natural Science Foundation of China (NSFC. 61502503).
‡ Corresponding author.
§ A center of Danish National Research Foundation.

problem, where the goal is to design a data structure to support efficient methods to sample from data in the query range. These queries do not fit in the traditional range searching frameworks (such as, the semi-group range searching framework). However, the ability to generate random samples for a given range is useful in many database applications, such as online aggregation [14], interactive queries [6] and query optimization [10]. Within the context of database systems, the importance of sampling queries were identified early on. Olken and Roten's survey from 1995 [20] presents various possible sampling strategies as well as attempts to solve them (see also [19]). However, for spatial queries, i.e., range queries, most of the existing solutions have shortcomings. In one category of solutions, the idea is to use R-trees or Quadtrees where the performance of the data structures depend on input parameters such as "density" and "coverage" [19]. Thus, the worst-case performance of such solutions could be very bad. In another category of solutions, one can select a random sample of the points, preprocess and store them in a data structure (see [15] for more details) but this does not guarantee independence between future and past queries (i.e., asking the same query twice will return the same set of samples).

Hu *et al.* [15] studied the *independent range sampling* problem for the first time using the worst-case analysis. In this variant, it is required that the results of every query must be independent from those returned by the previous queries. Thus, issuing the same query multiple times will fetch different samples, which is desirable in many data analytic applications [12, 26, 17], For example, in *interactive spatial exploration and analytics* [12, 26], the user specifies a query range on the map, and the goal is to continuously generating samples from that range for analytic purpose. The query process is interactive since the user can terminate the query whenever s/he finds the precision of the analysis is acceptable. Independence among the sampling results of all queries is crucial in interactive spatial exploration and analytics, since the user may issue queries with similar query ranges and expect to get independent estimations. Hu *et al.* [15] studied the problem in one dimension for unweighted points, and proposed a data structure that consumes $O(n)$ space, can be updated in $O(\log n)$ time and can answer queries in $O(\log n + t)$ time, where t is the number of samples. In this paper, we study the problem in two and three dimensions, and obtain optimal data structures for some important categories of queries: three dimensional halfspaces and by extension, three-dimensional dominance queries, and two-dimensional three-sided queries. We also propose optimal data structure for one-dimensional *weighted* range sampling problem, in which the sampling probability is defined by the weights of the points.

We focus on the space-query time trade-off for *static* data structures that solve the independent range sampling problem. We focus on the *with-replacement* sampling, in which each sample is independent selected from the query range. We defer the discussion on *without-replacement* sampling to the full version of the paper [1]. For the unweighted case, the input is a set $P = \{p_1, \cdots, p_n\}$ of n points in R^d where U is the domain size, and a range space $\mathcal{R}$. Given a range $R \in \mathcal{R}$ and an integer $t \geq 1$, the query returns a *sequence* of t points, where each element of the sequence is a random point of $P \cap R$ that is sampled uniformly and independently (i.e., with probability $\frac{1}{|P \cap R|}$). We impose the constraint that the sampling result must be independent from those returned by the previous queries.

For the weighted case, each input point p_i is associated with a real-valued weight w_i. The query returns a sequence of t points, where each element of the sequence is a point $p_i \in P \cap R$ sampled independently and with probability $w_k / \sum_{p_j \in P \cap R} w_j$. Note that if range R is omitted in each query, such sampling oracle can be implemented with a classic data structure called *Walker's alias method*, which uses linear space and returns a weighted sample in constant time. Alias method has been successfully adapted in many data mining

algorithms [16, 7], so it is also of theoretical interest to see if it can be extended to support range queries. We assume the existence of an oracle that can generate random real numbers or integers in $O(1)$ time. We assume *real RAM model of computation*: a machine that is equipped with w-bit integer registers, for $w = \Omega(\log n)$, as well as real-valued registers that can store any real number. Arithmetic operations take constant time but storing the contents of a real-valued register into an integer register (via the "floor" function) is only allowed when the result has at most w bits [1].

Our results. We obtain an optimal data structure for three dimensional halfspace ranges for the unweighted independent range sampling problem. Given a query halfspace h, it can extract t independent uniform random samples from h in $O(\log n + t)$ expected time. The structure uses $O(n)$ space. This also implies optimal data structures for two-sided and three-dimensional dominance queries. For weighted range sampling problem, we obtain an optimal data structure for one dimensional point sets in the real RAM model. More precisely, we assume the coordinate of each point can fit in a word of $\Theta(\log U)$ bits, and the real value weights are stored in real registers. The reason we make this assumption is to assure that the space used to store the weights cannot be charged to the space used to store the points. The query is given as an interval $[a, b]$, where a and b are indices. The goal is to extract t independent samples from the indices in $[a, b]$, such that each index $i \in [a, b]$ is selected independently with probability proportional to its weight w_i. Our solution uses $O(n)$ space and answers a query in $O(\text{Pred}(U, w, n) + t)$ time, where $\text{Pred}(U, w, n)$ is the query time of a predecessor search data structure that uses $O(n)$ space on an input of size n from the universe $[U]$ and on a machine with w-bit integers [22].

1.1 Related Work

In the database community, the problem has a long history and it dates back to the 80's when it was introduced as the *random sampling queries* problem. For a database and a given query (range, relational operator, etc.), the goal is to return a random sample set in the query result rather than the entire query result itself. Olken and Rotem considered the problem of independently returning random samples from a query range on B-trees [20], and obtained a structure that returns a sample with $O(\log_B n)$ cost. Olken and Rotem also studied the range sampling problem in high dimensional space using R-tree based structures [21]. We refer the readers to see an excellent survey in [20].

This problem has regained attention recently, due to the "big query" phenomenon where a query result may contain a huge number of elements, and thus it is infeasible to list them all. As mentioned, Hu *et al.* [15] studied the range sampling problem for one dimensional points, with insertions and deletions. They proposed a dynamic RAM structure of $O(n)$ space that answers a range sampling query in $O(\log n + t)$ expected time, and supports an update in $O(\log n)$ time. The static *unweighted* range sampling problem is trivial for one dimensional point sets, since given a query with range $R = [a, b]$ and parameter t, one can perform two predecessor queries to identify the boundaries of the points in R, and one range counting query with constant cost to obtain $P \cap R$, the number of points in R. Then, we can simply sample from $P \cap R$ by generating t random integers between 1 and $|P \cap R|$ and accessing the corresponding t points.

[1] We actually don't need a "floor" instruction since we can simulate it using binary search in $O(w)$ time. As this is used during the preprocessing phase, the query cost can still be kept constant.

Walker's Alias Method. In the weighted sampling problem, the input is a set of non-negative real numbers $w_1, \ldots, w_n$, and the goal is to build a data structure, such that a query extracts an index i with probability $p_i = w_i / \sum_{j=1}^{n} w_j$. The indices returned by different queries should be independent. The classic solution to this problem is Walkers' alias method [25], which uses $O(1)$ query time and $O(n)$ preprocessing time. See the full version of th paper [1] for short description of this method.

Another problem very related to halfspace range sampling is the halfspace range reporting problem where the goal is to simply report all the points in the query halfspace. To see this relationship, observe that extracting t random samples from a query range that contains only t points should extract a faction of the points in the range with constant probability. Halfspace range reporting has been extensively studied for over 30 years, and various results were obtained on this problem. In 2D, the problem was optimally solved in 1985 by Chazelle, Guibas, and Lee [11] but in contrast, the first optimal solution in 3D was obtained relatively recently in 2009 by Afshani and Chan [3], where they showed that one can report the set of points in a query halfspace in $O(\log n + t)$ time using $O(n)$ space where t is the output size. The 3D solution is based on powerful tools created by Matoušek [24, 18] which have been a vital part of all the previous attempts to solve halfspace range reporting problems in three and higher dimensions [8, 24, 23]. Note that the fact that we can match the performance of the best reporting data structures for the queries considered is very desirable.

2 Unweighted Range Sampling in Three Dimensions

Let $p_1, \cdots, p_n$ be a set of three-dimensional points. The main result of this section is an optimal data structure that given a query halfspace h, it can extract t independent uniform random samples from h in $O(\log n + t)$ expected time.

We will use most of the known tools in range reporting: shallow cutting, shallow partition, and partition theorems together with new ideas that take advantage of the structure of the range sampling queries. The rough summary of our approach is as follows: we first build a "core" data structure to sample from query halfspaces that contain many points; later by using shallow cuttings we can extend this to all query halfspaces. To build the core data structure, we create a hierarchy of shallow cuttings and then for each cell in the resulting cuttings, we build data structures that can sample a point uniformly randomly from inside the cell. This part is the main technical contribution since without additional ideas, this approach is not going to give us a linear-space solution[2]. To use only linear space, we build one "global" data structure which is an array that stores the point set in some order, and then for each cell in a shallow cutting, we store a data structure of *sublinear* size that can be used to generate one "random" entry point per sample, into the global array; the final sample point is obtained through this random entry point in *constant time*. A careful analysis shows that the space complexity of the data structure is indeed linear and that each sample is picked with the correct probability.

[2] An expert reader can verify that this approach can easily give us a solution that uses $O(n \log n)$ space, if we spend $O(n)$ space per each shallow cutting level. By using another classical idea, that is, building the shallow cuttings every $\log \log n$ levels and bootstrapping using simplex range searching data structures, this can be reduced to $O(n \log \log n)$. However, this approach seems hopeless to get to $O(n)$ space.

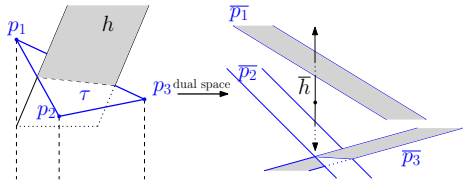


Figure 1 (left) For a k-shallow cutting $\mathcal{F}$, a triangle $\tau \in \Delta(\mathcal{F})$ is shown. For every point on τ, there are at least k and at most $O(k)$ hyperplanes passing below it (not shown here). H_τ is the set of hyperplanes h that are below at least one of the vertices of τ. (right) In dual space, $\overline{h}$ is a point that is below at least one of the hyperplanes corresponding to the vertices of τ.

2.1 Preliminaries and Definitions

We present the dual of a hyperplane h (resp. point p) with $\overline{h}$ (resp. $\overline{p}$): a point p that is below a hyperplane h is mapped to a hyperplane $\overline{p}$ that passes below the point $\overline{h}$. An xy-monotone function in $\mathcal{R}^3$ is a surface, s.t., any line parallel to the z-axis intersects the surface exactly once. Given an xy-monotone surface $\mathcal{F}$ in 3D and a point $q = (q_x, q_y, q_z) \in \mathcal{R}^3$, we say q is *above* $\mathcal{F}$ iff the "downward" ray $(q_x, q_y) \times [q_z, -\infty)$ intersects $\mathcal{F}$. The *below* relationship is defined analogously. Let P be a set of points in 3D and let H be a set of n hyperplanes dual to points in P. A *k-shallow cutting* $\mathcal{F}$ for H is an xy-monotone surface that is a piece-wise linear function composed of $O(n/k)$ vertices, edges, and triangles, s.t., there are at least k and at most $O(k)$ hyperplanes passing below every point of $\mathcal{F}$. The *conflict list* of a point p on $\mathcal{F}$ is defined as the set of all the hyperplanes in H that pass below p and it is denoted by H_p. The conflict list of a triangle $\tau \in \Delta(\mathcal{F})$ is the set of hyperplanes that pass below τ and is denoted by H_τ. The set of points dual to H_τ is denoted by P_τ. See Figure 1.

▶ **Theorem 1** (Shallow Cutting Theorem [24]). *For any given set H of n hyperplanes in 3D and an integer $1 \le k < n/2$, k-shallow cuttings exist. Furthermore, for $k_i = 2^i$, $0 \le i < \log n$, k_i-shallow cuttings $\mathcal{F}_i$, together with the conflict lists of all their vertices, can be constructed in $O(n \log n)$ total time.*

▶ **Lemma 2.** *Given a shallow cutting $\mathcal{F}_i$ and its set $\Delta(\mathcal{F}_i)$ of $O(n/k_i)$ triangles, we can build a data structure of size $O(n/k_i)$ s.t., given a point $p \in \mathcal{R}^3$, we can decide if p is below $\mathcal{F}_i$ or not. In the first case, the triangle $\tau \in \Delta(\mathcal{F}_i)$ that lies directly above p can be found in $O(\log n)$ time.*

Proof. Simply project all the triangles onto the xy-plane. Since $\mathcal{F}_i$ is xy-monotone, we obtain a decomposition of the plane into $O(n/k_i)$ triangles. Build a point location data structure on the planar decomposition [13]. Given the query point p, project it onto the xy-plane, find the triangle τ whose projection contains the projection of p, and decide if p is below τ or not. ◀

▶ **Theorem 3** (Partition Theorem [18]). *Given a set P of n points in 3D and an integer $0 < r \le n/2$, there exists a partition of P into r subsets $P_1, \cdots, P_r$, each of size $\Theta(n/r)$, s.t., each subset P_i is enclosed by a tetrahedron T_i, s.t., any hyperplane crosses $O(r^{2/3})$ tetrahedra.*

▶ **Theorem 4** (Shallow Partition Theorem [24]). *Given a set P of n points in 3D and an integer $0 < r \le n/2$, there exists a partition of P into r subsets $P_1, \cdots, P_r$, each of size $\Theta(n/r)$, s.t., each subset P_i is enclosed by a tetrahedron T_i, s.t., any halfspace that has at most n/r points of P crosses $O(\log r)$ tetrahedra.*

We also need the following known optimal data structures for halfspace range reporting (Theorem 5) and approximate halfspace range counting (Theorem 6).

▶ **Theorem 5** ([3]). *Given a set P of n points in $\mathcal{R}^3$, one can build a data structure of linear size s.t., given a query halfspace h, it can list the points in $P \cap h$ in $O(\log n + |P \cap h|)$ time.*

▶ **Theorem 6** ([4, 2]). *Given a set P of n points in $\mathcal{R}^3$, and a constant $\varepsilon > 0$, one can build a data structure of linear size s.t., given a halfspace h, in $O(\log n)$ time, one can produce an integer $\tilde{k}$ s.t., $\tilde{k}/(1+\varepsilon) \leq |h \cap P| \leq \tilde{k}$.*

2.2 The Overall Data Structure

We now return to our original problem. Our input is a set P of n points in $\mathcal{R}^3$. Let H be the set of hyperplanes dual to P. Define $k_i = 2^i, 0 \leq i < \log n$. We say an integer m is *large* if it is greater than $2^{C(\log\log n)^2}$, for a global constant C to be set later. The following lemma will be proved in the next subsection.

▶ **Lemma 7.** *Given a set P of n points, we can build a structure of linear size to answer the following queries. Given any query halfspace h in which $|P \cap h|$ is large, we can extract t independent random samples from $P \cap h$ in $O(\log n + t)$ time.*

Furthermore, the query can be carried over in an "online" fashion. After the initial search time of $O(\log n)$, the data structure can fetch each subsequent sample in $O(1)$ expected time, until interrupted by the user.

By standard techniques, this gives us our main theorem. See the full version of the paper [1] for the proof.

▶ **Theorem 8.** *Given a set P of n points in $\mathcal{R}^3$, we can build a data structure of size $O(n)$ s.t., given a halfspace h and a parameter t, we can extract t samples from the subset $P \cap h$ in $O(\log n + t)$ expected time.*

Furthermore, the query can be carried over in an "online" fashion, without knowledge of t: After the initial search time of $O(\log n)$, the data structure can fetch each subsequent sample in $O(1)$ expected time, until interrupted by the user.

The above theorem easily extends to sampling from dominance queries as well. Given two points p and q in d-dimensional space, q *dominates* p if every coordinate of q is greater than that of p. In dominance reporting, the goal is to preprocess a set of n points s.t., given a query point q all the points dominated by q can be reported efficiently. As observed by Chan *et al.* [9], three-dimensional dominance queries can be solved using halfspace queries. It is also known that a dominance query can solve two-dimensional a three-sided query, that is, a query region $[a, b] \times (-\infty, c]$ given by three values a, b, and c.

2.3 Proof of Lemma 7

As previously mentioned, this is the heart of the problem and this is where we significantly deviate from the previous techniques (even though we use similar building blocks): To obtain optimal halfspace range reporting, Afshani and Chan [3] rely heavily on the fact that if a halfspace h contains too many points, then the data structure is allowed to spend a lot of time on the query, since we will spend a lot of time producing the output; in other words, the search time can be charged to the output size. In our case, we might be interested only in a small subset of points in h and thus the search cost cannot be charged to the output size.

Our idea is to build two main components: a global array and a number of local structures. The global array will store each point once in an array of size n, in some carefully selected order. The array compactly stores a number of "canonical sets" of total size $O(n \log n)$. We

use shallow cuttings to build the local structures. The important point is that the local structures in total will have *sublinear* size and their utility is to find entry points into the global structure: given a query, using the local structures we locate a subarray of the global array and then uniformly sample from the subarray. We present the technical details below.

Using Shallow Cutting Theorem, we build a k_i-shallow cutting $\mathcal{F}_i$ (as well as its set of triangles $\Delta(\mathcal{F}_i)$), for each large k_i where $k_i = 2^i, 0 \leq i < \log n$, We have $|\Delta(\mathcal{F}_i)| = O(n/k_i)$ by the Shallow Cutting theorem. For a triangle $\tau \in \Delta(\mathcal{F}_i)$, the conflict lists H_τ and P_τ are defined as before. Observe that for each triangle $\tau \in \Delta(\mathcal{F}_i)$, with vertices p_1, p_2 and p_3, H_τ is the union of H_{p_1}, H_{p_2}, and H_{p_3} (with duplicates removed). In this subsection, h will refer to the query halfspace in the primal space. In dual space we will denote $\overline{h}$ with q. Thus, our objective is to either sample a random point of P below h, or a random member of H_q (a random hyperplane of H that passes below q).

The Global Structure. Using Shallow Partition Theorem, we build a partition tree T_{global} as follows. The root of T_{global} represents P. Consider a node of T_{global} that represents a subset $S \subseteq P$. We use Shallow Partition Theorem with parameter $r = |S|^\varepsilon$ to obtain subsets $S_1, \cdots, S_r$, for a small enough constant $\varepsilon > 0$. If a subset S_i contains at most b points, for a parameter b to be defined later, we call it a *base subset*. Unlike the approach in [3], we only recurse on subsets S_i that are not base subsets. Thus, the leaves of T_{global} are base subsets. For each base subset B, we build a secondary data structure that is another partition tree T_B: The root of T_B represents B. At a node of T_B that represents a subset $S \subseteq B$, we use Partition Theorem (*not* the shallow version) with parameter $r = |S|^\varepsilon$ to obtain subsets $S_1, \cdots, S_r$. We recurse on each subset S_i until we reach subproblems of constant size. We store an in-order traversal of the leaves of T_B, in an array A_B; the size of A_B is exactly equal to $|B|$ and for every internal node $v \in T_B$ the points in the subtree of v are mapped to a contiguous interval of array A_B. We build our global array A by concatenating all the arrays A_B over all base subsets B. Finally, we build a data structure for approximate range counting queries (Theorem 6).

The Local Structure for τ. Consider a triangle $\tau \in \Delta(\mathcal{F}_i)$ and let p_1, p_2, and p_3 be the vertices of τ. Remember that H_τ was defined as the union of conflict lists of p_1, p_2, and p_3 after duplicate removal (Figure 1). We will store a local structure for τ that consumes $o(|H_\tau|)$ space ($O(|H_\tau|/\log^{O(1)} n)$ to be more precise). p_1, p_2 and p_3 correspond to three different hyperplanes, $\overline{p_1}, \overline{p_2}$ and $\overline{p_3}$ in the primal space; a hyperplane $h \in H_\tau$ corresponds to a point $\overline{h} \in P$ that is below one of the hyperplanes $\overline{p_1}, \overline{p_2}$ or $\overline{p_3}$. Let P_τ be the set of such points (in other words, P_τ is the set of points dual to hyperplanes in H_τ). Let $B_1, \cdots, B_m$ be the base subsets that are intersected by or are below at least one hyperplane $\overline{p_i}, 1 \leq i \leq 3$. We will store a data structure of size $O(mb^{3/4})$ at triangle τ: For each base subset B_i, we consider the partition tree T_{B_i}. For every node $v \in T_{B_i}$, the subset of points in the subtree of v defines a canonical subset of B_i. By the properties of partition trees (see e.g., [18]), we can write $P_\tau \cap B_i$ as the union of $O(|B_i|^{2/3} \log |B_i|^{O(1)}) = O(|B_i|^{3/4}) = O(b^{3/4})$ canonical subsets of B_i. However, as each subtree of T_{B_i} maps to a contiguous interval of A_{B_i}, it follows that we can represent $P_\tau \cap B_i$ as the union of $O(|B_i|^{3/4})$ intervals from A_{B_i}. We collect all these intervals, over all base subsets $B_1, \cdots, B_m$. Let $I_1, \cdots, I_M$ be the set of all such intervals. Observe that we have $|I_1| + |I_2| + \cdots + |I_M| = |P_\tau|$ since the intervals partition P_τ. Also, $M = O(mb^{3/4})$. We store the numbers $|I_1|, |I_2|, \cdots, |I_M|$ in a data structure $T_{\text{sample}}(\tau)$ for weighted sampling, using the Alias method; the data structure consumes $O(M) = O(mb^{3/4})$ space and in $O(1)$ time can produce a pointer to an interval I_j with probability $\frac{|I_j|}{|I_1|+|I_2|+\cdots+|I_M|} = \frac{|I_j|}{|P_\tau|}$.

Answering Queries. Using approximate halfspace range counting data structure, we can produce an integer $\tilde{k}$ s.t., $\tilde{k}/2 \leq k \leq \tilde{k}$. Let i be the smallest index s.t., $\tilde{k} \leq k_i$. We can find $\tilde{k}$ and i in $O(\log n)$ time, by Theorem 6. Clearly, q is below k_i-shallow cutting $\mathcal{F}_i$. Let $\tau \in \Delta(\mathcal{F}_i)$ be the triangle that lies above the query point q. τ can be found in $O(\log n)$ time using a point location query. We claim it is sufficient to be able to sample from H_τ: to sample a hyperplane that passes below q, we repeat extracting independent uniform samples from the set H_τ until we find one that passes below q. Since H_q is a subset of H_τ, this guarantees independent uniform sampling. On the other hand, since $|H_q| \geq \tilde{k}/2$ we get that $|H_\tau| = O(k)$, and thus on average we need to extract $O(t)$ random samples from H_τ to produce t random samples from H_q[3]. Note that after initial $O(\log n)$ time to find τ, we spend $O(1)$ expected time per sample and thus we can continue without knowledge of t.

Sampling from H_τ. Consider the intervals $I_1, \cdots, I_M$ stored for the triangle τ; by construction, the points stored in these intervals form a partition of P_τ. Using structure $T_{\text{sample}}(\tau)$, in $O(1)$ time, we can select an interval I_j with probability $\frac{|I_j|}{|P_\tau|}$. Next, we generate a random integer ℓ between 1 and $|I_j|$ and output the ℓ-th point in the interval I_j. Clearly, the probability of outputting an element of P_τ is exactly equal to $\frac{1}{|P_\tau|}$ and query time is $O(1)$ per sample.

Space Analysis. This is the main part of the proof. First, observe that the global structure clearly consumes linear space since points in every base subset B are stored only once in the array A_B. Thus it remains to bound the space usage of the local structures. Consider a triangle $\tau \in \Delta(\mathcal{F}_i)$ with its three vertices p_1, p_2, and p_3. Let H_{p_1} be the set of hyperplanes of H below p_1, or equivalently, let $P_{\overline{p_1}}$ be the subset of points of P below the hyperplane $\overline{p_1}$. Let $k = |H_{p_1}|$. Let $f(n,k)$ be the maximum number of base subsets of T_{global} intersected by any hyperplane h that lies above k point of P. Remember that we have used the Shallow Partition theorem with parameter $r = n^\varepsilon$. If $n \leq b$, then we are already at a base subset so $f(b,k) = 1$. Otherwise, depending on the value of k, we might intersect either all the r subsets or only $O(\log r)$ subsets. So we get the following recurrence, which is a generalization of the one found in [3] (we must note that the recurrence in [3] bounds the query time where here we are only bounding the crossing number):

$$f(n,k) \leq \begin{cases} 1 & \text{if } n \leq b \\ \sum_{i=1}^{O(\log n)} f(cn^{1-\varepsilon}, k_i) & \text{if } k \leq n^{1-\varepsilon} = \frac{n}{r} \\ \sum_{i=1}^{n^\varepsilon} f(cn^{1-\varepsilon}, k_i) & \text{if } k > n^{1-\varepsilon} = \frac{n}{r} \end{cases} \quad \text{where } k = \textstyle\sum_i k_i.$$

We solve the recurrence by guessing that it solves to the "correct" bound, that is, it solves to $f(n,k) = 2^{c(\log\log n)^2} + kg(n)/b^{1-3\varepsilon}$, where $g(n)$ is a monotonously increasing function that is always upper bounded by a constant and c is a constant. This is similar to the analysis done in [3], so we postpone the details to the full version of the paper [1].

The analysis shows that the total number of the base sets intersected by three hyperplanes $\overline{p_1}, \overline{p_2}$, or $\overline{p_3}$ is at most $3f(n, O(k_i))$. Thus, the value m in the local structure of τ is bounded by $3f(n, O(k_i))$. The local structure of τ consumes $O(mb^{3/4})$ space. Remember that we are aiming to build the data structure for halfspace containing large number of points. Thus,

[3] This is the only part that breaks down for the weighted range sampling problem. This is also the only hurdle that makes the query time "expected". All the other parts of the data structure work with a worst-case query time.

$k_i = \Omega(2^{C(\log\log n)^2})$. We set the constant C in the definition of a large integer to $2c$, which means $k_i = \Omega(2^{2c(\log\log n)^2})$. We also set $b = \log^{C'} n$ for a large enough constant C'. We plug the values in $f(n,k)$, and thus space used by the local structure of τ is bounded by

$$O(mb^{\frac{3}{4}}) = O\left(f(n, O(k_i))b^{\frac{3}{4}}\right) = O\left(\left(2^{c(\log\log n)^2} + \frac{k_i}{b^{1-3\varepsilon}}\right) b^{\frac{3}{4}}\right)$$
$$= O\left((\log n)^{\frac{3C'}{4}}\, 2^{c(\log\log n)^2} + \frac{k_i}{(\log n)^{(\frac{3C'}{4}-3\varepsilon)}}\right) = O\left(\frac{k_i}{\log^2 n}\right)$$

if we set C' large enough and set $\varepsilon < 1/4$. The number of triangles τ in $\Delta(\mathcal{F}_i)$ is $O(n/k_i)$ and thus the total amount of space used by the triangles is $O(n/\log^2 n)$ and over all the indices this sums up to $o(n)$. This proves all the local data structures consume sublinear space and concludes the proof of Lemma 7.

3 Weighted Range Sampling in One Dimensions

In this section, we address the one-dimensional range sampling problem. Let U be an integer that denotes the universe size of the coordinates. We assume the word size $w = \Theta(\log U)$, such that the coordinate of each point can fit in a word. The input is a set $P = \{p_1, \cdots, p_n\}$ of n points on grid $[u]$, and each point p_i is associated with a non-negative real-valued weight w_i. Given an interval $R = [a, b]$ and an integer $t \geq 1$, the query returns a sequence of t points, where each element of the sequence is random point $p_i \in P \cap R$ that is sampled independently and with probability $w_k / \sum_{p_j \in P \cap R} w_j$.

Note that the coordinate of a point can be stored in a word of $w = \Theta(\log U)$ bits, but a weight is a real number and cannot be stored in an *integer* word. We say the space usage of a data structure is $S(n)$ if it uses at most $S(n)$ words and at most $S(n)$ real registers.

Weighted vs. Uniform IRS. We first offer some intuition to show that weighted independent range sampling is a non-trivial problem, even in one-dimension. Consider one-dimensional uniform independent range sampling problem. There is a simple solution: we store the points of P in ascending order using an array A. Given a query range $[a, b]$ and an integer t, we perform predecessor/successor search to identify the subsequence in A that consists of the elements covered by q. Then, we can simply sample from the subsequence by generating t random ranks and accessing t points. The total query cost is $O(\text{Pred}(U, w, n) + t)$ where $\text{Pred}(U, w, n)$ is the query time of a predecessor search data structure that uses $O(n)$ space on an input of size n from the universe $[U]$ and on a machine with w-bit integers [22]. For the weighted IRS problem, the above approach does not work. The main issue is that sampling from the identified subsequence requires an alias structure designed specifically to that subsequence. Since there are $\Omega(n^2)$ difference subsequences, one needs $\Omega(n^2)$ space to make this approach work.

Notations. We begin by defining some notations. Given a set S, we use $W(S)$ to denote its weight. With a slight abuse of notation, we also use $W(S)$ to denote the set S. Given two integers $1 \leq a \leq b \leq u$, $[a, b]$ is the range from a to b. With a slight abuse of notation, we will also use $[a, b]$ to denote the points in range $[a, b]$, and $W(a, b) = \sum_{p_k \in [a,b]} w_k$ to denote the total weights in $[a, b]$. We use $P_{pre}(a)$ to denote the predecessor of a in P, and $P_{suc}(a)$ to denote the successor of a in P. Given a point $p_i \in [a, b]$, we use $W_{pre}(p_i, a, b) = \sum_{p_j \in [a,b], j<i} w_j$ to denote the prefix sum of point p_i in $[a, b]$, and $W_{suc}(p_i, a, b) = \sum_{p_j \in [a,b], j>i} w_j$ to denote the suffix sum of point p_i in $[a, b]$, respectively.

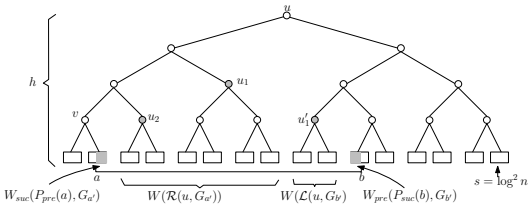


Figure 2 A schematic illustration of the fat points and partial sums.

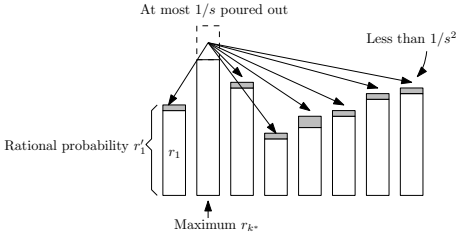


Figure 3 A schematic illustration of the rounding process.

Let $\mathcal{T}$ denote a balanced binary tree on the n points, with height $h = \log n$. Given an internal node u, we use $W(u)$ to denote the total weight of the subtree rooted by u. Fixing an internal node u and a leaf v in u's subtree, let $\mathcal{P}(u, v)$ be the set of nodes on the path from u to v, excluding node u. We define the *left canonical set* of $\mathcal{P}(u, v)$ to be $\mathcal{L}(u, v) = \{w \in \mathcal{P}(u, v) \mid w \text{ is a left child}\} \cup \{v\}$, and similarly the *right canonical set* to be $\mathcal{R}(u, v) = \{w \in \mathcal{P}(u, v) \mid w \text{ is a right child}\} \cup \{v\}$. It is easy to see that the point set in range $[a, b]$ is made up by the subtrees rooted at the nodes in $\mathcal{R}(u, P_{pre}(a)) \cup \mathcal{L}(u, P_{suc}(b))$. Here we define $P_{pre}(a) = P_{suc}(a)$ if a is in S.

A baseline structure. We will use the following baseline structure, which uses $O(n \log^2 n)$ space draws sample with constant cost. The proof of Lemma 9 is deferred to the ful version of the paper.

▶ **Lemma 9.** *For the one-dimensional weighted IRS problem, there is a structure of $O(n \log^2 n)$ space that can answer a weighted sampling query in $O(Pred(U, w, n) + t)$ time.*

3.1 A structure with linear space and $O(\log^* n)$ query cost

In this subsection, we improve the space of our structure to linear by sacrificing the per-sample query cost.

Structure. We group the points into $m = n/s$ fat points, $G_1, \cdots, G_m$, where each fat point G_i includes $s = \log^2 n$ consecutive points. The weight of G_i is defined to be the summation of weights in G_i. Then we build the baseline structure, denoted by $\mathcal{T}$, on the fat points. Since the number of fat points is $n/s = n/\log^2 n$, the space usage is reduced to $O(n)$. Inside each fat point G_i, we bootstrap a baseline structure, denoted by $\mathcal{T}(G_i)$, for all s points contained in G_i. This takes $O(s \log^2 s) = O(\log^2 n \log^2 \log n)$ space for each fat point, and $O(n \log^2 \log n)$ space for all $n/\log n$ fat points. For each point $p_k \in G_i$, we also store $W_{pre}(p_k, G_i)$ and $W_{suc}(p_k, G_i)$, the prefix and suffix sums of point p_k in G_i, respectively. Finally, we store n global prefix sums, $W_{pre}(p_i, P)$, for $i = 1, \ldots, n$. It is easy to see the total space usage is $O(n \log^2 \log n)$.

Answering Queries. Given a query range $[a, b]$, we first find $P_{pre}(a)$, the predecessor of a and $P_{suc}(b)$, the successor of b in P, in $\text{Pred}(U, w, n)$ time. Then we locate the fat points $G_{a'}$ and $G_{b'}$ that contains $P_{pre}(a)$ and $P_{suc}(b)$, respectively. Figure 2 illustrates that $W(a, b)$ can be decomposed into the summation of partial weights in fat leaves $G_{a'}$ and $G_{b'}$, and weights of subtrees in canonical sets $\mathcal{R}(u, G_{a'})$ and $\mathcal{L}(u, G_{b'})$. More precisely, we have

$$W(a, b) = W_{suc}(P_{pre}(a), G_{a'}) + W_{pre}(P_{suc}(b), G_{b'}) + W(\mathcal{R}(u, G_{a'})) + W(\mathcal{L}(u, G_{b'})).$$

We retrieve these four weights and sample one of the weights. If $W(\mathcal{R}(u, G_{a'}))$ or $W(\mathcal{L}(u, G_{b'}))$ is selected, we sample a fat leaf G_i from $G_{a'}, \ldots, G_{b'}$ using baseline solution $\mathcal{T}$, and then sample a point p_k from G_i using the alias structure $\mathcal{A}(G_i)$. Otherwise, assume that the partial sum $W_{suc}(P_{pre}(a), G_{a'})$ is selected. We simply query the baseline structure in $\mathcal{T}(G_{a'})$ with range $[a, \infty)$ to retrieve a sample as the query result.

Analysis. To see that above sampling procedure gives the correct probability distribution, note that a point p_k in fat point $G_{a'}$ is selected if and only if the partial sum $W_{suc}(P_{pre}(a), G_{a'})$ is sampled from $W(a, b)$, and p_k is sampled from $W_{suc}(P_{pre}(a), G_{a'})$. Thus the probability is

$$\frac{w_k}{W_{suc}(P_{pre}(a), G_{a'})} \cdot \frac{W_{suc}(P_{pre}(a), G_{a'})}{W(a, b)} = \frac{w_k}{W(a, b)}.$$

On the other hand, consider a point p_k in fat point G_i that lies completely in (a, b). Without loss of generality, we assume G_i is in left canonical set $\mathcal{R}(u, G_{a'})$ of the baseline structure $\mathcal{T}$. Observe that p_k is selected if and only if the following events happen: 1. $W(\mathcal{R}(u, G_{a'}))$ is selected from $W(a, b)$; 2. $W(G_i)$ is selected from alias structure $\mathcal{A}(\mathcal{R}(u, G_{a'}))$; 3. p_k is selected from alias structure $\mathcal{A}(G_i)$. Thus the probability for p_k being picked can be computed as

$$\frac{W(\mathcal{R}(u, G_{a'}))}{W(a, b)} \cdot \frac{W(G_i)}{W(\mathcal{R}(u, G_{a'}))} \cdot \frac{w_k}{W(G_i)} = \frac{w_k}{W(a, b)}.$$

Bootstrap. Now that we have a structure that uses $O(n \log^2 \log n)$ space and answers weighted IRS queries in $O(\text{Pred}(U, w, n) + t)$ time, we can bootstrap this structure to reduce the space usage. More precisely, we note that the extra $\log^2 \log n$ factor comes from the baseline structure in each fat point. Thus, we can group the points in a fat point into secondary fat point of size $\log^2 \log n$ and build the baseline structure in the secondary fat point to reduce the space usage to $O(n \log^2 \log \log n)$. Repeat the bootstrap process $\log^* n$ times and we will have a structure with $O(n)$ space and $O(\log^* n)$ per-sample query time. The number of predecessor queries need to be performed is $O(\log^* n)$. However, for dataset with size $O(\log \log n)$, a predecessor query can be answered in constant time, which implies that the time for performing predecessor queries is still bounded by $O(\text{Pred}U, w, n)$.

▶ **Lemma 10.** *There is a structure of $O(n)$ space that can answer a one-dimensional weighted IRS query in $O(\text{Pred}(U, w, n) + t \log^* n)$ time.*

3.2 A structure with linear space and constant query cost.

In this subsection, we show how to obtain constant query cost by using *RAM tricks* to pack multiple integers into a single word.

Packing weights. We first apply the fat point technique twice to reduce the size of a fat point to $s = \log^2 \log n$. Note that if there is a linear size structure for s points with constant per-sample query time, we can apply it to each fat point, and achieve a linear size structure and constant per-sample query time for arbitrary number of weights.

Consider point sequence $p_1, \ldots, p_s$ with weights $w_1, \cdots, w_s$, where $s = \log^2 \log n$. If we maintain an alias structure for point sequence $\{w_i, \ldots, w_j\}$, for any pair $1 \le i \le j \le s$, then we can answer weighted IRS queries with constant time per sample. The problem is that there are $O(s^2)$ such pairs, so it requires $\Omega(s^3)$ space to store these structures.

To reduce the space cost, we round the probabilities to rational numbers with precision up to $1/s^2$, and pack multiple rational numbers into a single word. While constructing $O(s^2)$ alias structures for real weights is costly, constructing $O(s^2)$ alias structures for rational weights can be made space-efficient.

More precisely, given index pair $1 \le i \le j \le s$, let $r_k = w_k/W(p_i, p_j)$, $k = i, \ldots, j$, be the probability that p_k is sampled from $\{p_i, \ldots, p_j\}$, and let r_{k^*} denote the maximum probability in $\{r_i, \ldots, r_j\}$. We *conceptually* perform the following probability transfers: for every index $\ell \in [i, j]$, $\ell \neq k^*$, we define *rational probability* $r'_\ell = \lceil r_\ell s^2 \rceil / s^2$, and *deviation* $\alpha_\ell = r'_\ell - r_\ell < 1/s^2$. We then pour α_ℓ probability mass from r_{k^*} to r_ℓ, to form the rational probability r'_ℓ, for $\ell \neq k^*$. Note that the probability mass left for k^* is

$$r'_{k^*} = r_{k^*} - \sum_{j \neq i_1} \alpha_j > r_{k^*} - s \cdot \frac{1}{s^2} \ge r_{k^*} - \frac{1}{s} > 0.$$

See figure 3. Then we build an alias structure $\mathcal{A}'(i, j)$ for $r'_i, \ldots, r'_j$. Here $\mathcal{A}'$ indicates that we build the alias structure on the rational probabilities rather than on the original probabilities. The key insight for this probability transfer is that we can store each rational probability r'_i with $2 \log s$ bits, thus the alias structure $\mathcal{A}'(a, b)$ can be represented by $O(s \log s)$ bits. Over all possible pairs (i, j), this sums up to $O(s^3 \log s) = O(\log^6 \log n \log \log \log n) = o(\log n)$ bits. Thus, we only need one word to store all rational alias structures. We also record the index k^* for each pair (i, j), which takes $s^2 \cdot \log s = o(\log n)$ bits and fits in a word. Finally, we maintain all s prefix sums $W(1, p_i)$, $i = 1 \ldots, s$ and this requires $O(s)$ *real-valued* storage. It is easy to see that the structure takes $O(s)$ space.

Answering queries. We focus on query ranges of form $[p_i, p_j]$, $1 \le i \le j \le s$. Recall that s is the size of the secondary fat leaves. Note that each query visits at most two fat leaves of size s, so if we can generate a sample in constant time from ranges of form $[p_i, p_j]$, $1 \le i \le j \le s$, we can answer weighted IRS queries on n points in $O(\text{Pred}(U, w, n) + t)$ time.

Given such a query $[p_i, p_j]$, we first compute $W(p_i, p_j)$ by subtracting two prefix sums $W(p_1, p_j) - W(p_1, p_{i-1})$. Then we retrieve w_{k^*}, the maximum weight in $[p_i, p_j]$, and compute probability $r_{k^*} = w_{k^*}/W(a, b)$. We then sample an point p_k from the rational alias structure $\mathcal{A}'(a, b)$. If $k = k^*$, we return p_{k^*} as a sample. Otherwise, we compute $r_k = w_k/W(a, b)$ and roll a random dice z uniformly chosen in $[0, r'_k]$. If $z \le r_i$, we return p_k as the sample, otherwise, we return p_{k^*} as the sample.

Analysis. Since $W(p_i, p_j)$ and k^* can be supplied in constant time, the total query cost is constant. To verify the probability distribution, first consider a point $p_k \in [p_i, p_j]$, $k \neq k^*$. Observe that p_k is sampled if and only if its rational probability r'_k is sampled from $\mathcal{A}'(a, b)$, and the random dice z from $[0, r'_k]$ is smaller than r_k. The probability is $r'_i \cdot \frac{r_i}{r'_i} = r_i$. On the other hand, this also implies that k^* is returned with probability $1 - \sum_{p_k \in [p_i, p_j], k \neq k^*} r_k = r_{k^*}$. Thus the sampling probability distribution is correct, and we obtain the following theorem.

▶ **Theorem 11.** *Given a set $P = \{p_1, \cdots, p_n\}$ of n points on grid $[U]$, such that each point p_i is associated with an non-negative real-valued weight w_i, we can build a data structure of size $O(n)$, such that given a interval $[a, b]$ and a parameter t, we can extract t weighted random samples from the subset $P \cap [a, b]$ in $O(Pred(U, w, n) + t)$ time.*

4 Conclusions

In this paper we considered the range sampling queries where the goal is to store a given set of points in a data structure such that given a geometric range, a query returns a random sample of the points contained in the query range. We optimally solved some of the important cases of the problem: 3D halfspace queries for unweighted points, 3D dominance queries and 2D three-sided queries, and 1D two-sided (interval) queries for weighted points.

There are still a number of interesting open problems left to consider. For example, we have not investigated weighted 2D orthogonal queries at all. Also, while we solve three-sided and two-sided queries for the unweighted case, 2d four-sided queries for the unweighted case is still unsolved. Another direction is to consider weighted 3D halfspace queries.

References

1 http://weizhewei.com/papers/esa17-full.pdf.

2 Peyman Afshani and Timothy M. Chan. On approximate range counting and depth. *Discrete and Computational Geometry*, 42:3–21, 2009. doi:10.1007/s00454-009-9177-z.

3 Peyman Afshani and Timothy M. Chan. Optimal halfspace range reporting in three dimensions. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 180–186, 2009.

4 Peyman Afshani, Chris Hamilton, and Norbert Zeh. A general approach for cache-oblivious range reporting and approximate range counting. *Computational Geometry: Theory and Applications*, 43:700–712, 2010. preliminary version at SoCG'09.

5 Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. *Advances in Discrete and Computational Geometry*, pages 1–56, 1999.

6 Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

7 Arnab Bhadury, Jianfei Chen, Jun Zhu, and Shixia Liu. Scaling up dynamic topic models. In *Proceedings of International World Wide Web Conferences (WWW)*, pages 381–390. International World Wide Web Conferences Steering Committee, 2016.

8 Timothy M. Chan. Random sampling, halfspace range reporting, and construction of (<= k)-levels in three dimensions. *SIAM Journal of Computing*, 30(2):561–575, 2000.

9 Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. Orthogonal Range Searching on the RAM, Revisited. *arXiv preprint arXiv:1103.5510*, 2011.

10 Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. Random sampling for histogram construction: How much is enough? In *ACM SIGMOD Record*, pages 436–447. ACM, 1998.

11 Bernard Chazelle, Leonidas J. Guibas, and D. T. Lee. The power of geometric duality. *BIT Numerical Mathematics*, 25(1):76–90, 1985.

12 Robert Christensen, Lu Wang, Feifei Li, Ke Yi, Jun Tang, and Natalee Villa. Storm: Spatio-temporal online reasoning and management of large spatio-temporal data. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 1111–1116. ACM, 2015.

13 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3 edition, 2008.

14 Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. *ACM SIGMOD Record*, 26(2):171–182, 1997.

15 Xiaocheng Hu, Miao Qiao, and Yufei Tao. Independent range sampling. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 246–255. ACM, 2014.

16 Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. Reducing the sampling complexity of topic models. In *Proceedings of ACM Knowledge Discovery and Data Mining (SIGKDD)*, pages 891–900. ACM, 2014.

17 Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, pages 615–629. ACM, 2016.

18 Jiří Matoušek. Efficient partition trees. *Discrete & Computational Geometry*, 8(3):315–334, 1992.

19 Frank Olken. *Random sampling from databases*. PhD thesis, University of California at Berkeley, 1993.

20 Frank Olken and Doron Rotem. Random sampling from databases: a survey. *Statistics and Computing*, 5(1):25–42, 1995.

21 Frank Olken and Doron Rotem. Sampling from spatial databases. *Statistics and Computing*, 5(1):43–57, 1995.

22 Mihai Patrascu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

23 Edgar A. Ramos. On range reporting, ray shooting and k-level construction. In *Symposium on Computational Geometry (SoCG)*, pages 390–399, 1999.

24 Jiří Matoušek. Reporting points in halfspaces. *Computational Geometry, Theory and Applications*, 2(3):169–186, 1992.

25 Alastair J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 10(8):127–128, 1974.

26 Lu Wang, Robert Christensen, Feifei Li, and Ke Yi. Spatial online sampling and aggregation. *Proceedings of the VLDB Endowment*, 9(3), 2015.

Approximate Nearest Neighbor Search Amid Higher-Dimensional Flats*

Pankaj K. Agarwal[1], Natan Rubin[2], and Micha Sharir[3]

1 **Department of Computer Science, Duke University, Durham, NC, USA**
pankaj@cs.duke.edu
2 **Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel**
rubinnat.ac@gmail.com
3 **School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
michas@tau.ac.il

Abstract

We consider the *approximate nearest neighbor* (ANN) problem where the input set consists of n k-flats in the Euclidean $\mathbb{R}^d$, for any fixed parameters $0 \leq k < d$, and where, for each query point q, we want to return an input flat whose distance from q is at most $(1+\varepsilon)$ times the shortest such distance, where $\varepsilon > 0$ is another prespecified parameter. We present an algorithm that achieves this task with $n^{k+1}(\log(n)/\varepsilon)^{O(1)}$ storage and preprocessing (where the constant of proportionality in the big-O notation depends on d), and can answer a query in $O(\text{polylog}(n))$ time (where the power of the logarithm depends on d and k). In particular, we need only near-quadratic storage to answer ANN queries amid a set of n lines in any fixed-dimensional Euclidean space. As a by-product, our approach also yields an algorithm, with similar performance bounds, for answering *exact* nearest neighbor queries amid k-flats with respect to any polyhedral distance function. Our results are more general, in that they also provide a tradeoff between storage and query time.

1998 ACM Subject Classification E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems, I.3.5 Computational Geometry and Object Modeling

Keywords and phrases Approximate nearest neighbor search, k-flats, Polyhedral distance functions, Linear programming queries

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.4

1 Introduction

Nearest neighbor search is one of the most fundamental problems in computational geometry and has been studied extensively in many different fields, including computational geometry,

* Work by P. A. and M. S. was supported by Grant 2012/229 from the U.S.-Israel Binational Science Foundation. Work by P. A. was also supported by NSF under grants CCF-11-61359, IIS-14-08846, and CCF-15-13816, and by an ARO grant W911NF-15-1-0408. Work by N. R. was supported by grant 1452/15 from Israel Science Foundation by grant 2014384 from the U.S.-Israeli Binational Science Foundation, and by Ralph Selig Career Development Chair in Information Theory; the project leading to this application has received funding from European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No. 678765. Work by M. S. has also been supported by Grant 892/13 from the Israel Science Foundation, by the Blavatnik Research Fund in Computer Science at Tel Aviv University, by the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11), and by the Hermann Minkowski-MINERVA Center for Geometry at Tel Aviv University.

databases, machine learning, and data mining; see [4, 11] for comprehensive surveys. The very basic scenario, referred to as the *post-office problem* in [23], asks to preprocess a collection S of n points in $\mathbb{R}^d$ (called *sites*), where d is a fixed parameter, into a data structure, so that the site in S nearest to a query point $q \in \mathbb{R}^d$, i.e., the site $\mathsf{NN}(q, S) = \arg\min_{s \in S} \mathsf{dist}(q, s)$, where $\mathsf{dist}(\cdot, \cdot)$ is the Euclidean distance, can be reported quickly.[1] This basic version has been extended in numerous ways over the last four decades. Most notably, in such extensions the sites and/or the queries can be chosen from richer families of geometric objects (say, lines, k-flats, or even convex polyhedra, not to mention curved objects like balls), and $\mathsf{dist}(\cdot, \cdot)$ can be another distance function, such as an l_p-norm, a polyhedral distance function, or the Hausdorff distance (for non-point sites or queries) [4, 11]. The best known solution for the post-office problem requires roughly $n^{\lceil d/2 \rceil}$ storage in the worst case, for answering queries in $O(\log n)$ time, in any fixed dimension $d \geq 2$. The extended versions of the problem, for non-point sites and/or for other metrics/distance-functions, are naturally even more challenging. In the search for more efficient data structures, we therefore give up the goal of finding the exact nearest neighbor, and settle for structures that can answer efficiently *approximate nearest neighbor* (or, shortly, ANN) queries. That is, given a prespecified error parameter $\varepsilon > 0$, an ε-ANN query returns a site $s \in S$ satisfying $\mathsf{dist}(q, s) \leq (1 + \varepsilon)\mathsf{dist}(q, \mathsf{NN}(q, S))$. In what follows we use $\mathsf{ANN}(q, S)$ to denote the set of all sites with this property, i.e.,

$$\mathsf{ANN}(q, S) = \{s \mid \mathsf{dist}(q, s) \leq (1 + \varepsilon)\mathsf{dist}(q, \mathsf{NN}(q, S))\}.$$

This paper focuses on the scenario in which S is a collection of n k-flats lying in the Euclidean space $\mathbb{R}^d$, of any fixed dimension $d > k$ (where d and k are treated as constants), and the queries are points $q \in \mathbb{R}^d$. For a point $q \in \mathbb{R}^d$ and a site $s \in S$ (assumed to be closed), $\mathsf{dist}(q, s)$ is the minimum Euclidean distance between q and a point of s, i.e., $\mathsf{dist}(q, s) = \min_{p \in s} \mathsf{dist}_{p \in s}(q, p)$. Given S and a parameter $\varepsilon > 0$, the goal is to preprocess S into a data structure so that, for any query point $q \in \mathbb{R}^d$, a k-flat $s \in \mathsf{ANN}(q, S)$ can be reported quickly.

Related work. As mentioned above, nearest-neighbor (NN) searching, especially when the input sites are points, has been studied extensively. It is beyond the scope of this paper to review all the related work on nearest-neighbor searching, so we focus on the most relevant work, and refer the reader to [4, 11, 30] for more comprehensive reviews.

The most obvious approach to answering NN queries is to construct the *Voronoi diagram* of the set S of input objects, and perform point location in the diagram with the query point. Recall that the Voronoi diagram of S is the decomposition of space into cells, where each cell, associated with one of the input sites, consists of all points whose nearest site in S is that site. It is well known that the complexity of the Euclidean Voronoi diagram of a set of n points in $\mathbb{R}^d$ is $\Theta(n^{\lceil d/2 \rceil})$ in the worst case. Better bounds on the complexity of the diagram are known, though, in some special cases. For example, the expected complexity of the Voronoi diagram of a set of n random points chosen uniformly in $[0, 1]^d$ is linear; see [19].

Recently, there has been some work on Voronoi diagrams of non-point sites. For example, Chew *et al.* [17] have shown that the complexity of the Voronoi diagram of a set of n lines in $\mathbb{R}^3$ under the polyhedral metric (or distance function) defined by a convex polytope Q of constant complexity (see Section 2 for the definition of polyhedral distance functions) is $O(n^2\alpha(n)\log n)$, where the constant of proportionality depends on the complexity of

[1] The site $\mathsf{NN}(q, S)$ is uniquely defined, unless q belongs to a set of measure zero (namely, if q lies on the boundaries of two or more cells in the Voronoi diagram of S).

Q. The near-quadratic bound was subsequently extended to the case when the input sites are constant-complexity convex polyhedra in $\mathbb{R}^3$ [25]. It is an open question whether the complexity of the Euclidean Voronoi diagram of a set of lines in $\mathbb{R}^3$ is nearly quadratic; so far, the near-quadratic upper bound has been confirmed only for lines with constantly many orientations [24]. See the book by Aurenhammer *et al.* [13] for comprehensive studies of Voronoi diagrams.

Because of the potentially large complexity of the Voronoi diagram, there has also been work on constructing a data structure for answering NN queries directly, that does not require the construction of the Voronoi diagram. For example, an NN query amid a set of n points in $\mathbb{R}^d$ can be answered in $\widetilde{O}(n^{1-1/\lceil d/2\rceil})$ time using linear space.[2] More generally, for a given parameter $n \leq m \leq n^{\lceil d/2\rceil}$, a query can be answered in $\widetilde{O}(n/m^{1/\lceil d/2\rceil})$ time using $O(m)$ space. The known lower-bound results on range searching [2] suggest that this is the best bound one can hope for.

Consequently, attention has focused on answering ANN queries, as described above (see, e.g., [7, 12, 15, 20, 22], to name a few works that follow this paradigm). Earlier methods for answering ANN queries stored the input points in a (compressed) quad tree, k-tree, or their variants, and performed a spiral search to return a point in $\mathsf{ANN}(q, S)$ for a given query point q; see, e.g., [21]. More recently, the notion of an *approximate Voronoi diagram* (AVD for short) has been introduced; similar to a Voronoi diagram, AVD is a decomposition of space into cells, each associated with a site s, so that s is an approximate nearest neighbor for all query points in that cell. Har-Peled [20] constructed an approximate Voronoi diagram (AVD) of a set of n points in $\mathbb{R}^d$ of size $\widetilde{O}\left(\frac{1}{\varepsilon^d}n\right)$. Another AVD was proposed by Arya, Malamitos and Mount [8, 9]; its size is linear in n, and it can be constructed in near-linear time.

A more elaborate approach yields a data structure for ANN queries that can answer an ε-ANN query in $O(\log(n/\varepsilon))$ time using $O(n/\varepsilon^{d/2})$ space; more generally, for a parameter $\log\frac{1}{\varepsilon} \leq \theta \leq \frac{1}{\varepsilon^{d/2}\log(1/\varepsilon)}$, a query can be answered in $O(\log n + \frac{1}{\theta\varepsilon^{d/2}})$ time using $O(n\theta)$ space [7].

The performance of these and of many earlier data structures for answering ANN queries depends exponentially on d, so they are not efficient for large values of d. This has lead to extensive work on data structures for ANN-queries whose query time and size have polynomial dependence on d, most notably using the locality-sensitive-hashing (LSH) technique and its variants [3, 6]. The best-known data structure of this kind computes in $n^{7/(8c^2)+O(1/c^3)}$ time a $(c-1)$-ANN with high probability, for $c > 1$, using $n^{1+7/(8c^2)+O(1/c^3)}$ space. See [4] for a survey of higher-dimensional NN problems and techniques.

Relatively little is known about ANN-queries for non-point input sites (e.g., lines, k-flats, or even convex polyhedra); see, e.g., [5, 27, 29]. The best structures obtained for ANN-search in such extended setups are typically more expensive than those obtained for the point-to-point problem. The result of Koltun and Sharir [25] implies that an AVD for a set of n pairwise disjoint triangles in $\mathbb{R}^3$, of size $\widetilde{O}(n^2)$, can be constructed in near-quadratic time, and thus an ANN-query for this setting can be answered in $O(\log n)$ time using $\widetilde{O}(n^2)$ space. A simple grid-like construction shows that any AVD for a set of n lines in any dimension $d \geq 2$ has $\Omega(n^2)$ complexity [20], which suggests that a near-linear-size data structure with $O(\log n)$ query time is unlikely. For higher dimensions, the best known data structure for lines is by Mahabadi [27]; it answers an ε-ANN query for lines in $\mathbb{R}^d$ in $(d + \log n + 1/\varepsilon)^{O(1)}$ time, using $(n+d)^{O(1/\varepsilon^2)}$ space; see also [14, 26].

[2] Throughout this paper, we use $\widetilde{O}(f(n))$ to denote $O(f(n)\,\mathrm{polylog}(n))$.

There is also some work on the dual problem, in which the input sites are points but the query objects are k-flats. For the case when the query is a line, i.e., a 1-flat, Andoni *et al.* [5] proposed a data structure that answers an ε-ANN query in $O\left(d^3 n^{1/2+\delta}\right)$ time, using $d^2 n^{O(1/\varepsilon^2+1/\delta^2)}$ space, for any constant $\delta > 0$. Later, Mulzer *et al.* [29] proposed a data structure for the case where the query objects are k-flats. Assuming there is an ANN data structure, when both input sites and query objects are points in $\mathbb{R}^d$, with $O(n^\rho)$ query time and $O(n^\sigma)$ space, for some parameters $\rho, \sigma > 0$, their data structure answers a query with a k-flat in time $O\left(n^{k/(k+1-\rho)+\delta}\right)$, using $O\left(n^{1+\sigma k/(k+1-\rho)} + n \log^{O(1/\delta)} n\right)$ space, for any constant $\delta > 0$.

Our results. We present an efficient data structure for answering ANN-queries when the input sites are k-flats in $\mathbb{R}^d$. The main results are summarized in the following theorem.

▶ **Theorem 1.** *Let d be a constant, let $0 \leq k \leq d-1$ be an integer, let $\varepsilon > 0$ be a given error parameter, and let $\gamma = O\left((1/\varepsilon)^{\frac{d-1}{2}\min(d-k,k+1)}\right)$. For a given parameter m with $n \leq m \leq n^{k+1}$, a given set S of n k-flats in $\mathbb{R}^d$ can be preprocessed in $\widetilde{O}(\gamma m)$ expected time into a data structure of $\widetilde{O}(\gamma m)$ size, so that, for a query point $q \in \mathbb{R}^d$, a flat $f \in \mathsf{ANN}(q, S)$, with respect to the Euclidean metric, can be reported in $\widetilde{O}\left(\gamma n / m^{\frac{1}{k+1}}\right)$ time.*

In particular, in the high-storage/fast-query regime, choosing $m = n^{k+1}$, we can perform, in *any* dimension $d > k$, ANN search with $\widetilde{O}(1)$ query cost (a) amid points ($k = 0$), using a near-linear structure, or (b) amid k-flats, for $k \geq 1$, using a structure of size $\widetilde{O}\left(\gamma n^{k+1}\right)$. For $k = 1$, i.e., for lines in $\mathbb{R}^d$ ($d \geq 2$), our data structures requires storage that is nearly quadratic in n in order to answer a query in $\widetilde{O}(1)$ time. For $d = 3$, our bound nearly coincides with that obtained from the three-dimensional AVD construction of Chew *et al.* [17], but no near-quadratic data structure was known for $d > 3$.

Unlike some of the recent ANN data structures for point sites [8, 9], we do not explicitly maintain the AVD of S. Instead, we approximate the Euclidean distance by a suitable polyhedral metric (see Section 2 for the definition), and use multi-level partition trees (designed for simplex range searching) [2] to answer (exact) NN-queries amid the flats of S with respect to the approximating polyhedral metric. As a byproduct, we obtain a simple and efficient data structure for answering *exact* NN queries amid k-flats with respect to a polyhedral distance function; see Theorem 2. An advantage of this approach is that it allows a trade-off between the size of the data structure and the query time, as stated in Theorem 1. In particular, an ANN query amid k-flats can be answered in $\widetilde{O}\left(n^{1-1/(k+1)}\right)$ time with near-linear storage.

2 Warm-up: Lines in $\mathbb{R}^3$

In this section we establish Theorem 1 for a set of lines in $\mathbb{R}^3$. Let L be a set of n lines in $\mathbb{R}^3$, and let $\varepsilon > 0$ be a parameter. We wish to preprocess L into a data structure that answers efficiently queries of the form: given a point $q \in \mathbb{R}^3$, find a line $\ell \in L$ such that

$$\mathsf{dist}(q, \ell) \leq (1+\varepsilon)\mathsf{dist}(q, L), \qquad \text{where} \qquad \mathsf{dist}(q, L) := \min_{\ell' \in L} \mathsf{dist}(q, \ell'),$$

and where dist denotes the Euclidean distance.

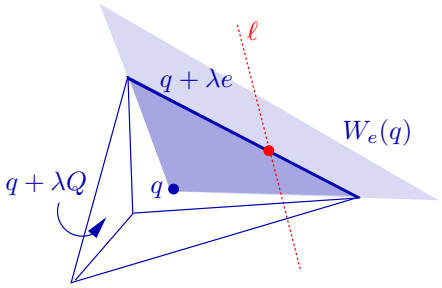


Figure 1 The Q-distance $\mathsf{dist}_Q(q, \ell)$ is the scaling factor λ for which the line ℓ touches $q + \lambda Q$, at some edge $q + \lambda e$ (and misses the interior of $q + \lambda Q$).

Polyhedral distance functions. In the general d-dimensional case, given a centrally symmetric convex polytope $Q \subset \mathbb{R}^d$, the *polyhedral distance* (with respect to Q) $\mathsf{dist}_Q(p, q)$, for a pair of points $p, q \in \mathbb{R}^d$, is defined as[3]

$$\mathsf{dist}_Q(p, q) = \sup \{t \mid q \notin p + tQ\},$$

and, more generally, for a point q and a convex object c not containing q,

$$\mathsf{dist}_Q(q, c) = \sup \{t \mid c \cap (q + tQ) = \emptyset\}.$$

The classical result of Dudley [18] implies that, for any $\varepsilon > 0$, there exists a convex polytope Q_ε, which is an intersection of $O\left(\frac{1}{\varepsilon^{(d-1)/2}}\right)$ halfspaces, or, alternatively, the convex hull of a similar number of vertices, such that $\mathsf{dist}_{Q_\varepsilon}$ approximates the Euclidean metric up to a factor of $1 + \varepsilon$; that is, for any pair of points p, q, we have

$$\mathsf{dist}(p, q) \leq \mathsf{dist}_{Q_\varepsilon}(p, q) \leq (1 + \varepsilon)\mathsf{dist}(p, q). \tag{1}$$

The advantage of using polyhedral distance functions for answering ANN-queries is that, when q is a point and f is a k-flat, $\mathsf{dist}_Q(q, f)$ can be characterized as the smallest expansion factor t for which f makes contact with some $(d-k-1)$-face of the expanding polytope $q+tQ$. This allows us to process each of the $O(1)$ $(d - k - 1)$-faces σ of Q for fast *face-shooting* queries, in which, given a query point q, we seek the smallest t for which $q + t\sigma$ hits a flat in S, and return that flat. For example, for the case of line sites in $\mathbb{R}^3$, the case studied in this section, each such query shoots a fixed segment from the query point q; the expanding segment traces a flat (two-dimensional) wedge that emanates from q and is a translate of some fixed wedge (that depends on the edge of Q that we shoot). We seek the first time at which the expanding wedge hits an input line and return that line.[4] Hence, in the general case of k-flats in $\mathbb{R}^d$, we prepare a constant number of face-shooting structures, one for each face of Q of the appropriate dimension, search with the query point q in each of them, and return the smallest expansion factor that the queries output, and the corresponding flat of S as the nearest neighbor.

In what follows we return to the special case of lines in $\mathbb{R}^3$. Given the error parameter $\varepsilon > 0$, we approximate the Euclidean unit ball by a centrally symmetric convex polytope $Q = Q_\varepsilon$, of complexity $O(1/\varepsilon)$ (using Dudley's bound). We then solve the *exact* NN problem

[3] In fact, the polyhedral metric also can be defined for non-centrally-symmetric polytopes (or, for that matter, for any compact convex body Q), but, to simplify matters in this presentation and to ensure that the distance function is a metric, we take Q to be centrally symmetric.

[4] Note that the wedge might miss the line completely, in which case we output $+\infty$.

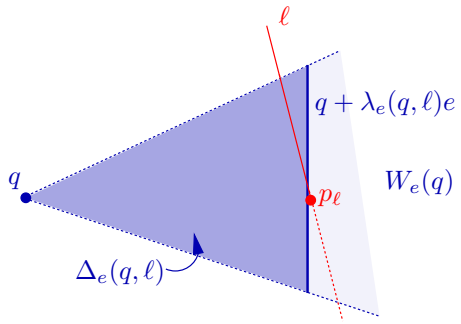


Figure 2 The wedge $W_e(q)$ is the union of all the copies $q + \lambda\Delta_e$, for $\lambda \geq 0$. $\lambda_e(q,\ell)$ is the scaling factor λ for which the triangle $q + \lambda\Delta_e$ touches ℓ at its edge $q + \lambda e$, and $\Delta_e(q,\ell)$ is that triangle. For each boundary edge e of Q, we seek the line $\ell \in L$ which minimizes the scaling distance $\lambda_e(q,\ell)$.

for the lines of L with respect to dist_Q, that is, for any query point q, the algorithm computes $\mathsf{dist}_Q(q, L)$, and returns the line of L that is nearest to q under this metric. In fact, the procedure presented next solves the exact NN problem for any convex polytope Q, not even assuming that it is centrally symmetric with respect to the origin.

Exact NN-search for L with respect to an arbitrary polytope Q. Given a point q and a line ℓ, there exists at least one edge e of Q (that depends on q and ℓ), such that the distance $\mathsf{dist}_Q(q,\ell)$ is the scaling factor λ for which (i) $q + \lambda e$ and ℓ touch one another, and (ii) ℓ does not meet the interior of $q + \lambda Q$. See Figure 1.

To decompose the problem, we consider, for each edge e of Q, the triangle $\Delta_e \subset Q$ spanned by the origin o and e. Assume with no loss of generality that no $\ell \in L$ is parallel to Δ_e.[5] Thus, for each $\ell \in L$ there exists a unique scaling factor $\lambda_e(q,\ell) \in \mathbb{R} \cup \{\infty\}$, such that the homothetic placement $q + \lambda_e(q,\ell)\Delta_e$ touches ℓ at a point of $q + \lambda_e(q,\ell)e$ (we put $\lambda_e(q,\ell) := +\infty$ when there is no such placement). We have $\lambda_e(q,\ell) < \infty$ if and only if ℓ intersects the planar wedge $W_e(q)$ which is the union of all the copies $q + \lambda\Delta_e$, for $\lambda \geq 0$. In what follows, we denote the resulting placement $q + \lambda_e(q,\ell)\Delta_e$ by $\Delta_e(q,\ell)$; see Figure 2.

As already noted, our strategy for computing $\mathsf{dist}_Q(q, L)$ is to design a separate data structure $\mathcal{D}_e$ for each edge e of Q, which answers efficiently queries of the form: Given a point q, find the smallest scaling factor $\lambda_e(q) := \min_{\ell \in L} \lambda_e(q,\ell)$, and report the corresponding line ℓ^* that attains $\lambda_e(q) = \lambda_e(q,\ell^*)$.

With this machinery available, we return to our approximating polytope Q_ε, query each of its $O(1/\varepsilon)$ edge-structures $\mathcal{D}_e$ with q, and report the minimum of the corresponding output values $\lambda_e(q)$, and the line attaining that minimum. As is easily checked, the output gives a $(1+\varepsilon)$-approximation to the Euclidean $\mathsf{dist}(q, L)$.

The edge structures $\mathcal{D}_e$. Let e be a fixed edge of Q. Given a point q, we wish to return $\lambda_e(q) := \min_{\ell \in L} \lambda_e(q,\ell)$ as well as the corresponding line ℓ_e^* that attains $\lambda_e(q) = \lambda_e(q,\ell^*)$. To simplify the presentation, and with no loss of generality, assume that e is the edge $z = 0$, $x = 1$, $-a \leq y \leq a$, for some $a > 0$.

Let us express a query in algebraic terms. Recall that we assume no line in L to be parallel to Δ_e, i.e., no line in L is normal to the z-axis. Hence, we parametrize such a line ℓ

[5] If L contains lines that are parallel to Δ_e, we apply an infinitesimally small rotation to Q which preserves all of its essential properties.

in $\mathbb{R}^3$ by the pair of equations

$$x = u_x(\ell)z + v_x(\ell), \qquad y = u_y(\ell)z + v_y(\ell),$$

for a suitable quadruple of real parameters $(u_x(\ell), v_x(\ell), u_y(\ell), v_y(\ell))$.

Let $q = (x_0, y_0, z_0)$ be the query point, and let ℓ be a line in L with the parameters $(u_x(\ell), v_x(\ell), u_y(\ell), v_y(\ell))$. Notice that $W_e(q)$ is contained in the plane $z = z_0$, and this plane meets ℓ at the point

$$p_\ell = (u_x(\ell)z_0 + v_x(\ell),\ u_y(\ell)z_0 + v_y(\ell),\ z_0).$$

The condition that p_ℓ lies in the wedge $W_e(q)$ can be expressed as

$$-a\left(u_x(\ell)z_0 + v_x(\ell) - x_0\right) \le u_y(\ell)z_0 + v_y(\ell) - y_0 \le +a\left(u_x(\ell)z_0 + v_x(\ell) - x_0\right),$$

$$\begin{aligned} \text{or} \quad & (u_y(\ell) + au_x(\ell))z_0 + (v_y(\ell) + av_x(\ell)) \ge y_0 + ax_0 \\ & (u_y(\ell) - au_x(\ell))z_0 + (v_y(\ell) - av_x(\ell)) \le y_0 - ax_0. \end{aligned} \tag{2}$$

Both constraints in (2) are linear in $u_x(\ell), v_x(\ell), u_y(\ell), v_y(\ell)$, with coefficients depending on the query q and the constant a (that is, on the edge e). Among the lines that satisfy these inequalities, our goal is to return the one that minimizes the scaling factor $\lambda_e(q, \ell)$, given by

$$\lambda_e(q, \ell) = u_x(\ell)z_0 + v_x(\ell) - x_0,$$

which is also linear in the chosen parameterization of ℓ.

In view of the above observations, we construct a three-level partition tree (see [1, 2, 16, 28]) on the lines of L. The first two levels are used to collect, for any given query point $q = (x_0, y_0, z_0)$, the lines that satisfy both conditions in (2), as the (disjoint) union of a small number of pre-stored "canonical" subsets, and the third level supports linear-programming-like queries, where each query specifies a linear objective function and asks for the point in the canonical subset that attains its minimum.

In more detail, we represent each line $\ell \in L$, parametrized by $(u_x(\ell), v_x(\ell), u_y(\ell), v_y(\ell))$, by the triple of points $p^+(\ell), p^-(\ell), p^\circ(\ell) \in \mathbb{R}^2$, where

$$\begin{aligned} p^+(\ell) &:= (u_y(\ell) + au_x(\ell),\ v_y(\ell) + av_x(\ell)) \\ p^-(\ell) &:= (u_y(\ell) - au_x(\ell),\ v_y(\ell) - av_x(\ell)), \quad \text{and} \\ p^\circ(\ell) &:= (u_x(\ell),\ v_x(\ell)), \end{aligned}$$

and put

$$P^+ := \{p^+(\ell) \mid \ell \in L\}, \qquad P^- := \{p^-(\ell) \mid \ell \in L\}, \qquad P^\circ := \{p^\circ(\ell) \mid \ell \in L\}.$$

A line ℓ satisfies (2) if and only if $p^+(\ell)$ lies above the line $z_0x + y = y_0 + ax_0$ and $p^-(\ell)$ lies below the line $z_0x + y = y_0 - ax_0$.

Following the standard methodology of multi-level data structures, each of the three levels of our partition tree, each of whose levels supports halfplane range searching queries amid points of one of the planar sets P^+, P^- or P°. This is done as follows. We fix a parameter $n \le m \le n^2$. The first level is a partition tree T, as described in [16], over the set P^+. Each node of T is associated with some *canonical subset* P_v^+. For a query halfplane h, $h \cap P^+$ can be represented as the disjoint union of $O(n/\sqrt{m} + \log n)$ canonical subsets (those stored at the nodes of T that the query with h reaches). Next, for each node v of T,

we construct a similar partition tree $T^{(v)}$, as a second-level structure, on the corresponding subset $P_v^- = \{p^-(\ell) \mid p^+(\ell) \in P_v^+\}$ of P^-. Again, each node $z \in T^{(v)}$ is associated with a suitable canonical subset $P_{z,v}^- \subset P_v^-$. Finally, at the third level, we preprocess the point set $P_{z,v}^\circ = \{p^\circ(\ell) \mid p^-(\ell) \in P_{z,v}^-\}$ into a data structure so that, for a query vector $u \in \mathbb{R}^2$, the point of $P_{z,v}^\circ$ that is minimal in direction u can be computed efficiently. Using a linear-size halfplane range reporting data structure[6] (see, e.g., [2]), such an extremal query can be answered in $O(\log n)$ time. Putting everything together, we obtain a data structure of $\widetilde{O}(m)$ size, so that, for a query point $q \in \mathbb{R}^3$, $\lambda_e(q)$, and the corresponding line $\ell_e^* \in L$, can be computed in $\widetilde{O}(1 + n/\sqrt{m})$ time. The further details, omitted here, can be found in the aforementioned papers.

Hence, for any choice of $n \le m \le n^2$, and for each of the $O(1/\varepsilon)$ edges e of Q_ε, we construct, in $\widetilde{O}(m)$ time, the data structure $\mathcal{D}_e$, as just described. This takes a total of $\widetilde{O}(m/\varepsilon)$ storage and preprocessing. Given a query point $q \in \mathbb{R}^3$, we query with q in each of these structures, and output the smallest scaling factor $\lambda_e(q)$, over all edges e, and the line $\ell \in L$ that attains this minimum. The total cost of a query is $\widetilde{O}(\frac{1}{\varepsilon}(n/m^{1/2}))$.

In particular, we can answer ANN queries amid a set of n lines in $\mathbb{R}^3$ under the Euclidean distance, in $\widetilde{O}(1)$ time using a data structure that requires only $\widetilde{O}(n^2/\varepsilon)$ storage and preprocessing time.

3 Proof of Theorem 1

The preceding algebraic approach can be extended, in a fairly straightforward manner, to nearest-neighbor problems involving k-flats, for $k \ge 1$, in $\mathbb{R}^d$, for $d \ge 3$ (and $d > k$). This is done as follows.

Let F be a collection of n k-flats, in general position, in $\mathbb{R}^d$, for some fixed $d > k \ge 1$ and $d \ge 3$. We approximate the Euclidean unit ball by a fixed convex polytope $Q = Q_\varepsilon$, which is centrally symmetric with respect to the origin o, so that the resulting Q-distance function d_Q, satisfies (1). By Dudley's theorem [18], this can be achieved by a polytope Q_ε that has either $O\left(1/\varepsilon^{\frac{d-1}{2}}\right)$ vertices, or $O\left(1/\varepsilon^{\frac{d-1}{2}}\right)$ facets.

As in Section 2, we next present a solution of the (exact) NN search problem for F with respect to the Q-distance function dist_Q, for an arbitrary fixed convex polytope Q, not even requiring it to be centrally symmetric.

Exact nearest neighbor search with respect to Q. For a k-flat $f \in F$ and a point q, the distance $\lambda = \mathsf{dist}_Q(q, f)$ is attained at a point $v \in f$ such that v lies on a $(d-k-1)$-face of $q + \lambda Q$. Thus, in complete analogy to the preceding treatment, we construct, for each $(d-k-1)$-face σ of Q, a data structure that supports queries of the form: given a query point q, find the smallest λ such that $q + \lambda\sigma$ touches a flat of F.

By triangulating Q, if necessary, we may assume that σ is a simplex. Let E_σ be the $(d-k)$-dimensional affine space spanned by o and σ, and let $K_\sigma := \bigcup_{\lambda \ge 0} \lambda\sigma$ be the $(d-k)$-dimensional wedge contained in E_σ.

The region $K_\sigma(q) := q + K_\sigma = \bigcup_{\lambda \ge 0}(q + \lambda\sigma)$ is a $(d-k)$-dimensional simplicial wedge whose (also $(d-k)$-dimensional) affine hull $E_\sigma(q)$ is $q + E_\sigma$. Assuming general position, each flat f of F intersects $E_\sigma(q)$ at a unique point, denoted as $f_\sigma(q)$.

[6] In this very special case, the structure is simply the convex hull of the underlying set.

For each $(d-k)$-face σ of Q, we collect those points $f_\sigma(q)$ that lie in $K_\sigma(q)$, and choose among them a point that minimizes the scaling factor $\lambda_\sigma(q) = \lambda_\sigma(q, f)$ at which $q + \lambda\sigma$ touches the point $f_\sigma(q)$. As in Section 2, this is done by constructing a separate data structure $\mathcal{D}_\sigma$ for each $(d-k-1)$-face σ of Q.

The face structures $\mathcal{D}_\sigma$. Without loss of generality, assume that the coordinate system is such that the linear subspace E_σ spanned by σ is the $x_1x_2\cdots x_{d-k}$-space $\mathbb{R}^{d-k}$ (given by $x_{d-k+1} = x_{d-k+2} = \cdots = x_d = 0$). Regard K_σ as the intersection of $d-k$ fixed halfspaces (through the origin) $h_1^+, h_2^+, \ldots, h_{d-k}^+$ within $\mathbb{R}^{d-k}$, and write $h_j^+ = \{x \in E_\sigma \mid x \cdot u_i \geq 0\}$, for fixed respective vectors $u_1, \ldots, u_{d-k}$ in E_σ.

We now cast the preceding observations in algebraic form. In general position, each k-flat $f \in F$ can be expressed by $d-k$ linear equations of the form

$$x_i = \sum_{j=1}^{k} a_{ij}(f)x_{d-k+j} + b_i(f), \tag{3}$$

for $i = 1, \ldots, d-k$. Let $A(f)$ denote the $(d-k)\times k$ matrix of the coefficients $a_{ij}(f)$, and let $b(f)$ denote the $(d-k)$-dimensional vector $(b_1(f), \ldots, b_{d-k}(f))$.

For each flat $f \in F$, the condition that $f_\sigma(q)$ lies in $K_\sigma(q)$ is equivalent to the condition that $f_\sigma(q) - q$ lies in K_σ (within E_σ). The point $f_\sigma(q)$ is obtained by substituting in (3) the last k coordinates of q. To simplify the notation, add the vector $b(f)$ as a last column of $A(f)$ (and continue to denote the matrix as $A(f)$). Then $f_\sigma(q) = A(f)q^*$, where q^* is the $(k+1)$-dimensional vector whose first k coordinates are the last k coordinates of q, and whose last coordinate is 1.

Hence, the condition that $f_\sigma(q) - q$ lies in K_σ is

$$u_j^T(A(f)q^* - q) \geq 0, \qquad \text{for } j = 1, \ldots, d-k. \tag{4}$$

Let u_{d-k+1} denote the outward normal of σ within E_σ. In analogy with Section 2, we construct a $(d-k+1)$-level partition tree, whose first $d-k$ levels are used to collect the set F_q of k-flats f that satisfy (4), and whose bottommost level is used to determine the flat $f \in F_q$ that minimizes the linear function $f_\sigma(q) \cdot u_{d-k+1} = u_{d-k+1}^T A(f) \cdot q^*$ in F_q.

Notice that the intrinsic dimension at each level is only $k+1$, as we represent each $f \in F$ by the $d-k+1$ $(k+1)$-dimensional vectors:

$$c_j(f) = u_j^T A(f), \quad \text{for } j = 1, \ldots, d-k+1.$$

Since the vectors u_j, for $1 \leq j \leq d-k+1$, are fixed, each vector $c_j(f)$ is a linear expression in $A(f)$, independent of the query q.

We thus prepare a $(d-k+1)$-level $(k+1)$-dimensional partition tree, each of whose levels corresponds to a $(k+1)$-dimensional halfspace range-searching data structure. Again, we fix a parameter m with $n \leq m \leq n^{k+1}$, and construct a partition tree of size $O(m)$ in $O(m \log m)$ expected time, using Chan's algorithm [16] over the set $\{c_1(f) \mid f \in F\} \subset \mathbb{R}^{k+1}$. Suppose we have constructed $j-1$ levels of the data structure, for $2 \leq j \leq d-k$. For each canonical subset F' of the $(j-1)$-level of the data structure, we construct a partition tree, using Chan's algorithm, over the set $\{c_j(f) \mid f \in F'\} \subset \mathbb{R}^{k+1}$. Finally, for each canonical node F' of level $d-k$, we again construct a partition tree on the point set $\{c_{d-k+1}(f) \mid f \in F'\} \subset \mathbb{R}^{k+1}$ so that, for a query vector $u \in \mathbb{R}^{k+1}$, the minimal point in direction u, i.e., $f_u = \arg\min_{f \in F'} c_{d-k+1}(f) \cdot u$ is returned. The overall preprocessing time and size of the data structure are $\widetilde{O}(m)$ [2, 16].

Answering queries. Given a query point q, we query with q, for each $(d-k-1)$-face σ of Q, the corresponding structure $\mathcal{D}_\sigma$, so as to find the flat $f \in F$ that satisfies (4) and (among all such flats) attains the minimum value $c_{d-k+1}(f) \cdot q^* - u_{d-k+1} \cdot q$.

Specifically, at each level $1 \le j \le d-k$, we access each of its structures, built over the canonical sets that the query retrieved at the preceding level $j-1$, and query it with the halfspace $c_j(f) \cdot q^* \ge u_j \cdot q$. As a result, after accessing all levels $j = 1, \ldots, d-k$, we obtain a compact representation of the above set F_q of flats $f \in F$ that satisfy (4), as a union of canonical sets that are stored at the $(d-k)$-level. We then query, for each of these canonical sets $F' \subseteq F_q$, its $(d-k+1)$-level structure, so as to find the flat $f \in F'$ that minimizes the objective function $c_{d-k+1}(f) \cdot q^* - u_{d-k+1} \cdot q$, and return the flat f_σ that attains the overall minimum value, along with that value, which is in fact equal to $\lambda_\sigma(q) = \lambda_\sigma(q, f_\sigma)$, as defined above. Note that f_σ exists if and only if (4) is satisfied for at least one $f \in F$. If this process has failed to find any flat $f \in F_q$, we make f_σ undefined, and return $\lambda_\sigma(q) = +\infty$. Nevertheless, there always exists at least one $(d-k-1)$-face σ of Q for which f_σ exists, so at least one of the output values $\lambda_\sigma(q, f_\sigma)$ will be finite.

We iterate this process for each $(d-k-1)$-face σ of Q, and return the flat f_σ with the minimum corresponding scaling factor $\lambda_\sigma(q, f_\sigma)$; as just observed, this minimum is always finite, so the output flat is always well defined (and is unique for a generic query q).

Using the standard results on multi-level partition-trees and on halfspace range searching [2, 16, 28], the overall size and preprocessing time of the data structure are $\widetilde{O}(m)$ and a query can be answered in $\widetilde{O}(n/m^{1/d})$ for every face of Q. Summing this bound over all faces of Q, we obtain the following general result for exact NN-search with respect to a polyhedral distance functions.

▶ **Theorem 2.** *Let $d \ge 2$ be a constant, let $0 \le k \le d-1$, let Q be a convex polytope in $\mathbb{R}^d$ with γ faces of dimension $d-k-1$. For a given parameter m with $n \le m \le n^{k+1}$, a given set F of n k-flats in $\mathbb{R}^d$ can be preprocessed in $\widetilde{O}(\gamma m)$ expected time into a data structure of $\widetilde{O}(\gamma m)$ size, so that, for a query point $q \in \mathbb{R}^d$, a flat $f \in F$ that attains the smallest Q-distance $d_Q(q, f)$ can be reported in $\widetilde{O}\left(\gamma\left(n/m^{\frac{1}{k+1}}\right)\right)$ time.*

Back to Euclidean ANN-search. We now apply the machinery just derived to obtain an efficient solution to the Euclidean ANN-search problem. Given $\varepsilon > 0$, we take a convex centrally symmetric polytope Q_ε that approximates the Euclidean ball, in the sense that its corresponding Q_ε-distance function satisfies (1). Recall that, using Dudley's bound, we can take Q_ε to have either $O\left((1/\varepsilon)^{\frac{d-1}{2}}\right)$ vertices or $O\left((1/\varepsilon)^{\frac{d-1}{2}}\right)$ facets.

The maximum number γ of $(d-k-1)$-faces of such a polytope Q_ε satisfies

$$\gamma = O\left((1/\varepsilon)^{\frac{d-1}{2}\min(d-k,k+1)}\right).$$

In this bound, we use a polytope Q_ε with a small number of facets (resp., vertices) when $k+1 \le d-k$ (resp., $k+1 > d-k$).

Plugging this into Theorem 2 finally yields Theorem 1. □

4 Discussion and Open Problems

Our data structure answers ANN queries amid a set F of k-flats in $\mathbb{R}^d$ by answering exact NN queries amid F with respect to a suitable polyhedral Q-metric. The most obvious direction towards further improving the bounds of Theorem 1 is to replace the exact NN-search under the Q-norm by some approximate version. Ideally, this would allow us to avoid the use of the

fairly expensive halfspace range searching structures. Unfortunately, our parametrization of k-flats by $(k+1)$-dimensional points does not preserve distances, so the existing machinery of approximate range searching, such as in [10], does not directly apply.

The most interesting instance of the problem involves lines in $\mathbb{R}^d$. Notice that our fast structure, using only nearly quadratic storage in n, does not yield an approximate Voronoi diagram whose description complexity is also nearly quadratic. A challenging open problem is whether an approximate Voronoi diagram of near-quadratic size exists for a set of lines in $\mathbb{R}^d$, for $d > 3$. More generally, does an approximate Voronoi diagram of size $O(n^{k+1})$ exist for a set of k-dimensional flats in $\mathbb{R}^d$, for $d > k$?

Acknowledgement. The authors thank Sariel Har-Peled for helpful discussions. We also thank ESA referees for the helpful suggestions which will be incorporated into the full version of the paper.

References

1 Pankaj K. Agarwal. Geometric range searching. In J. E. Goodman, J. O'Rourke, and Cs. D. Tóth, editors, *Handbook of Discrete and Computational Geometry*. CRC Press LLC, Boca Raton, FL, 3rd edition, 2017 (to appear).

2 Pankaj K. Agarwal. Simplex range searching. In M. Loebl, J. Nešetril, and R. Thomas, editors, *Journey Through Discrete Mathematics*. Springer, Heidelberg, to appear.

3 Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008. doi:10.1145/1327452.1327494.

4 Alexandr Andoni and Piotr Indyk. Nearest neighbors in high-dimensional spaces. In J. E. Goodman, J. O'Rourke, and Cs. D. Tóth, editors, *Handbook of Discrete and Computational Geometry*. CRC Press LLC, Boca Raton, FL, 3rd edition, 2017 (to appear).

5 Alexandr Andoni, Piotr Indyk, Robert Krauthgamer, and Huy L. Nguyen. Approximate line nearest neighbor in high dimensions. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'09, pages 293–301, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=1496770.1496803.

6 Alexandr Andoni, Piotr Indyk, Huy L. Nguyen, and Ilya Razenshteyn. Beyond locality-sensitive hashing. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'14, pages 1018–1028, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=2634074.2634150.

7 Sunil Arya, Guilherme D. da Fonseca, and David M. Mount. Optimal approximate polytope membership. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'17, pages 270–288, Philadelphia, PA, USA, 2017. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=3039686.3039704.

8 Sunil Arya and Theocharis Malamatos. Linear-size approximate voronoi diagrams. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'02, pages 147–155, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=545381.545400.

9 Sunil Arya, Theocharis Malamatos, and David M. Mount. Space-efficient approximate voronoi diagrams. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC'02, pages 721–730, New York, NY, USA, 2002. ACM. doi:10.1145/509907.510011.

10 Sunil Arya and David M. Mount. Approximate range searching. *Computational Geometry*, 17(3):135–152, 2000. doi:10.1016/S0925-7721(00)00022-5.

11 Sunil Arya and David M. Mount. Computational geometry: Proximity and location. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, chapter 3. Chapman and Hall/CRC, Boca Raton, FL, 2004.

12 Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, November 1998. doi:10.1145/293347.293348.

13 Franz Aurenhammer, Rolf Klein, and Der-Tsai Lee. *Voronoi Diagrams and Delaunay Triangulations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1st edition, 2013.

14 Ronen Basri, Tal Hassner, and Lihi Zelnik-Manor. Approximate nearest subspace search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(2):266–278, February 2011. doi:10.1109/TPAMI.2010.110.

15 Marshall Bern. Approximate closest-point queries in high dimensions. *Information Processing Letters*, 45(2):95–99, 1993. doi:10.1016/0020-0190(93)90222-U.

16 Timothy M. Chan. Optimal partition trees. *Discrete Comput. Geom.*, 47(4):661–690, June 2012. doi:10.1007/s00454-012-9410-z.

17 L. Paul Chew, Klara Kedem, Micha Sharir, Boaz Tagansky, and Emo Welzl. Voronoi diagrams of lines in 3-space under polyhedral convex distance functions. *Journal of Algorithms*, 29(2):238–255, 1998. doi:10.1006/jagm.1998.0957.

18 R. M. Dudley. Metric entropy of some classes of sets with differentiable boundaries. *Journal of Approximation Theory*, 10(3):227–236, 1974. doi:10.1016/0021-9045(74)90120-8.

19 Rex A. Dwyer. Higher-dimensional voronoi diagrams in linear expected time. *Discrete & Computational Geometry*, 6(3):343–367, Sep 1991. doi:10.1007/BF02574694.

20 S. Har-Peled. A replacement for voronoi diagrams of near linear size. In *Proceedings of the 42Nd IEEE Symposium on Foundations of Computer Science*, FOCS'01, pages 94–, Washington, DC, USA, 2001. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=874063.875592.

21 Sariel Har-Peled. *Geometric Approximation Algorithms*. American Mathematical Society, Boston, MA, USA, 2011.

22 Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC'98, pages 604–613, New York, NY, USA, 1998. ACM. doi:10.1145/276698.276876.

23 Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

24 Vladlen Koltun and Micha Sharir. 3-dimensionall euclidean voronoi diagrams of lines with a fixed number of orientations. *SIAM J. Comput.*, 32(3):616–642, March 2003. doi:10.1137/S0097539702408387.

25 Vladlen Koltun and Micha Sharir. Polyhedral voronoi diagrams of polyhedra in three dimensions. *Discrete & Computational Geometry*, 31(1):83–124, Jan 2004. doi:10.1007/s00454-003-2950-5.

26 Avner Magen. Dimensionality reductions in ℓ2 that preserve volumes and distance to affine spaces. *Discrete Comput. Geom.*, 38(1):139–153, July 2007. doi:10.1007/s00454-007-1329-4.

27 Sepideh Mahabadi. Approximate nearest line search in high dimensions. In *Proceedings of the Twenty-sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'15, pages 337–354, Philadelphia, PA, USA, 2015. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=2722129.2722154.

28 Jiří Matoušek. Range searching with efficient hierarchical cuttings. *Discrete Comput. Geom.*, 10(1):157–182, December 1993. doi:10.1007/BF02573972.

29 Wolfgang Mulzer, Huy L. Nguyên, Paul Seiferth, and Yannik Stein. Approximate k-flat nearest neighbor search. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, STOC'15, pages 783–792, New York, NY, USA, 2015. ACM. doi:10.1145/2746539.2746559.

30 Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk. *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice (Neural Information Processing)*. The MIT Press, 2006.

Output Sensitive Algorithms for Approximate Incidences and Their Applications*

Dror Aiger[1], Haim Kaplan[2], and Micha Sharir[3]

1 **Google Inc., Mountain View, CA, USA**
aigerd@google.com
2 **Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
haimk@tau.ac.il
3 **Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
michas@tau.ac.il

Abstract

An ε-approximate incidence between a point and some geometric object (line, circle, plane, sphere) occurs when the point and the object lie at distance at most ε from each other. Given a set of points and a set of objects, computing the approximate incidences between them is a major step in many database and web-based applications in computer vision and graphics, including robust model fitting, approximate point pattern matching, and estimating the fundamental matrix in epipolar (stereo) geometry.

In a typical approximate incidence problem of this sort, we are given a set P of m points in two or three dimensions, a set S of n objects (lines, circles, planes, spheres), and an error parameter $\varepsilon > 0$, and our goal is to report all pairs $(p, s) \in P \times S$ that lie at distance at most ε from one another. We present efficient output-sensitive approximation algorithms for quite a few cases, including points and lines or circles in the plane, and points and planes, spheres, lines, or circles in three dimensions. Several of these cases arise in the applications mentioned above. Our algorithms report all pairs at distance $\leq \varepsilon$, but may also report additional pairs, all of which are guaranteed to be at distance at most $\alpha\varepsilon$, for some problem-dependent constant $\alpha > 1$. Our algorithms are based on simple primal and dual grid decompositions and are easy to implement. We note that (a) the use of duality, which leads to significant improvements in the overhead cost of the algorithms, appears to be novel for this kind of problems; (b) the correct choice of duality in some of these problems is fairly intricate and requires some care; and (c) the correctness and performance analysis of the algorithms (especially in the more advanced versions) is fairly non-trivial. We analyze our algorithms and prove guaranteed upper bounds on their running time and on the "distortion" parameter α.

1998 ACM Subject Classification F.2.2 Geometrical problems and computations

Keywords and phrases Approximate incidences, near-neighbor reporting, duality, grid-based approximation

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.5

* Work by Haim Kaplan has been supported by Grant 1161/2011 from the German-Israeli Science Foundation and by Grant 1841-14 from the Israel Science Foundation. Work by Micha Sharir has been supported by Grant 2012/229 from the U.S.-Israel Binational Science Foundation, by Grant 892/13 from the Israel Science Foundation, by the Blavatnik Research Fund in Computer Science at Tel Aviv University, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University. Both Kapalan and Sharir have also been supported by the Israeli Centers for Research Excellence (I-CORE) program (center no. 4/11).

1 Introduction

Approximate incidences. Given a finite point set S_1 and finite set S_2 of geometric primitives (e.g., lines, planes, circles, or spheres in $\mathbb{R}^2$ or $\mathbb{R}^3$), and some $\varepsilon > 0$, we define the set of *ε-incidences* (also referred to as *ε-approximate incidences*, or just *approximate incidences*) between S_1 and S_2 to be

$$I_\varepsilon(S_1, S_2) = \{(s_1, s_2) \mid s_1 \in S_1, s_2 \in S_2, \mathsf{dist}(s_1, s_2) \leq \varepsilon\},$$

where $\mathsf{dist}(s_1, s_2) = \inf\{\mathsf{dist}(s_1, y) \mid y \in s_2\}$ is the Euclidean distance between s_1 and s_2. We are interested in efficient algorithms for computing $I_\varepsilon(S_1, S_2)$, ideally in time linear in $|S_1| + |S_2| + |I_\varepsilon(S_1, S_2)|$. Most classical work in discrete and computational geometry is focused on exact incidences ($\varepsilon = 0$). When S_2 is a set of lines in the plane and $\varepsilon = 0$, detecting whether $I_0(S_1, S_2)$ is empty or not is the well studied *Hopcroft's problem* (see, e.g., [8]). In contrast, the notion of approximate incidences, as we define here, probably received less theoretical attention, but has many important applications which we review below. We consider the problem of reporting all pairs in $I_\varepsilon(S_1, S_2)$. Our algorithms, though, can also estimate $|I_\varepsilon(S_1, S_2)|$, rather than report its members, and do it faster when $|I_\varepsilon(S_1, S_2)|$ is small.

This problem can be viewed as a range searching problem. Specifically, we treat each member s_2 of S_2 as the range $s_2(\epsilon) = \{p \in R^d \mid \mathsf{dist}(p, s_2) \leq \varepsilon\}, d = 2, 3$, which is the Minkowski sum of s_2 with a disk (ball in $\mathbb{R}^3$) of radius ε (centered at the origin); thus points become disks, lines become slabs (in $\mathbb{R}^2$) or cylinders (in $\mathbb{R}^3$), circles become annuli (in $\mathbb{R}^2$) or tori (in $\mathbb{R}^3$), and so on. The goal now is to report all pairs $(s_1, s_2) \in S_1 \times S_2$ such that $s_1 \in s_2(\varepsilon)$. As mentioned, the known algorithms for such tasks have a rather large overhead. For example, when S_1 is a set of m points and S_2 is a set of n lines in the plane, i.e., the ranges $s_2(\varepsilon)$ are fixed-width slabs, the best known algorithms for solving the problem have an overhead close to $m^{2/3}n^{2/3}$, and there are matching lower bounds in certain models of computation. The overhead is larger when the objects in S_2 are of more complex shapes (e.g., arbitrary circles) or when we move to three (or higher) dimensions; see [1]. In addition, these algorithms, while interesting and sophisticated from a theoretical point of view, are a nightmare to implement in practice.

Instead, with the goal of obtaining algorithms that are really simple to implement (and therefore with good performance in practice), and that run in time linear in the input and output sizes, we adopt the approach of using *approximation schemes*, in which we still report all the pairs (s_1, s_2) that satisfy $\mathsf{dist}(s_1, s_2) \leq \varepsilon$, but are willing to report additional pairs, provided that all pairs that we report satisfy $\mathsf{dist}(s_1, s_2) \leq \alpha\varepsilon$, for some constant problem-dependent parameter $\alpha > 1$. To be more precise, assuming that the test whether $\mathsf{dist}(s_1, s_2) \leq \varepsilon$ is cheap, we can filter the reported pairs by such a test, and actually report only the pairs that pass it. The actual number of pairs that we have to inspect will typically be larger than $|I_\varepsilon(S_1, S_2)|$, but it will always be at most $|I_{\alpha\varepsilon}(S_1, S_2)|$ (and in practice considerably less than that), and the hope is that the number of inspected pairs will not be much larger than those that we actually report. (We expect it to be larger by only a constant factor, which depends on α and on the geometry of the setup under consideration.)

Our results. We present simple and efficient output-sensitive algorithms (in the above sense) for approximate-incidence reporting problems between points and various simple geometric shapes, in two and three dimensions.

To calibrate the merits of our solutions, we first note that these approximate incidence reporting problems can also be solved by naive grid-based algorithms, as follows. Consider,

for example, the problem of reporting approximate incidences between a set S_1 of m points and a set S_2 of n lines in the plane. We assume that all the incidences that we seek occur in the unit disk (ball in $\mathbb{R}^3$). We partition the unit disk by a uniform grid, each of whose cells is a square of side length ε. We store each point in S_1 in a bucket corresponding to the grid cell that contains it, and, for each line $\ell \in S_2$, we report all the pairs involving ℓ and the points in the grid cells that ℓ crosses, and in their neighboring cells. The running time is $O(m + n/\varepsilon + k)$, where k is the number of reported approximate incidences. Clearly, all pairs $(p, \ell) \in S_1 \times S_2$ with $\mathsf{dist}(p, \ell) \leq \varepsilon$ are reported, and each reported pair (p, ℓ) satisfies $\mathsf{dist}(p, \ell) \leq 2\sqrt{2}\varepsilon$, as is easily checked. If n is much larger than m, we can use duality (where some care is needed to preserve point-line distances), to map the points to lines and the lines to points, and thereby reduce the complexity to $O(n + m + \min\{m, n\}/\varepsilon + k)$. This method can also be applied in three dimensions, and yields the same time bounds as in the preceding primal-only approach (duality is much trickier in these situations), namely, $O(m + n/\varepsilon + k)$, when S_2 consists of one-dimensional objects (e.g., lines or circles), but the running time deteriorates to $O(m + n/\varepsilon^2 + k)$ when S_2 consists of surfaces (e.g., planes or spheres). In these latter cases (involving planes or *congruent* spheres) duality can be applied, to improve the time bound to $O(n + m + \min\{m, n\}/\varepsilon^2 + k)$.

While superficially these simple solutions might look ideal, as they are linear in m, n, and k, their dependence on ε is too naive and weak, and when m and n are large and ε small (as is typically the case in practice), the algorithms are rather slow in practice.

In this paper we address this issue, and develop a series of "primal-dual" grid-based algorithms for several approximate incidence reporting problems, that are faster than this naive scheme for suitable ranges of the parameters m, n, and ε (which cover most of the practical instances of these problems). Specifically, we present the following results. In all of them, S_1 is a set of m points, contained in the unit ball in two or three dimensions.

- **(a)** In the plane, for a set S_2 of n lines, all k approximate incidences can be reported in time $O\left(m + n + \sqrt{mn}/\sqrt{\varepsilon} + k\right)$. (The dependency of the complexity on ε is improved by a factor of $\sqrt{\varepsilon}$ compared to the naive scheme when n and m are comparable.)
- **(b)** In three dimensions, for a set S_2 of n planes, all k approximate incidences can be reported in time $O\left(m + n + \sqrt{mn}/\varepsilon + k\right)$. (The dependency of the complexity on ε is improved by a factor of ε compared to the naive scheme, when n and m are comparable.)
- **(c)** In the plane, for a set S_2 of n congruent circles, all k approximate incidences can be reported in time $O\left(m + n + \sqrt{mn}/\sqrt{\varepsilon} + k\right)$.
- **(d)** In the plane, for a set S_2 of n arbitrary circles, all k approximate incidences can be reported in time $O\left(m + n + m^{1/3}n^{2/3}/\varepsilon^{2/3} + k\right)$.
- **(e)** In three dimensions, for a set S_2 of n congruent spheres, all k approximate incidences can be reported in time $O\left((m + n)/\varepsilon + k\right)$.
- **(f)** In three dimensions, for a set S_2 of n lines, all k approximate incidences can be reported in time $O\left(m + n + m^{1/3}n^{2/3}/\varepsilon^{2/3} + k\right)$.
- **(g)** In three dimensions, for a set S_2 of n congruent circles, all k approximate incidences can be reported in time $O\left((m + n)/\varepsilon^{1/2} + m^{1/3}n^{2/3}/\varepsilon^{7/6} + k\right)$.

In Section 4, we use the algorithms in (e) and (g), to obtain an efficient algorithm to find nearly congruent triangles which is the first step in solving the approximate point pattern matching problem in $\mathbb{R}^3$.

A comparison with the naive solutions sketched above clearly shows the superiority of our technique. For example, for lines or congruent circles in the plane, assuming that $n \leq m$, our algorithms (in (a) and (c), respectively) are asymptotically faster than the naive method when $\sqrt{mn/\varepsilon} \leq n/\varepsilon$, that is, when $\varepsilon \leq n/m$, an assumption that holds in most practical applications.

To recap, we show that, by allowing to report some additional approximate incidences between pairs that are at most $\alpha\varepsilon$ apart, one can obtain substantially better bounds than the naive ones. Our methods are based on grids and on duality – they construct much coarser primal grids, and pass each subproblem, consisting of the points in a grid cell and of the objects that pass through or near that cell, to a secondary dual stage, in which another coarse grid is constructed in a suitably defined dual space. The output pairs are obtained from the cells of these secondary grids, and the gain is in the overhead, as each primal or dual object crosses much fewer grid cells than in the naive solutions. Although this primal-dual paradigm is fairly standard, its power in the approximate incidences context, as considered here, has not been demonstrated before (to the best of our knowledge). The analysis (and the particular duality one has to use) for some of the three-dimensional variants is fairly challenging, but the algorithms all remain simple to describe and to implement.

Motivation and applications. Approximate incidence reporting and counting problems arise in several basic practical applications, in computer vision, pattern recognition, and related areas. Three major applications of this sort are robust *model fitting*, *approximate point pattern matching* under rigid motions, and estimating the fundamental matrix in (stereo) *epipolar geometry*. All three problems share a common paradigm, which we first explain for model fitting. In this problem, we are given a set P of n points, say in $\mathbb{R}^3$ (typically, these are so-called *interest points*, extracted from some image or $3D$ sensors), and we want to fit objects (called *models*) from some given family, such as lines, circles, planes, or spheres, so that each model passes near (i.e., is approximately incident to) many points of P; the quality of the model is measured in terms of the number of approximately incident points. The standard approach is to construct (usually, by repeated random sampling) a sufficiently rich collection of candidate models. (For example, for line models, one can simply sample pairs of points of P, and for each pair construct the line passing through its points.) One then counts, for each candidate line, the number of approximately incident points (for some specified error parameter $\varepsilon > 0$), and reports the models that have sufficiently many such points.

Similar reductions arise in the other problems. In approximate point pattern matching, we are given two sets A, B of points, and want to find rigid motions that map sufficiently large subsets of A to sets whose (unidirectional) Hausdorff distance to B is at most ε. Here too we construct candidate rigid motions, and test the quality of each of them. For example, in the plane, we sample pairs of points from A, and find, for each sampled pair, the pairs of points of B that are nearly at the same distance. For each such pair of pairs we construct a rigid motion that maps the first pair to near the other pair, and then test the quality of each of these motions, namely, the number of points of A that lie, after the motion, near points of B. The first step can be reduced to approximate incidence counting involving circles (whose radii correspond to the distances between the sampled points of A, and which are centered at the points of B) and the points of B. In three dimensions, we need to sample triples of points of A, and for each triple a, b, c, we need to find those triples of B that span triangles that are nearly congruent to Δabc (because to determine a rigid motion in $\mathbb{R}^3$ we need to specify how it maps three (noncollinear) source points to three respective image points). This step is described in detail in Section 4.

In epipolar geometry, we have two stereo images A, B of the same scene, and we want to estimate the fundamental matrix F that best matches A to B, where a point $p \in A$ is (exactly) matched to a point $q \in B$ if $p^T F q = 0$. We construct a sample of candidate matrices, by repeatedly sampling $O(1)$ interest points from both images, and test the quality

of each matrix. To do so for a candidate matrix F, we left-multiply each point $p \in A$ by F, interpret the resulting vectors $p^T F$, for $p \in A$, as lines, and count the approximate incidences of each line with the points of B. If sufficiently many lines have sufficiently high counts, we regard F as a good fit and output it.

To recap, in each of these applications, and in other applications of a similar nature, we generate a random sample of candidate models, motions, or matrices, and need to test the quality of each candidate. Approximate incidence reporting and counting arises either in the generation step, or in the quality testing step, or in both. Improving the efficiency of these steps is therefore a crucial ingredient of successful solutions for these problems. The standard approach, used "all over" in computer vision in practice, is the RANSAC technique [6, 9], which checks in brute force each model against each point. Replacing it by efficient methods for approximate incidence counting, which is our focus here, can drastically improve the running time of these applications.

To support the claim that this is indeed the case in practice, we have conducted preliminary experiments (not reported here) with some of our algorithms, tested them on real and random data, and compared them with other existing methods. Roughly, they demonstrate that our approach is significantly faster than the other approaches. Our experiments also support our feeling that the cost of reporting more pairs than really needed (pairs that might be at most $\alpha\varepsilon$ apart, rather than just ε), is negligible compared to the cost of the other steps (in themselves much more efficient than the competing techniques). We leave the project of conducting a through experimental study for future work, and focus this paper on developing the algorithms and establishing their worst-case guarantees.

Related work. Model fitting and point pattern matching have been the focus of many studies, both theoretical and practical; see for example [2, 3, 4, 5, 7, 10, 11, 12, 14].

We first note that many of the common approaches used in practice (e.g., RANSAC for model fitting [6, 9]), reporting or counting approximate incidences between models and points is done using brute force, examining every pair of a model and a point. Some heuristic improvements have also been proposed (see, e.g., [5] and the references therein). A similar brute-force technique is commonly used for approximate point pattern matching too (e.g., in the *Alignment* method [12] and its many variants).

The use of (exact) geometric incidences in algorithms for *exact* point pattern matching is well established; see, e.g., Brass [4] for details. Similar connections have also been used for the more practical problem of *approximate* point pattern matching. Gavrilov et al. [10] gave efficient algorithms for approximate pattern matching in two and three dimensions (where the entire sets A and B are to be matched), that use algorithms for reporting approximate incidences. One of the main results in [10] is that in the plane, all pairs of points at distance in $[(1-\varepsilon)r, (1+\varepsilon)r]$ can be reported in $O(n\sqrt{r/\varepsilon})$ time, using a grid-based search. (In a way, part of the study in this paper formalizes, extends, and improves this method.)

Aiger et al. [3] proposed a method for point pattern matching in $\mathbb{R}^3$, called 4PCS (4-Points Congruent Sets), which iterates over all coplanar pairs of quadruples of points, one from A and one from B, that can be matched via an affine transformation, and then tests the quality of each pair, focusing on pairs where the transformation is rigid. This algorithm does not use approximate incidences, and assumes the existence of coplanar tuples.

In a more recent work, Aiger and Kedem [2] describe another algorithm for computing approximate incidences of points and circles, following a similar approach by Fonseca and Mount [7] for points and lines, which is better than the one of [10] for $n = \Omega(1/\varepsilon^{3/2})$, and use this for approximate point pattern matching. This algorithm has been used in Mellado

et al. [14], to reduce the running time of the 4PCS algorithm in [3] to be asymptotically linear in n and in the output size.

The method of [2, 7] provides an alternative approach to approximate incidence reporting, for the cases of points and lines or congruent circles (the analysis in [2] is rather sketchy, though). This technique runs in $O(m + n + \log(1/\varepsilon)/\varepsilon^2 + k)$ time. For the case of lines in the plane, the scheme exploits the fact that we can approximate (up to an error of $O(\varepsilon)$) all lines in the plane that cross the unit disk, by $O(1/\varepsilon^2)$ representative lines, such that if a point in the unit disk is close to a representative line ℓ, then it is also close to all the lines in the input that ℓ represents (and vice versa). Assuming, for example, that m is constant, this alternative scheme is better than our new algorithm (for these restricted scenarios) when $\sqrt{n}/\sqrt{\varepsilon} \geq 1/\varepsilon^2$, that is, when $n \geq 1/\varepsilon^3$ (we ignore the factor $\log(1/\varepsilon)$ in this calculation). (This technique seems to be extendible to three dimensions, and to surfaces, but the formal details have not yet been worked out, as far as we know.)

Paper organization. The full version of the paper presents seven algorithms for various instances of approximate incidence reporting, as listed in (a)–(g) above. Although the high-level structure of the algorithms is fairly uniform, the specific details are rather different, and each case requires careful analysis to ensure its correctness and efficiency. Working out the details, including the appropriate form of duality (which, in some cases, is rather intricate and requires extra care), the choice of the various parameters, and the analysis that makes everything work, turned out to be fairly demanding and nontrivial. Due to lack of space, this version contains full details of only the first algorithm (for points and lines in the plane), and of the last one (finding all nearly congruent triangles in $\mathbb{R}^3$), and then describes, briefly and informally, the main features of the rest.

2 Approximate incidences in point-line configurations

We consider the approximate incidences problem between a set P of m points in the unit disk B in $\mathbb{R}^2$, and a set L of n lines that cross B, with a given accuracy parameter $0 < \varepsilon \leq 1/2$.

We approximate the distance $\mathsf{dist}(p, \ell)$ by the vertical distance between $p \in P$ and $\ell \in L$, which we denote by $\mathsf{dist}_{\mathsf{v}}(p, \ell)$. For this approximation to be good, the angle between ℓ and the x-direction should not be too large. To ensure this, we partition L into two subfamilies, one consisting of the lines with positive slopes, and one of the lines with negative slopes. We fix one subfamily, rotate the plane by $45°$, and get the desired property.

Without loss of generality, we replace the unit disk B by the unit square $S = [0, 1]^2$, and apply the following two-stage partitioning procedure. First we partition S into $1/\delta_1^2$ pairwise openly disjoint smaller squares, each of side length δ_1, where δ_1 is a parameter whose exact value will be set later. See Figure 1.

Enumerate these squares as $S_1, S_2, \ldots, S_{1/\delta_1^2}$. For $i = 1, \ldots, 1/\delta_1^2$, let P_i denote the set of all points of P that lie either in S_i or in one of the two squares that are directly above and below S_i (if they exist), and let L_i be the set of all the lines of L that cross S_i. Put $m_i := |P_i|$ and $n_i := |L_i|$. We have $\sum_i m_i \leq 3m$ and $\sum_i n_i \leq 2n/\delta_1$, because each line of L crosses at most $2/\delta_1$ squares S_i.

We now apply a duality transformation to each small square S_i separately. For notational simplicity, and without loss of generality, we may assume that $S_i = [-\delta_1/2, \delta_1/2]^2$. (Technically, this means that we shift the cells by $\delta_1/2$ in both coordinate directions, so that the grid vertices now represent the centers of the cells.) We map each point $p = (\xi, \eta)$ in P_i to the line $p^* : \; y = \xi x - \eta$, and each line $\ell : \; y = cx + d$ in L_i to the point

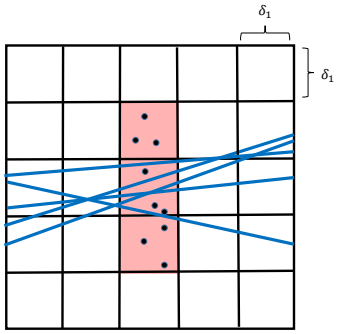


Figure 1 The partition of S into subsquares, and the subproblem associated with the middle highlighted subsquare.

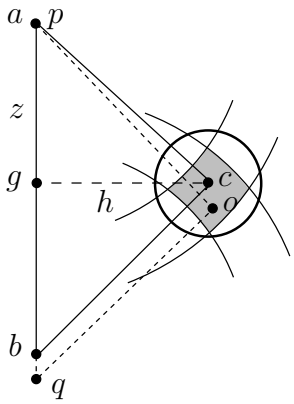


Figure 2 The reference triangle Δabc aligned with Δpqo. The shaded region is K. The circle is the a cross section of $T_{p,q}$.

$\ell^* = (c, -d)$. This duality preserves the vertical distance $\mathsf{dist_v}$ between a point p and a line ℓ; that is, $\mathsf{dist_v}(p, \ell) = \mathsf{dist_v}(\ell^*, p^*)$. Note that the slope condition ensures that $\mathsf{dist}(p, \ell) \le \mathsf{dist_v}(p, \ell) \le \sqrt{2}\mathsf{dist}(p, \ell)$.

Let $\ell : y = cx + d$ be a line in L_i, that is, ℓ crosses S_i. By the slope condition we have $-1 \le c \le 1$ and $-\delta_1 \le d \le \delta_1$, so the dual point ℓ^* lies in the rectangle $R := [-1, 1] \times [-\delta_1, \delta_1]$. Each point $p = (\xi, \eta) \in P_i$ satisfies $-\delta_1/2 \le \xi \le \delta_1/2$ and $-3\delta_1/2 \le \eta \le 3\delta_1/2$ so the coefficients of the dual line $p^* : y = \xi x - \eta$ satisfy these inequalities.

We now partition R into $1/\delta_2^2$ small rectangles, each of width $2\delta_2$ and height $2\delta_1\delta_2$, where δ_2 is another parameter that we will shortly specify. Each dual line p^* crosses at most $2/\delta_2$ small rectangles. To facilitate the following analysis, we choose δ_1, δ_2 so that they satisfy $\delta_1\delta_2 = \varepsilon$; we still have one degree of freedom in choosing them, which we will exploit later.

▶ **Lemma 1.** *For each small rectangle R', if ℓ^* is a dual point in R' and p^* is a dual line that crosses either R' or one of the small rectangles directly above or below R' (in the y-direction, if they exist), then the vertical distance $\mathsf{dist_v}(\ell^*, p^*)$ (which is the same as $\mathsf{dist_v}(p, \ell)$) is at most $5\delta_1\delta_2 = 5\varepsilon$.*

Proof. Indeed, if p^* crosses a small rectangle R'', which is either R' or one of the two adjacent rectangles, as above, then, since the slope of p^* is in $[-\delta_1/2, \delta_1/2]$, its maximum vertical deviation from R'' is at most $2\delta_2 \cdot (\delta_1/2) = \delta_1\delta_2$. Adding the heights $2\delta_1\delta_2$ of R'', and of R' when $R'' \ne R'$, the claim follows. ◀

▶ **Lemma 2.**

(a) *Let $(p, \ell) \in P \times L$ be such that $\mathsf{dist}(p, \ell) \le \varepsilon$. Let S_i be the small square containing p. If $\delta_1 \ge \varepsilon\sqrt{2}$, then ℓ must cross either S_i or one of the two squares directly above and below S_i. In other words, there exists a j such that $(p, \ell) \in P_j \times L_j$.*

(b) *Continue to assume that $\mathsf{dist}(p, \ell) \le \varepsilon$, let i be such that $(p, \ell) \in P_i \times L_i$, and let R' be the dual small rectangle (that arises in the dual processing of S_i) that contains ℓ^*. Then the dual line p^* must cross either R' or one of the two small rectangles lying directly above and below R' (in the y-direction, if they exist).*

Proof. Both claims are obvious; in (a) we use the fact that $\mathsf{dist_v}(p, \ell) \le \varepsilon\sqrt{2}$, and the assumption that $\varepsilon\sqrt{2} \le \delta_1$; see below how this is enforced. In (b) we use the fact that $\mathsf{dist_v}(p, \ell) = \mathsf{dist_v}(\ell^*, p^*)$ and that the height of a small rectangle is $2\delta_1\delta_2 = 2\varepsilon > \varepsilon\sqrt{2}$. ◀

The algorithm. We first compute, for each point $p \in P$, the square S_i it belongs to; this can be done in $O(1)$ time, assuming a model of computation in which we can compute the floor function in constant time. Similarly, we find, for each line $\ell \in L$ the squares that it crosses, in $O(1/\delta_1)$ time. This gives us all the sets P_i, L_i, in overall $O(m + n/\delta_1)$ time.

We then iterate over the small squares in the partition of S. For each such square S_i, we construct the dual partitioning of the resulting dual rectangle R into the smaller rectangles R'. As above, we find, for each dual point ℓ^*, for $\ell \in L_i$, the small rectangle that contains it, and, for each dual line p^*, for $p \in P_i$, the small rectangles that it crosses. This takes $O(n_i + m_i/\delta_2)$ time.

We now report, for each small rectangle R', all the pairs $(p, \ell) \in P_i \times L_i$ for which ℓ^* lies in R' and p^* crosses either R' or one of the small rectangles lying directly above or below R' (if they exist). We repeat this over all small squares S_i and all respective small rectangles R'. Note that a pair (p, ℓ) may be reported more than once in this procedure, but its multiplicity is at most some small absolute constant. The running time of this algorithm is

$$O\left(m + \frac{n}{\delta_1} + \sum_{i=1}^{1/\delta_1^2}\left(n_i + \frac{m_i}{\delta_2}\right) + k\right) = O\left(\frac{n}{\delta_1} + \frac{m}{\delta_2} + k\right),$$

where k is the number of pairs that we report. Lemma 1 guarantees that each reported pair is at distance $\leq 5\varepsilon$ and Lemma 2 guarantees that every pair (p, ℓ) at distance at most ε is reported.

We optimize the running time by choosing δ_1, δ_2 to satisfy $m/\delta_2 = n/\delta_1$ and $\delta_1\delta_2 = \varepsilon$. That is, we want to choose $\delta_1 = \sqrt{n\varepsilon/m}$ and $\delta_2 = \sqrt{m\varepsilon/n}$. These choices are effective, provided that both δ_1, δ_2 are at most 1, for otherwise the primal partition or the dual partitions does not exist. If $\delta_2 > 1$, that is, if $n < m\varepsilon$, we simply choose $\delta_1 = \varepsilon$, and run only the primal part of the algorithm, outputting all the pairs in $\bigcup_i P_i \times L_i$. The cost is now $O(m + n/\varepsilon + k) = O(m + k)$. (This is the naive implementation, which is now efficient since n is so small.) If $\delta_1 > 1$, we pass directly to the dual plane, flip the roles of P and L, and solve the problem in the naive manner just described, at the cost of $O(n + k)$. Otherwise (when both δ_1 and δ_2 are ≤ 1), the cost is $O\left(\sqrt{mn}/\sqrt{\varepsilon} + k\right)$. The cost of the algorithm is therefore always bounded by $O\left(n + m + \sqrt{mn}/\sqrt{\varepsilon} + k\right)$.

Recall also that in the proof of Lemma 2 we needed the inequality $\varepsilon\sqrt{2} \leq \delta_1$. This will hold when $m \leq n$ (and $\varepsilon \leq 1/2$, as we assume). In the complementary case $m > n$, we simply flip the roles of points and lines (that is, we start the analysis in the dual plane).

In conclusion, we have obtained the following main result of this section.

▶ **Theorem 3.** *Let P be a set of m points in the unit disk B in the plane, let L be a set of n lines that cross B, and let $0 < \varepsilon \leq 1/2$ be a prescribed parameter. We can report all pairs $(p, \ell) \in P \times L$, for which $\mathsf{dist}(p, \ell) \leq \varepsilon$, in time $O\left(n + m + \sqrt{mn}/\sqrt{\varepsilon} + k\right)$, where k is the actual number of pairs that we report; all pairs at distance at most ε are reported, and every reported pair lies at distance at most 5ε.*

3 Review of the other algorithms

Near neighbors in point-plane configurations. Here we are given a set P of m points in the unit ball B in $\mathbb{R}^3$, a set Π of n planes crossing B, and a prescribed error parameter $0 < \varepsilon \leq 1/2$, We solve the approximate incidences problem for P and Π with accuracy ε. As in the planar case, we approximate the point-plane distance $\mathsf{dist}(p, \pi)$ by the z-vertical distance $\mathsf{dist}_{\mathsf{v}}(p, \pi)$. We partition Π into $O(1)$ subfamilies, according to the directions of the

normals, and treat each family separately, assuming that all the normal directions in this family are close to the z-direction, making the distance approximation behave well.

We assume that $P \subset S = [0,1]^3$, and apply a two-stage partitioning, one in the primal space and one in the dual space, with a suitable choices for the corresponding parameters δ_1, δ_2, similar to the way it was done in the plane. We obtain an approximate incidence reporting algorithm that runs in $O\left(n + m + \sqrt{mn}/\varepsilon + k\right)$ time, where k is the actual number of pairs that we report.

Nearly congruent pairs in the plane. We are given two point sets P, Q, of respective sizes m and n, and parameters r, ε, and present an algorithm that reports all pairs $(p,q) \in P \times Q$ in the unit disk B, such that $|pq| \in [r-\varepsilon, r+\varepsilon]$, and each pair that it reports lie at distance in $[r-\alpha\varepsilon, r+\alpha\varepsilon]$, for some constant $\alpha > 1$. The problem is equivalent to an approximate incidences problem between P and the set of congruent circles $C := \{c_q \mid q \in Q\}$ where c_q is the circle of radius r centered at a point q. We assume that r is bounded away from 0 and that $\varepsilon \ll r \leq 1/2$.

We present two different solutions. The first one, inspired by an idea of Indyk et al. [13], does not use duality, so it is insensitive to cases where m and n differ significantly. The second solution does use duality, and is sensitive to such differences; it is more similar to the preceding solutions for the point-line and point-plane approximate incidences problems. We review here only the first solution.

We take the circle c_o of radius r centered at the origin o, and partition it into $2\pi/\sqrt{\varepsilon}$ equal canonical arcs, each with a central angle $\sqrt{\varepsilon}$. We replace each arc γ by a sector of an annulus A_γ of radii $r \pm \varepsilon$ that has γ as its 'midline', and enclose A_γ by a rectangle R_γ. Simple calculations show that the sides of R_γ are at most $\sqrt{\varepsilon} \times 3\varepsilon$.

We fix γ, and for each $q \in Q$ we translate R_γ to $R_\gamma(q) := q + R_\gamma$. We get a collection of n isothetic rectangles, and the m points of P. We tile up the unit disk by a grid whose cells are isothetic to R_γ, partition the points of P among the grid cells, and, for each $R_\gamma(q)$, report all pairs (p,q) such that p lies in one of the at most four cells that $R_\gamma(q)$ overlaps. We repeat this for each of the $O(1/\sqrt{\varepsilon})$ canonical arcs. The resulting algorithm runs in time $O\left((m+n)/\sqrt{\varepsilon} + k\right)$, where k is the actual number of pairs that we report. Our second approach, which uses duality, yields runs in $O\left(m + n + \sqrt{mn}/\sqrt{\varepsilon} + k\right)$ time, which is an improvement when m and n differ significantly.

Near-neighbor point-circle configurations. The duality-based approach can be extended to handle the approximate incidence reporting problem for points and arbitrary (rather than congruent) circles in the plane. The main difference is that general circles can be dualized into points in three dimensions, so our algorithm uses a standard grid decomposition in the primal plane, as in the cases of lines and congruent circles, but the dual partitionings take place in three dimensions, as in the case of planes.

To facilitate the second dual decomposition step, we replace the standard distance between points and circles by the *power* of a point with respect to a disk. We show that the distortion caused by this change is small, and our gain is that in the dual setup the points of P become planes (and the circles become points), so the machinery used for points and planes can be easily adapted to handle this case too. The algorithm runs in time $O\left(m + n + m^{1/3}n^{2/3}/\varepsilon^{2/3} + k\right)$, where k is the actual number of pairs that we report.

Reporting all nearly congruent pairs in three dimensions. Here we consider the three-dimensional version of the problem of nearly congruent pairs, where we are given sets P and

Q of m and n points, respectively, in the unit ball B in $\mathbb{R}^3$, and parameters $0 < \varepsilon \ll r \leq 1/2$, and wish to report all pairs $(p, q) \in P \times Q$ such that $\mathsf{dist}(p, q) \in [r - \varepsilon, r + \varepsilon]$, ensuring that each pair (p, q) that we report satisfies $\mathsf{dist}(p, q) \in [r - \alpha\varepsilon, r + \alpha\varepsilon]$, for some absolute constant $\alpha > 1$. This is an approximate incidence reporting problem between P and spheres of radius r centered at the points of Q.

As before, we have two alternative solutions, one using the technique of Indyk et al. [13], and one using duality. Both extensions are reasonably routine, although some nontrivial technical issues have to be faced when extending the techniques to three dimensions. The first approach, runs in time $O((m + n)/\varepsilon + k)$. By using duality one can get a better bound (replacing $(m + n)/\varepsilon$ by $\sqrt{mn}/\varepsilon$) when the sizes of P and Q differ substantially.

Reporting all point-line neighbors in three dimensions. Let P be a set of m points in the unit ball B in three dimensions, let L be a set of n lines that cross B, and let $\varepsilon > 0$ be a given error parameter. We present an algorithm for the approximate incidence reporting problem involving P and L.

We represent each line in $\mathbb{R}^3$ by the pair of equations $y = ax + b$, $z = cx + d$. Let ℓ be the line $y = ax + b$, $z = cx + d$, and let $p = (\xi, \eta, \zeta) \in \mathbb{R}^3$. We approximate $\mathsf{dist}(p, \ell)$ by slicing space by the plane $\pi_p : \ x = \xi$, and by computing the distance between the points p and $\ell_p := \ell \cap \pi_p = (\xi, a\xi + b, c\xi + d)$. As before, for this approximation to be good, the angle between ℓ and the x-direction should not be too large, say at most $\pi/4$, and we ensure this by partitioning L into $O(1)$ subfamilies, such that each subfamily has this property with respect to some direction u'. We focus on a single family, keep calling it L, and assume that u' is the x-axis. We show that $\mathsf{dist}(p, \ell_p) \leq \sqrt{2}\mathsf{dist}(p, \ell)$ for all $p \in P$ and $\ell \in L$.

We assume that P is contained in the unit cube $S = [0, 1]^3$, and apply the following two-stage partitioning procedure. For a pair of parameters δ_1, δ_2, whose values will be set later we partition S into $1/\delta_1^3$ pairwise openly disjoint smaller cubes, each of side length δ_1. For each small cube S_i, let P_i denote the set of all points of P that lie in S_i or in one of the (at most) eight cubes that surround S_i and have the same x-projection as S_i, and let L_i denote the set of all the lines of L that cross S_i. For each such small cube S_i, we pass to a parametric dual four-dimensional space, in which we represent each line $\ell \in L_i$, given by $y = ax + b$, $z = cx + d$, by the point $\ell^* = (a, b, c, d)$, and represent each point $p = (\xi, \eta, \zeta) \in P_i$ by the 2-plane (in $\mathbb{R}^4$) $p^* = \{(a, b, c, d) \mid a\xi + b = \eta,\ c\xi + d = \zeta\}$; p^* is the locus of all points dual to lines that pass through p.

We define the distance in the dual space between a point $\ell^* = (a, b, c, d)$ and a plane p^*, for a primal point $p = (\xi, \eta, \zeta)$, to be the distance between ℓ^* and the point $(a, \eta - a\xi, c, \zeta - c\xi)$, which is the intersection of p^* with the plane defined by $x = a$ and $z = c$. It follows that the distance between ℓ^* and p^*, as defined above, is equal to $\mathsf{dist}(p, \ell_p)$ in the primal space.

Fix a small cube S_i, and assume without loss of generality that $S_i = [0, \delta_1]^3$. Let ℓ be a line in L_i, given by $y = ax + b$, $z = cx + d$. One can show that ℓ^* lies in the box R given by $-1 \leq a,\ c \leq 1$ and $-\delta_1 \leq b,\ d \leq 2\delta_1$. We now partition R into $1/\delta_2^4$ smaller boxes, each of which is a homothetic copy of R scaled down by δ_2. Concretely, each smaller box R' is congruent to the box $[0, 2\delta_2] \times [0, 3\delta_1\delta_2] \times [0, 2\delta_2] \times [0, 3\delta_1\delta_2]$.

We then show that, for each small box R', if $\ell^* = (a_\ell, b_\ell, c_\ell, d_\ell)$ is a dual point (of some $\ell \in L_i$) in R' and p^* is a dual plane (of some point $p = (\xi, \eta, \zeta) \in P_i$) that crosses R' or one of its surrounding boxes of the same xz-range, then $\mathsf{dist}(p, \ell) \leq 8\sqrt{2}\delta_1\delta_2$. Conversely, if $\mathsf{dist}(p, \ell) \leq \delta_1\delta_2$ then (p, ℓ) belong to some subproblem $P_j \times L_j$, and p^* crosses the small dual region R' containing ℓ^* or one of its nearby regions.

The algorithm is now immediate: We compute the sets P_i, L_i, for $i = 1, \ldots, 1/\delta_1^3$, in overall $O(m + n/\delta_1)$ time. Then, for each small cube S_i, we consider the partitioning of the

resulting dual box R into the smaller boxes R'. As above, we find, for each $\ell \in L_i$, the small region that contains the dual point ℓ^*, and, for each $p \in P_i$, the small regions that the dual plane p^* crosses. We report, for each small region R', all the pairs $(p, \ell) \in P_i \times L_i$ for which ℓ^* lies in R' and p^* crosses either R' or one of the at most eight small regions that surround R' and have the same xz-range. We repeat this over all small cubes S_i and all respective small regions R'. With a suitable optimization of the values of δ_1 and δ_2, the running time is $O\left(m + n + m^{1/3}n^{2/3}/\varepsilon^{2/3} + k\right)$.

Reporting all point-circle neighbors in three dimensions. In preparation for the final algorithm, that finds all nearly congruent copies of a given triangle in a set of n points in $\mathbb{R}^3$, we first solve the following problem. Let P be a set of m points in the unit ball B in $\mathbb{R}^3$, let C be a set of n congruent circles in $\mathbb{R}^3$ of radius $r \leq 1/2$ that cross B, and let $\varepsilon \ll r$ be a prescribed error parameter. We present an efficient algorithm for the approximate incidence reporting problem for P and C.

This is perhaps the most complex algorithm in our collection. We slice each circle into canonical arcs, replace each arc by a sector of a torus of width ε around it, enclose each torus sector by a suitable (bounded) cylinder, and reduce our problem to that of reporting point-cylinder containments. We further reduce the problem by cutting space by parallel slabs of width $\sqrt{\varepsilon}$ in some suitable direction, say the x-direction, by partitioning the points of P among the slabs, and by considering only those toric/cylindrical pieces that form sufficiently small angle with the x-direction. For each such slab σ, we take the points in σ, replace each cylinder that intersects σ, or a nearby slab, by the full line that supports its axis, and run the approximate incidence reporting algorithm involving the points in the slab and the lines associated with the slab, repeating this over all slabs and tori sectors. The resulting algorithm runs in time $O\left((m+n)/\varepsilon^{1/2} + m^{1/3}n^{2/3}/\varepsilon^{7/6} + k\right)$, where k is the number of (distinct) reported pairs.

4 Reporting all nearly congruent triangles

In this section we put to work the algorithms in (e) and (g) (see Section 1), to obtain an efficient solution of the first step in solving the approximate point pattern matching problem in $\mathbb{R}^3$ (see its review in the introduction), where we are given a sampled "reference" triangle Δabc, for a triple of points a, b, c in the first set A, and a prescribed error parameter $\varepsilon > 0$. Our goal is to report all triples p, q, o in the second set B that span a triangle "nearly congruent" to Δ; that is, triples that satisfy

$$\big||pq| - |ab|\big| \leq \varepsilon, \quad \big||po| - |ac|\big| \leq \varepsilon, \quad \text{and} \quad \big||qo| - |bc|\big| \leq \varepsilon. \tag{1}$$

We allow to report triples that satisfy (1) with $\alpha\varepsilon$ on the right-hand sides rather than ε, for some fixed constant α. Let ab be the longest edge of Δ. We require that $\beta \leq |ab| \leq 1/2$ for some fixed constant β. We also require that the height h of Δ from c (perpendicular to ab) is larger than some fixed constant s. We assume that $\beta, s \gg \varepsilon$. Our approximation guarantee α increases as β and s decrease.

We first report all pairs $(p, q) \in B^2$ such that $\big||pq| - |ab|\big| \leq \varepsilon$, using the algorithm specified in (e) (incidences between congruent spheres and points). This takes $O(n/\varepsilon + N)$ time, where N is the number of pairs that we report. Let Π denote the set of reported pairs. We know that all the desired pairs are included in Π, and that every pair (p, q) in Π satisfies $\big||pq| - |ab|\big| \leq \alpha'\varepsilon$, for some absolute constant α'. We prune Π, leaving in it only pairs (p, q) satisfying $\big||pq| - |ab|\big| \leq \varepsilon$. We continue to denote the resulting set as Π, and its size by N.

Let (p, q) be a pair in Π. Any point o that satisfies $\big||po| - |ac|\big| \leq \varepsilon$ and $\big||qo| - |bc|\big| \leq \varepsilon$ lies in the intersection $K = K_{p,q}$ of two spherical shells, one centered at p with radii $|ac| \pm \varepsilon$, and one centered at q with radii $|bc| \pm \varepsilon$. The following lemma allows us to replace K by a torus that is congruent to a fixed torus that depends only on Δ. See Figure 2.

▶ **Lemma 4.** *Assume that Δ is sufficiently fat, in the sense that $\beta \leq |ab| \leq 1/2$ and $h \geq s$, for some absolute positive constants β, s that satisfy $\varepsilon \ll \beta, s$. Then there exists a circle $\gamma_{p,q}$ of radius h such that K is contained in the torus $T_{p,q}$ that is the Minkowski sum of $\gamma_{p,q}$ and a ball of radius $\varepsilon' \leq \delta\varepsilon$ around the origin, where the constant δ depends on β and s.*

Proof. Denote the lengths of the edges of the triangle Δabc by $u = |ab|$, $v = |ac|$ and $w = |bc|$. Let g the point where h meets ab and let $z = |ag|$. We have $z^2 + h^2 = v^2$ and $(u - z)^2 + h^2 = w^2$, from which we obtain that $z = \frac{u^2+v^2-w^2}{2u}$, and we denote this expression as $z = z(u, v, w)$. Consider an alignment of Δ within the plane of Δpqo, such that a coincides with p and ab overlaps pq. Let g now be a point on pq at distance z from $p = a$. Then c lies on the circle $\gamma_{p,q}$ of radius h, centered at g, and contained in the plane perpendicular to pq through g. See Figure 2(b).

Fix some point $o \in K$. We claim that o must be at distance $\leq \delta\varepsilon$ from $\gamma_{p,q}$, for some fixed constant δ that depends on β and s. Indeed, since $(p, q) \in \Pi$ and $o \in K$, we can write $|pq| = u + \varepsilon_1$, $|po| = v + \varepsilon_2$, and $|qo| = w + \varepsilon_3$, where $|\varepsilon_i| \leq \varepsilon$ for $i = 1, 2, 3$.

Consider the alignment of Δ with Δpqo, as above, and imagine that we perturb the edges ab, ac, and bc of Δ by ε_1, ε_2, and ε_3, respectively, so that Δ is continuously deformed into Δpqo. We claim that o cannot move too far as a result of this deformation so the distance between o and c must be small.

To see this, let h' be the height of Δpqo from o, let g' be the point at which h' meets pq, and let $z' = |pg'|$. We claim that $|z' - z| \leq \delta\varepsilon$ and $|h' - h| \leq \delta\varepsilon$ for some absolute constant δ. To see this, using the function $z = z(u, v, w)$ defined above, we have $z' = z(u + \varepsilon_1, v + \varepsilon_2, w + \varepsilon_3)$, and routine calculations show that, for ε sufficiently small, we have $|z' - z| = O(|\nabla z(u, v, w) \cdot (\varepsilon_1, \varepsilon_2, \varepsilon_3)|) \leq \delta'\varepsilon$, where δ' depends on β.

Similarly, by Heron's formula, we can think of h as a function $h(u, v, w)$, given by

$$h(u, v, w) = \frac{2\text{Area}(\Delta)}{u} = \frac{2\sqrt{\tau(\tau - u)(\tau - v)(\tau - w)}}{u},$$

where $\tau = \frac{1}{2}(u + v + w)$. Then $h' = h(u + \varepsilon_1, v + \varepsilon_2, w + \varepsilon_3)$, and, by another routine calculation, $|h' - h| = O(|\nabla h(u, v, w) \cdot (\varepsilon_1, \varepsilon_2, \varepsilon_3)|) \leq \delta''\varepsilon$, for another constant δ'' that depends on β and s. Take $\delta = \sqrt{(\delta')^2 + (\delta'')^2}$, and the lemma follows. ◀

We have thus reached the following scenario. We have a set $\mathcal{T}$ of N congruent tori $T_{p,q}$, for $(p, q) \in \Pi$, and a set B (the original one) of n points. By construction, each triple (p, q, o) that defines a triangle for which (1) holds, satisfies $o \in T_{p,q}$. Using our algorithm for point-circle near neighbors in $\mathbb{R}^3$, as reviewed in Section 3, we can report all the triples (p, q, o) such that $o \in T_{p,q}$, in time $O\left(n + N/\varepsilon^{1/2} + n^{1/3}N^{2/3}/\varepsilon^{7/6} + k\right)$, where k is the number of (distinct) triples that we report; each of the desired triples is reported, and each triple that we report is such that the distance from o to $\gamma_{p,q}$ is at most $\alpha\varepsilon$ for some other fixed constant $\alpha > \delta$. Therefore each triple which we report satisfies (1) with $\alpha\varepsilon$ on the right-hand sides, rather than ε. In summary, we have:

▶ **Theorem 5.** *Let B be a set of n points in the unit ball in $\mathbb{R}^3$. Let Δabc be a fixed reference triangle and let ε an error parameter, so that Δ and ε satisfy the constraints specified in Lemma 4. We can then report all triples $(p, q, o) \in B^3$ that span a triangle nearly congruent to*

Δ, *in the sense of (1), in time* $\left(n + N/\varepsilon^{1/2} + n^{1/3}N^{2/3}/\varepsilon^{7/6} + k\right)$, *where* N *is the number of pairs reported by our algorithm for approximate congruent pairs in* $\mathbb{R}^3$ *(reviewed in Section 3), applied to* P *with distance* $|ab|$*, the largest edge length of* Δ*, and* k *is the number of (distinct) triples that the algorithm in this section reports; each of the desired triples is reported, and each triple that we report satisfies (1) with* $\alpha\varepsilon$ *replacing* ε*, where* α *is a suitable absolute constant. Each pair is reported at most* $O(1)$ *times.*

References

1. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry, Contemp. Math. 223, Amer. Math. Soc. Press, Providence, RI*, pages 1–56, 1999.
2. D. Aiger and K. Kedem. Approximate input sensitive algorithms for point pattern matching. *Pattern Recognition*, 43(1):153–159, 2010.
3. D. Aiger, N. J. Mitra, and D. Cohen-Or. 4-points congruent sets for robust pairwise surface registration. *ACM Trans. Graphics*, 27(3):Article 85, 2008.
4. P. Brass. Combinatorial geometry problems in pattern recognition. *Discrete Comput. Geom.*, 28(4):495–510, 2002.
5. O. Chum and J. Matas. Optimal randomized RANSAC. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(8):1472–1482, 2008.
6. *IEEE Int'l Workshop "25 Years of RANSAC" in conjunction with CVPR'06 (RANSAC25'06)*. IEEE Computer Society, 2006.
7. G. D. da Fonseca and D. M. Mount. Approximate range searching: The absolute model. *Comput. Geom. Theory Appls.*, 43(4):434–444, 2010.
8. J. Erickson. New lower bounds for Hopcroft's problem. *Discrete Comput. Geom.*, 16(4):389–418, 1996.
9. M. A. Fischler and R. C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, 1982.
10. M. Gavrilov, P. Indyk, R. Motwani, and S. Venkatasubramanian. Combinatorial and experimental methods for approximate point pattern matching. *Algorithmica*, 38(1):59–90, 2003.
11. P. J. Heffernan and S. Schirra. Approximate decision algorithms for point set congruence. *Comput. Geom. Theory Appls.*, 1(4):137–156, 1994.
12. D. P. Huttenlocher and S. Ullman. Recognizing solid objects by alignment with an image. *Internat. J. Computer Vision*, 5(2):195–212, 1990.
13. P. Indyk, R. Motwani, and S. Venkatasubramanian. Geometric matching under noise: Combinatorial bounds and algorithms. *Proc. 10th ACM–SIAM Sympos. Discrete Algorithms*, pages 457–465, 1994.
14. N. Mellado, D. Aiger, and N. J. Mitra. Super 4pcs: fast global pointcloud registration via smart indexing. *Comput. Graphics Forum*, 33(5):205–215, 2014.

Randomized Contractions for Multiobjective Minimum Cuts

Hassene Aissi[1], Ali Ridha Mahjoub[2], and R. Ravi*[3]

1 **Univ. Paris-Dauphine, PSL Research University, CNRS, UMR 7243, LAMSADE, Paris, France**
aissi@lamsade.dauphine.fr

2 **Univ. Paris-Dauphine, PSL Research University, CNRS, UMR 7243, LAMSADE, Paris, France**
mahjoub@lamsade.dauphine.fr

3 **Carnegie Mellon University, Pittsburgh, USA**
ravi@andrew.cmu.edu

Abstract

We show that Karger's randomized contraction method [7] can be adapted to multiobjective global minimum cut problems with a constant number of edge or node budget constraints to give efficient algorithms.

For global minimum cuts with a single edge-budget constraint, our extension of the randomized contraction method has running time $\tilde{O}(n^3)$ in an n-node graph improving upon the best-known randomized algorithm with running time $\tilde{O}(n^4)$ due to Armon and Zwick [1]. Our analysis also gives a new upper bound of $O(n^3)$ for the number of optimal solutions for a single edge-budget min cut problem. For the case of $(k-1)$ edge-budget constraints, the extension of our algorithm saves a logarithmic factor from the best-known randomized running time of $O(n^{2k}\log^3 n)$. A main feature of our algorithms is to adaptively choose, at each step, the appropriate cost function used in the random selection of edges to be contracted.

For the global min cut problem with a constant number of node budgets, we give a randomized algorithm with running time $\tilde{O}(n^2)$, improving the current best determinisitic running time of $O(n^3)$ due to Goemans and Soto [5]. Our method also shows that the total number of distinct optimal solutions is bounded by $\binom{n}{2}$ as in the case of global min-cuts. Our algorithm extends to the node-budget constrained global min cut problem excluding a given sink with the same running time and bound on number of optimal solutions, again improving upon the best-known running time by a factor of $O(n)$. For node-budget constrained problems, our improvements arise from incorporating the idea of merging any infeasible super-nodes that arise during the random contraction process.

In contrast to cuts excluding a sink, we note that the node-cardinality constrained min-cut problem containing a given source is strongly NP-hard using a reduction from graph bisection.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases minimum cut, multiobjective optimization, budget constraints, graph algorithms, randomized algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.6

* This material is based upon research supported in part by the U.S. Office of Naval Research under award number N00014-12-1-1001, the U.S. National Science Foundation under award number CCF-1527032, and a visiting professorship at LAMSADE, Paris Dauphine University.

1 Introduction

Cut problems play a central role in combinatorial optimization and arise routinely in many practical areas such as telecommunications, project networks and databases [7] as well as the bottleneck computation in the separation routine for important network optimization problems such as the TSP [12]. Let $G = (V, E)$ be an undirected simple graph with n nodes and m edges, and $c^1, \ldots, c^k : E \to \mathbb{Z}^+$ ($w^1, \ldots, w^{k-1} : V \to \mathbb{Z}^+$) be k ($k-1$) non-negative cost functions defined on the set of edges (nodes), where k is a constant. A cut X in G is a subset of nodes $X \subseteq V$ such that $\emptyset \neq X \neq V$, and it determines the set $\delta(X)$ of edges with exactly one end in X. The cost of cut X in criterion j is $c^j(\delta(X)) := \sum_{e \in \delta(X)} c^j(e)$ ($w^j(X) := \sum_{v \in X} w^j(v)$). Given $k-1$ cost bounds $b_1, \ldots, b_{k-1}$, we study the following multiobjective versions of the minimum cut problem.

- Edge-budget constraints: find a cut C^* minimizing edges cost c^k subject to the constraints $c^i(\delta(C^*)) \leq b_i$ for $i = 1, \ldots, k-1$.
- Node-budget constraints: find a cut C^* minimizing edges cost c^k subject to the constraints $w^i(C^*) \leq b_i$ for $i = 1, \ldots, k-1$.
- Node-budget constraints including a source s (excluding a sink t): given a specific node $s \in V$ ($t \in V$), find a cut C^* minimizing edges cost c^k such that $w^i(C^*) \leq b_i$ for $i = 1, \ldots, k-1$, and $s \in C^*$ ($t \notin C^*$).

1.1 Previous Work

Randomized contraction: Karger [7] gave an elegant randomized contraction algorithm that finds a global minimum cut with high probability. A consequence of its probabilistic analysis is a strongly polynomial bound on the number of (near-) optimal global minimum cuts. Karger and Stein [8] improve its running time using a recursive construction that carefully traded off the probability of success with the size of the recursive subproblems. Our work builds on these methods and extends them to budgeted versions of the global minimum cut problem.

Edge-budget constraints: While most budgeted versions of standard combinatorial optimization problems are NP-hard [4], Armon and Zwick [1] give an efficient strongly polynomial time algorithm for solving the minimum cut problem with a constant number k of edge-budget constraints. Their algorithm guesses the optimal value by performing a binary search using $O(\log n)$ calls to a subproblem called the *min-max cut problem*. Here, the goal is to find a cut $\bar{C}$ for which $\max_{i=1,\ldots,k} c^i(\bar{C})$ is minimized, *i.e.*, a cut $\bar{C}$ whose largest cost is the smallest possible. This problem is in turn reduced to enumerating all cuts that are at most at factor of k larger than the global minimum cut for a single cost function. Karger and Stein [8] show that every graph contains at most $O(n^{2k})$ such cuts. In order to enumerate them, Armon and Zwick use either the $O(mn^{2k})$ deterministic algorithm of Nagamochi et al. [11] or the $O(n^{2k} \log^2 n)$ randomized algorithm of Karger and Stein [8]. Thus, their approach leads to an $O(mn^{2k} \log n)$ time deterministic algorithm and an $O(n^{2k} \log^3 n)$ time randomized one. The minimum cut problem with edge-budget constraints may be of interest in itself but also arises as a subproblem in other fields, e.g., interdiction problems. Zenklusen [16] shows the link between the problem of maximally decreasing the optimal value of the global minimum cut by removing a limited set of edges and the minimum cut problem with a single edge-budget constraint.

Node-Budget Constraints: Armon and Zwick [1] consider the problem of finding a cut of minimum cost with at most b vertices on its smaller side. This problem corresponds to a special case of the single node-budget constraint ($k = 2$) with $w(v) = 1$ for all node $v \in V$. The authors reduce this problem to the problem of minimum cut with a single edge-budget constraint and give deterministic and randomized algorithms running in $O(mn^4 \log n)$ and $O(n^4 \log^3 n)$ times respectively. Goemans and Soto [5] consider the more general problem of minimizing a symmetric submodular functions (SSF) f over a family of sets $\mathcal{I}$ that are closed under inclusion. Note that the cut function over the node set of a graph $G = (V, E)$ is a SSF. Moreover, the family of all subset of nodes $X \subseteq V$ satisfying the node-budget constraints is a typical example of sets closed under inclusion. Goemans and Soto [5] extended Queyranne's algorithm [13] (which in turn is based on the work of Nagamochi and Ibaraki [10]) in order to enumerate all the $O(n)$ minimal minimizers using $O(n^3)$ oracle calls to function f and $\mathcal{I}$. In the particular case of graphs, their result implies that the minimum cut problem with node-budget constraints can be solved in $O(n^3)$ running time. Interestingly, Goemans and Soto's algorithm does not introduce any slowdown with respect to the running time of solving the global minimum cut problem.

Cardinality Constraints: Bruglieri et al. [2] study the version of minimum cut problem where the cardinality of the edge cut must be exactly the given bound k, or at least the given bound k, and show NP-hardness via reduction from MAX-CUT. The node-cardinality constrained version on the side containing a given source has been studied by Hayrapetyan et al. [6] under the name MINSBCC (Minimum-size bounded-capacity cut): their version bounds the cost of the cut and minimizes the node cardinality of the cut. They show NP-hardness of the problem on general graphs with uniform node weights and on trees (with non-uniform node weights), and provide bicriteria approximation algorithms with ratio $(\frac{1}{\lambda}, \frac{1}{1-\lambda})$ for any $0 < \lambda < 1$. The $s - t$ separating version of this unbalanced cut problem was studied by Li and Zhang [9], and by Zhang [17]. The node-cardinality constrained version of this problem generalizes the famous graph bisection problem. For the exact version of the problem where the side containing s must have exactly k nodes, an $O(\log n)$-approximation was given by Räcke [14].

1.2 Our contributions

The main contribution of our paper is to extend the Karger's randomized contraction algorithm [7] to handle node or edge budget constraints.

Edge-budget Constraints. The original randomized contraction algorithm for a single edge cost solves the global minimum cut problem by repeatedly picking a random edge with probability proportional to its cost and contracting it, until only two vertices remain. Karger shows that with high probability the cut formed by the edges joining these (super-)nodes is a minimum cut. The key ingredient in the proof of the success probability of Karger's algorithm is that the optimal value of the minimum cut problem is at most the cost of any cut formed by a singleton node. However, if budget constraints are added to the original problem, some of these singleton cuts may be infeasible (for the budget constraint) and hence may have a cost smaller than the optimal value of the budgeted problem. On the other hand, the cost of a feasible cut formed by a singleton node is larger than the optimal value but the current graph may contain few such nodes. Our main result uses new ideas to overcome this difficulty.

▶ **Theorem 1.** *For the global minimum cut problem with a single edge-budget constraint in a graph on n nodes, a randomized contraction algorithm returns any particular optimal solution in $O(n^3 \log^4 n \log\log n)$ time with probability $1 - \frac{1}{\Omega(n)}$.*

For the case of a single edge-budget constraint, our randomized contraction algorithm decides to contract, at each step, edges based on either the budget-cost function c^1 or the objective-cost function c^2, depending on whether the number of feasible cuts (obeying the budget) formed by the current singletons is sufficiently "large" or not. This modification is crucial to ensure the high success probability of returning at the end an optimal cut, and represents our main technical contribution.

Our final algorithm for this problem is presented in Section 2 and runs with high probability in $\tilde{O}(n^3)$ time. This result improves upon the current best running time of $\tilde{O}(n^4)$ given in [1]. As a byproduct of our analysis, we save a factor of $O(n)$ from the best-known upper bound on the number of optimal solutions of this problem given in [1].

In the general case, multiple edge-budget constraints make the problem harder because the number of infeasible cuts formed by a singleton may increase. With more than two budget constraints, a cut satisfying the i^{th} budget constraint may violate the j^{th} one. Therefore, even though the number of cuts formed by a singleton node satisfying the i^{th} budget constraint may be large (the property we used with a single budget constraint), few of them may satisfy all the budget constraints. Therefore, we need a different idea to tackle multiple budget constraints. For this case, we extend Karger's algorithm [7] differently by first sampling the cost function that is then used to randomly choose an edge to be contracted. Our final algorithm (Theorem 13) saves a logarithmic factor from the best-known running time of $O(n^{2k} \log^3 n)$ given in [1].

Node-budget Constraints. We derive in Section 3 faster randomized algorithms for finding global minimum cuts with a constant number of node budget constraints.

▶ **Theorem 2.** *For the global minimum cut problem with a constant number of node-budget constraint in a graph on n nodes, a randomized contraction algorithm returns any particular optimal solution in $O(n^2 \log n)$ time with probability $\Omega(1/\log n)$. Furthermore, all the optimal solutions can be computed with high probability in $O(n^2 \log^3 n)$ time.*

For this case, we use an observation similar to that of Goemans and Soto [5]: whenever the contraction produces two (node-budget) infeasible super-nodes, we merge them into one. Adding this idea to Karger's random contraction gives an algorithm with a randomized running time of $\tilde{O}(n^2)$ (Theorem 17). This considerably improves their current best running time of $O(n^3)$ [5] even though their algorithm is deterministic. As a byproduct, we show that the total number of distinct optimal global minimum cuts in the node-budget constrained case is also bounded by $\binom{n}{2}$ as in the non-budgeted case.

Our algorithm can be adapted to the node-budget constrained global min cut problem excluding a given sink $t \in V$ with the same running time and bound on number of optimal solutions (Theorem 18). In this case, the running time of our algorithm improves upon the previous deterministic running time of Goemans and Soto by a factor of $O(n)$.

Our results indicate that for the global minimum cut problem, the node-budget constraints are easier to handle than edge-budget ones, even though both are efficiently solvable. In contrast to the above results, we note that even the node-cardinality constrained global minimum cut problem containing a given source is strongly NP-hard using a reduction from graph bisection (Theorem 19).

Algorithm 1 Random edge contraction for a single edge-budget constraint.

Input: a simple graph $G = (V, E)$ with two nonnegative edge costs c^1, c^2, a bound b_1, and integer $q \geq 10$

Output: a cut $\emptyset \neq C^* \subset V$ minimizing cost c^2 subject to edge-budget constraint $c^1(\delta(C^*)) \leq b_1$

let $E_0 \leftarrow E$, $V_0 \leftarrow V$, $G_0 \leftarrow G$, $r \leftarrow 0$
while $|V_r| > 4$ **do**
  let $\hat{E}_r \leftarrow \emptyset$
  if $c^1(E_r) \leq \frac{b_1(|V_r|-1)}{6}$ **then**
    pick an edge $e \in E_r$ with probability $p(e) = \frac{c^2(e)}{c^2(E_r)}$ and add it to $\hat{E}_r$
  else
    for $i = 1$ to q **do**
      for each edge $e \in E_r \setminus \hat{E}_r$ **do**
        add e to $\hat{E}_r$ with probability $p'(e) = \frac{3c^1(e)}{b_1(|V_r|-1)}$
      end for
    end for
  end if
  if $\hat{E}_r \neq \emptyset$ **then**
    contract all the edges in $\hat{E}_r$ by merging their endpoints
    replace all resulting parallel edges $e_1, \ldots, e_p$ joining any pair of nodes $u, v \in V_r$ by a single edge e such that $c^h(e) = \sum_{i=1}^{p} c^h(e_i)$, $h = 1, 2$, and remove self-loops
  end if
  $r \leftarrow r + 1$
  let $G_r = (V_r, E_r)$ denote the resulting graph
end while
randomly partition the nodes in the final graph G' and return the cut C^* in G associated with this partition

2 Edge-budget constrained Global Minimum cuts

We discuss in Sections 2.1 and 2.2 our randomized algorithms for the single budget constraint and for multiple ones, respectively.

2.1 Single edge-budget constraint

The algorithm consists of two steps. The first reduces the graph by doing edge contractions until a minor graph G' with at most four nodes is obtained. In the second step, we randomly pick a cut in the resulting four-node graph.

Starting from $G_0 = (V_0, E_0) = G = (V, E)$, the first step of each iteration $r \geq 1$ consists of a possible reduction of graph $G_r = (V_r, E_r)$ to a graph $G_{r+1} = (V_{r+1}, E_{r+1})$ by contracting a sample edge set $\hat{E}_r \subseteq E_r$. The construction of $\hat{E}_r$ is performed as follows. First we set $\hat{E}_r = \emptyset$. Then two cases are considered: (i) If $c^1(E_r) \leq \frac{b_1(|V_r|-1)}{6}$, then we randomly pick an edge $e \in E_r$ with probability $p(e) = \frac{c^2(e)}{c^2(E_r)}$, and add it to $\hat{E}_r$. (ii) If this is not the case, then we add each edge $e \in E_r$ to $\hat{E}_r$ with probability $p'(e) = \frac{3c^1(e)}{b_1(|V_r|-1)}$. Note that the resulting sample edge set $\hat{E}_r$ may be empty. In order to boost the probability that $\hat{E}_r$ is non-empty, the process of random sampling is repeated q times, where q is a constant that will be specified later. If $\hat{E}_r \neq \emptyset$ at the end of the q trials, then we contract $\hat{E}_r$ and obtain a smaller graph $G_{r+1} = (V_{r+1}, E_{r+1})$. Otherwise, we set $G_{r+1} = G_r$.

An iteration r of the algorithm where condition $c^1(E_r) > \frac{b_1(|V_r|-1)}{6}$ holds and $\hat{E}_r = \emptyset$ is called *void*. Note that at most $|V| - 4$ non void iterations are performed but the total number of iterations may be large.

As a result of the edge contractions, parallel edges may join some pairs of vertices. Note that parallel edges are in the same cuts. Therefore, they can be replaced by a single edge with a cost equal to the sum of their costs. In contrast to Karger's algorithm [7], we need to consider only simple graphs at each step of Algorithm 1 in order to get the claimed running time (Lemma 10). This step is not essential for the analysis of Algorithm 1 but since it will be implemented recursively (Algorithm 2), $|E_r|$ must be bounded by $O(|V_r|^2)$ at each step r. All these details are summarized in Algorithm 1.

The following result gives a lower bound on the success probability that a particular optimal cut is returned by Algorithm 1.

▶ **Proposition 3.** *Any fixed optimal cut C^* is returned by Algorithm 1 with probability $\Omega\big(n^{-\frac{3}{1-\exp(-\frac{q}{2})}}\big)$.*

Our strategy to prove Proposition 3 is to handle separately the two cases in each iteration of the algorithm depending on whether $c^1(E_r) \leq \frac{b_1(|V_r|-1)}{6}$ or not. In the following two lemmas, we prove that the success probability of not contracting an edge in the optimal cut is at least $1 - \frac{3}{(|V_r|-1)(1-\exp(-\frac{q}{2}))}$ in each of these cases respectively.

Any edge in the current graph $G_r = (V_r, E_r)$ represents one or more edges in the original graph G. On the contrary, any edge in E is associated to at most one edge in E_r. Let E_r^{-1} denote the set of all the edges in E that are associated to the edges in E_r. For any set S of edges in E, let $E_r(S)$ denote, if any, the set of edges in E_r associated to the edges in S. An edge $e \in E$ has *survived* in graph G_r if $e \in E_r^{-1}$.

▶ **Lemma 4.** *Fix a particular optimal solution C^* and suppose that all the edges in $\delta(C^*)$ have survived in graph $G_r(V_r, E_r)$. If $c^1(E_r) \leq \frac{b_1(|V_r|-1)}{6}$, then the success probability of contracting an edge not in $E_r(\delta(C^*))$ is at least $1 - \frac{3}{|V_r|-1}$.*

Proof. Let $V_r^{\leq} \subseteq V_r$ denote the set of feasible nodes $v \in V_r$, *i.e.*, $c^1(\delta(\{v\})) \leq b_1$ for all $v \in V_r^{\leq}$. Observe that after replacing any parallel edges by a single one, the cost of any cut in the current graph G_r is the same as in the original graph G. Therefore, $c^2(\delta(C^*)) \leq c^2(\delta(\{v\}))$ for all node $v \in V_r^{\leq}$. Moreover, we have $\sum_{v \in V_r} c^1(\delta(\{v\})) = 2c^1(E_r) \leq \frac{b_1(|V_r|-1)}{3}$, and $\sum_{v \in V_r} c^1(\delta(\{v\})) \geq \sum_{v \in V_r \setminus V_r^{\leq}} c^1(\delta(\{v\})) > b_1|V_r \setminus V_r^{\leq}|$. Thus, $|V_r \setminus V_r^{\leq}| < \frac{b_1(|V_r|-1)}{3b_1} = \frac{|V_r|-1}{3}$, and hence,

$$|V_r^{\leq}| \geq \frac{2}{3}(|V_r| - 1). \tag{1}$$

Since all the edges in $\delta(C^*)$ have survived in G_r, we have $c^2(E_r(\delta(C^*)) = c^2(\delta(C^*))$. Therefore, the error probability of randomly picking an edge $e \in E_r(\delta(C^*))$ is

$$\begin{aligned} Pr(e \in E_r(\delta(C^*))) &= \frac{c^2(E_r(\delta(C^*))}{c^2(E_r)} = \frac{c^2(\delta(C^*))}{c^2(E_r)} \\ &\leq \frac{\sum_{v \in V_r^{\leq}} c^2(\delta(\{v\}))}{|V_r^{\leq}|c^2(E_r)} \leq \frac{\sum_{v \in V_r} c^2(\delta(\{v\}))}{|V_r^{\leq}|c^2(E_r)} \\ &= \frac{2}{|V_r^{\leq}|} \leq \frac{3}{|V_r| - 1} \text{ (by (1)).} \end{aligned}$$

◀

▶ **Lemma 5.** *Fix a particular optimal solution C^* and suppose that all the edges in $\delta(C^*)$ have survived in graph $G_r(V_r, E_r)$. If $c^1(E_r) > \frac{b_1(|V_r|-1)}{6}$, then $Pr(E_r(\delta(C^*)) \cap \hat{E}_r \neq \emptyset) \leq \frac{3}{|V_r|-1}$.*

Proof. Let $\hat{E}_r^i$ denote the set of edges in E_r added to $\hat{E}_r$ in trial $i = 1, \ldots, q$. Let μ denote the expected cardinality of $E_r(\delta(C^*)) \cap \hat{E}_r$. We have

$$\begin{aligned}\mu &= \sum_{i=1}^{q} \sum_{e \in E_r(\delta(C^*)) \cap \hat{E}_r^i} (1 - p'(e))^{i-1} p'(e) \leq \sum_{e \in E_r(\delta(C^*))} p'(e) \\ &= \sum_{e \in \delta(C^*)} \frac{3c^1(e)}{b_1(|V_r| - 1)} \text{ (all the edges in } \delta(C^*) \text{ have survived)} \\ &= \frac{3c^1(\delta(C^*))}{b_1(|V_r| - 1)} \leq \frac{3}{|V_r| - 1}.\end{aligned}$$

The last inequality comes from that C^* is a feasible cut, and thus $c^1(\delta(C^*)) \leq b_1$. Consequently,

$$Pr\big(E_r(\delta(C^*)) \cap \hat{E}_r \neq \emptyset\big) = Pr\big(|E_r(\delta(C^*)) \cap \hat{E}_r| \geq 1\big) \leq Pr\big(|E_r(\delta(C^*)) \cap \hat{E}_r| \geq \frac{|V_r| - 1}{3}\mu\big).$$

By Markov's inequality, $Pr\big(|E_r(\delta(C^*)) \cap \hat{E}_r| \geq \frac{|V_r|-1}{3}\mu\big) \leq \frac{3}{|V_r|-1}$ and thus

$$Pr(E_r(\delta(C^*)) \cap \hat{E}_r \neq \emptyset) \leq \frac{3}{|V_r| - 1}. \tag{2}$$

◀

▶ **Lemma 6.** *In graph $G_r = (V_r, E_r)$, if $c^1(E_r) > \frac{b_1(|V_r|-1)}{6}$, then $Pr(\hat{E}_r \neq \emptyset) > 1 - \exp(-\frac{q}{2})$.*

Proof. If $c^1(E_r) > \frac{b_1(|V_r|-1)}{6}$, then Algorithm 1 constructs $\hat{E}_r$ by randomly sampling all edges. Let F_r^i denote the event that the sample set $\hat{E}_r^i$ obtained during trial i is non-empty and $\bar{F}_r^i$ be the complementary event, $i = 1, \ldots, q$. We have

$$\begin{aligned}Pr(\hat{E}_r \neq \emptyset) &= Pr\big(\cup_{i=1}^{q} F_r^i\big) = 1 - Pr\big(\cap_{i=1}^{q} \bar{F}_r^i\big) \\ &= 1 - Pr(\bar{F}_r^q | \cap_{i=1}^{q-1} \bar{F}_r^i) Pr(\bar{F}_r^{q-1} | \cap_{i=1}^{q-2} \bar{F}_r^i) \cdots Pr(\bar{F}_r^2 | \bar{F}_r^1) Pr(\bar{F}_r^1) \\ &= 1 - \big(\Pi_{e \in E_r}(1 - p'(e))\big)^q = 1 - \big(\Pi_{e \in E_r}(1 - \frac{3c^1(e)}{b_1(|V_r| - 1)})\big)^q \\ &> 1 - \big(\Pi_{e \in E_r} \exp\big(-\frac{3c^1(e)}{b_1(|V_r| - 1)}\big)\big)^q = 1 - \big(\exp\big(-\sum_{e \in E_r} \frac{3c^1(e)}{b_1(|V_r| - 1)}\big)\big)^q \\ &= 1 - \exp\big(-\frac{3qc^1(E_r)}{b_1(|V_r| - 1)}\big) > 1 - \exp\big(-\frac{q}{2}\big).\end{aligned}$$

The last inequality comes from the fact that $c^1(E_r) > \frac{b_1(|V_r|-1)}{6}$. ◀

Proof of Proposition 3: If the condition of Lemma 4 holds, then the success probability at iteration r is $Pr(E_r(\delta(C^*)) \cap \hat{E}_r = \emptyset) \geq 1 - \frac{3}{|V_r|-1}$. Otherwise, we need to consider two cases depending on whether iteration r is void or not. In the former case, the success probability is $Pr(E_r(\delta(C^*)) \cap \hat{E}_r = \emptyset | \hat{E}_r = \emptyset) = 1$. Now if iteration r is non void, then by Lemma 5 we have $Pr(E_r(\delta(C^*)) \cap \hat{E}_r \neq \emptyset) \leq \frac{3}{|V_r|-1}$. In this case, we have

$$Pr(E_r(\delta(C^*)) \cap \hat{E}_r \neq \emptyset | \hat{E}_r \neq \emptyset) = \frac{Pr(E_r(\delta(C^*)) \cap \hat{E}_r \neq \emptyset)}{Pr(\hat{E}_r \neq \emptyset)} < \frac{3}{(|V_r| - 1)(1 - \exp(-\frac{q}{2}))},$$

where the last equality follows from Lemma 6. Therefore, the success probability satisfies $Pr(E_r(\delta(C^*)) \cap \hat{E}_r = \emptyset | \hat{E}_r \neq \emptyset) > 1 - \frac{3}{(|V_r|-1)(1-\exp(-\frac{q}{2}))}$.

Algorithm 2 Recursive random edge contraction for a single edge-budget constraint.

Input: a graph $G = (V, E)$ with two nonnegative edge costs c^1, c^2, a bound b_1, and $\alpha = \frac{3}{1-\exp(-\frac{q}{2})}$ for $q = \Omega(\log(\log^2 n))$ (this implies $\alpha = O(1)$)

Output: a cut $\emptyset \neq C^* \subset V$ minimizing cost c^2 subject to edge-budget constraint $c^1(\delta(C^*)) \leq b_1$

1: **if** $|V| \leq 6$ **then**
2: randomly partition the nodes in G and return the cut C^* defined by this partition
3: **else**
4: $t \leftarrow \lceil \frac{|V|}{\sqrt[\alpha]{2}} + 1 \rceil$
5: repeat twice
6: apply the while loop in Line 2 of Algorithm 1 and contract at each iteration r all the edges in $\hat{E}_r$ until obtaining a graph $G' = (V', E')$ with at most t nodes
7: recursively solve the problem on graph G'
8: return the best of the two cuts (obtained from the two different runs)
9: **end if**

By taking the product of all the success probabilities over all the iterations, the probability that all the edges in $\delta(C^*)$ have survived in the final graph G' is at least

$$(1-\frac{3/(1-\exp(-\frac{q}{2}))}{|V|-1})(1-\frac{3/(1-\exp(-\frac{q}{2}))}{|V|-2})\cdots(1-\frac{3/(1-\exp(-\frac{q}{2}))}{4}) = \Omega(|V|^{-\frac{3}{1-\exp(-\frac{q}{2})}}).$$

The probability of picking uniformly a cut in the final graph, formed by at most four nodes, is 2^{-4}. Therefore, multiplying both probabilities gives the desired result.

Using the same probabilistic argument to bound the number of minimum cuts as in Karger [7] and setting $q = O(\log(\log^2 n))$ in Proposition 3, we get the following result.

▶ **Corollary 7.** *The number of optimal solutions of the single edge-budget constrained global minimum cut problem is bounded by $O(n^3)$.*

The number of iterations required to have a nonempty sample set is a geometric random variable, which by Lemma 6, has an expected value bounded by $\frac{1}{1-\exp(-\frac{q}{2})}$. Observe that the $O(m) = O(n^2)$ running time of the random sampling is bottleneck in Algorithm 1. Therefore, the expected running time of the algorithm is $O(q \cdot n^3)$.

In order to amplify the success probability given by Proposition 3, one needs to perform $O(n^{\frac{3}{1-\exp(-\frac{q}{2})}} \log n)$ runs of Algorithm 1, which is excessive. Hence, we embed it in the recursive framework of Karger and Stein's [8] algorithm.

Our recursive algorithm can be represented using a binary tree where the root corresponds to graph G. And for every node of the tree, associated with some graph $H = (W, F)$, the algorithm constructs two graphs $H_1 = (W_1, F_1)$ and $H_2 = (W_2, F_2)$ obtained by performing two sequences of contractions as in Algorithm 1. However, in contrast to Algorithm 1, these contractions are stopped when the number of nodes in W is reduced by a factor $\sqrt[\alpha]{2}$, where $\alpha = \frac{3}{1-\exp(-\frac{q}{2})}$ and $q = \Omega(\log(\log^2 n))$. It is known that the depth of such tree is bounded by $\lfloor \log_{\sqrt[\alpha]{2}} n \rfloor$ and the number of leaves is at most

$$O(2^{\lfloor \log_{\sqrt[\alpha]{2}} n \rfloor}) \leq O(2^{\log_{\sqrt[\alpha]{2}} n}) = O(n^{\log_{\sqrt[\alpha]{2}} 2}) = O(n^{\alpha}) = O(n^{\frac{3}{1-\exp(-\frac{q}{2})}}) = O(n^3).$$

See Cormen et al. [3] for more details. This procedure is summarized in Algorithm 2.

The following result (restatement of Theorem 1) gives bounds on the probability of success and the running time of Algorithm 2.

▶ **Theorem 8.** *Algorithm 2 returns any particular optimal solution in* $O(n^3 \log^4 n \log\log n)$ *time with probability* $1 - \frac{1}{\Omega(n)}$.

The proof of Theorem 8 will be a consequence of the following lemmas (the proofs are omitted due to space limitations). The first one shows that Algorithm 2 has the same success probability as the recursive algorithm of Karger and Stein [8].

▶ **Lemma 9.** *A fixed optimal solution* C^* *is returned by Algorithm 2 with probability* $\Omega\big(\frac{1}{\log n}\big)$.

▶ **Lemma 10.** *For* $q = O(\log(\log^2 n))$, *the expected running time is* $O(n^3 \log n \log\log n)$.

Using the observation that the running time of Algorithm 2 can be analyzed as a sum of several sums of geometric random variables, we provide an upper bound that holds with high probability.

▶ **Lemma 11.** *The probability that the running time of Algorithm 2 exceeds* $O(n^3 \log^2 n \log\log n)$ *is bounded by* $O(1/n)$.

By Lemmas 9 and 10, a particular optimal solution C^* is returned with high probability by performing $O(\log^2 n)$ calls to Algorithm 2, with each call to this algorithm taking expected $O(n^3 \log n \log\log n)$ time. By using the same argument as in Lemma 11, the running times of all these calls is $O(n^3 \log^4 n \log\log n)$ with high probability. This shows Theorem 8.

2.2 Multiple edge-budget constraints

We consider in this section the more general case where we have a constant number k of edge-budget constraints. Note that if k is variable, the problem is strongly NP-hard [1]. In the case of a single edge-budget constraint, Lemmas 4 and 5 show that the edges of an optimal cut form a small fraction of all the edges. Algorithm 1 exploits this crucial property in order to return an optimal cut with high probability. If the condition of Lemma 4 holds, then the number of feasible cuts formed by a singleton node is large. With more than two budget constraints, a cut satisfying the i^{th} budget constraint may violate the j^{th} one. Therefore, even though the number of cuts formed by a singleton node satisfying the i^{th} budget constraint may be large, few of them may satisfy all the budget constraints. Therefore, we need a different idea to tackle the difficulties raised by multiple constraints.

The basic idea of the following algorithm is to repeat contracting randomly chosen edges until obtaining a graph formed by $2k$ nodes. At this point, the algorithm returns a cut uniformly chosen at random in this graph. The main difference with Algorithm 1 lies in the way how the random selection is done.

In graph $G_r = (V_r, E_r)$ obtained at iteration r of the algorithm, a node $v \in V_r$ is called *feasible* if the cut $\delta(\{v\})$ satisfies all the edge-budget constraints. Otherwise, it is called *infeasible*. Let V_r^i for $i = 1, \ldots, k-1$ denote a subset of infeasible nodes in V_r violating the edge-budget constraint associated to cost c^i and V_r^k denote the subset of feasible nodes in V_r. We partition the nodes in V_r by assigning all the feasible nodes to V_r^k and assigning arbitrary any infeasible node v to one of the subsets V_r^i such that $c^i(\delta(\{v\})) > b_i$. Let E_r^i denote the subset of edges in E_r incident to at least a node in V_r^i for $i = 1 \ldots, k$. We choose randomly a set V_r^i with probability $p_i = \frac{|V_r^i|}{|V_r|}$ and then pick an edge $e \in E_r^i$ with probability $\frac{c^i(e)}{c^i(E_r^i)}$ and contract it. This procedure is summarized in Algorithm 3.

The following result gives a lower bound on the success probability of Algorithm 3 following arguments similar to Proposition 3 (the proof is omitted due to space limitation).

▶ **Lemma 12.** *Algorithm 3 outputs any fixed optimal cut* C^* *with probability* $\Omega(n^{-2k})$.

Algorithm 3 Random edge contraction for the edge-budget constrained minimum cut problem.

Input: a graph $G = (V, E)$ with k nonnegative edges cost $c^1, \ldots, c^k$ and $k-1$ nonnegative bounds $b_1, \ldots, b_{k-1}$

Output: a cut $\emptyset \neq C^* \subset V$ minimizing edges cost c^k subject to the constraints $c^i(C^*) \leq b_i$, for $i = 1, \ldots, k-1$

let $E_1 \leftarrow E$, $V_1 \leftarrow V$, $G_1 \leftarrow G$, $r \leftarrow 1$
while $|V_r| > 2k$ **do**
  flip a biased coin and choose set E_r^i with probability $p_i = \frac{|V_r^i|}{|V_r|}$
  pick randomly an edge $e \in E_r^i$ with probability $p(e) = \frac{c^i(e)}{c^i(E_r^i)}$
  contract e by merging its vertices and removing self-loops
  $r \leftarrow r + 1$
  let $G_r = (V_r, E_r)$ denote the resulting graph
end while
randomly partition the nodes in the final graph and return the cut C^* in G associated to this partition

Note that the lower bound given in Lemma 12 is the same as the one given in [8, Theorem 8.5] for the success probability of computing a specific k-approximate cut, *i.e.* a cut within a multiplicative factor k of the minimum. Therefore, by embedding Algorithm 3 in the recursive algorithm of Karger and Stein, one can show the following result.

▶ **Theorem 13.** *Algorithm 3 returns all optimal solutions for the edge-budget constrained min cut problem with $k-1$ budgets in $O(n^{2k} \log^2 n)$ with high probability.*

3 Node-Constrained Cut Problems

3.1 Node Budget-constrained Global Minimum Cut Problem

We discuss in this section a randomized algorithm for the minimum cut problem with node-budget constraints based on an extension of Karger's randomized contraction algorithm [7]. The algorithm exploits an observation given by Goemans and Soto [5] for solving the problem of minimizing a SSF f over a family of sets $\mathcal{I}$ that are closed under inclusion over a ground set V. A typical example of such a family is the knapsack family: Given a weight function $w : V \to \mathbb{R}^+$, consider the family $\mathcal{I} = \{A \subseteq V : \sum_{v \in A} w(v) \leq 1\}$. Let us first briefly review Goemans and Soto's algorithm which is based on an extension of Queyranne's algorithm [13].

Queyranne gave a combinatorial algorithm for minimizing a SSF f by extending the deterministic minimum cut algorithm of Nagamochi and Ibaraki [10]. The basic idea of Queyranne's algorithm is to construct an ordering $(v_1, \ldots, v_n)$ of the elements of the ground set V such that $f(v_n) \leq f(X)$ for all $X \subset V$ that separates v_n and v_{n-1}. Note that the element v_1 may be chosen arbitrary in this algorithm. The ordered pair (v_{n-1}, v_n) is called a *pendant pair*. The algorithm stores $\{v_n\}$ as a candidate solution and merges v_n and v_{n-1}. The process continues until only two elements are left. The best among all the stored candidates is an optimal solution.

In order to handle the knapsack constraint, Goemans and Soto [5] construct first a new element v_1 obtained by merging all the infeasible elements of V (not in $\mathcal{I}$) and compute an ordering $(v_1, \ldots, v_r)$. The authors observed that as in Queyranne's algorithm [13], (v_{r-1}, v_r)

Algorithm 4 Random edge contraction for the node-budget constrained min cut problem.

Input: a graph $G = (V, E)$ with nonnegative edges cost c, nonnegative node weights w^i for $i = 1, \ldots, k-1$, and node budgets b^i for $i = 1, \ldots, k-1$

Output: a feasible cut $\emptyset \neq C^* \subset V$ with minimum cost

1: let $E_1 \leftarrow E$, $V_1 \leftarrow V$, $r \leftarrow 1$, $V^> \leftarrow \{v \in V | w^i(v) > b^i$ for some $i \in \{1, \ldots, k-1\}\}$, $G_1 \leftarrow G \odot V^>$, i.e. G with all nodes of $V^>$ merged into a single infeasible supernode.
2: **while** $|V_r| > 3$ **do**
3: choose an arbitrary edge $e \in E_r$ with probability $\frac{c(e)}{c(E_r)}$
4: contract e by merging its endpoints and removing self-loops
5: **if** there exists two supernodes v and v' in V_r that are infeasible **then**
6: merge v and v'
7: **end if**
8: $r \leftarrow r + 1$
9: let $G_r = (V_r, E_r)$ denote the resulting graph
10: **end while**
11: return a feasible cut C^* in the final graph G' with minimum cost

is still a *pendant pair*. Our approach uses the same idea but our starting point is the random contraction algorithm of Karger.

Denote a cut X or a supernode representing a cut *infeasible* if its shore exceeds any of the node budget constraints, i.e. $w^i(X) > b^i$ for some $i \in \{1, \ldots, k-1\}$. Algorithm 4 maintains at most one infeasible supernode (denoting the contraction of many vertices) at any time and repeatedly tries to contract a randomly chosen edge. After a random contraction if a new infeasible supernode is formed, it is merged with the previously existing infeasible supernode deterministically. This process continues until the final graph G_r formed by only three supernodes. At this point, the algorithm selects a feasible cut C^* in G_r with minimum cost and outputs it as a candidate optimal solution. The full algorithm is described in Algorithm 4.

If V_r contains at least two infeasible nodes then any feasible cut does not separate them. In this case, these supernodes are merged safely in Step 6. Otherwise, V_r contains at most one infeasible supernode and in this case, Algorithm 4 randomly contracts, in Step 4, an edge in E_r. The following results show that the algorithm always find a feasible cut in the final graph G' and returns any fixed optimal cut with high probability (the proofs are omitted due to space limitation).

▶ **Lemma 14.** *The final graph G' always contain a feasible cut.*

▶ **Lemma 15.** *Algorithm 4 outputs any fixed optimal cut C^* with probability $\Omega(n^{-2})$.*

Using the same probabilistic argument to bound the number of minimum cuts as in Karger [7], Lemma 15 implies the following result.

▶ **Corollary 16.** *The number of optimal solutions of the node-budget constrained global minimum cut problem is bounded by $\binom{n}{2}$.*

Note that Algorithm 4 has the same error probability and running time as the original contraction algorithm [8, Theorem 2.2]. Therefore, we can embed it in the sophisticated recursive algorithm [8, Section 4] in order to produce an optimal cut with the same success probability and the same running time as for the global minimum cut problem (without the budget constraints). Furthermore, similarly to [8, Theorem 4.4], by executing the recursive

algorithm $O(\log^2 n)$ times, all the optimal solutions can be computed with high probability. The following result (restatement of Theorem 2) summarizes the resulting running times.

▶ **Theorem 17.** *An optimal cut of the node-budget constrained global minimum cut problem on an n-node graph can be computed in $O(n^2 \log n)$ time with probability $\Omega(1/\log n)$. Furthermore, all the optimal solutions can be computed with high probability in $O(n^2 \log^3 n)$ time.*

3.2 Node Budget-constrained sink-excluding Global Minimum Cut Problem

It is not hard to adapt Algorithm 4 for the node-budget constrained global minimum cut problem excluding a given sink $t \in V$, where we have a set of $k-1$ node-weight budget constraints on the shore of the cut excluding t. We obtain the following result (the full algorithm description is given in the full paper).

▶ **Theorem 18.** *An optimal cut of the node-budget constrained global minimum cut problem excluding a given sink in an n-node graph can be computed in $O(n^2 \log n)$ time with probability $\Omega(1/\log n)$. Furthermore, all the optimal solutions can be computed with high probability in $O(n^2 \log^3 n)$ time.*

3.3 Node-cardinality constrained Source-including Min-cuts

In contrast to the sink-excluding case, we show that even the node-cardinality constrained minimum cut problem containing a given source is strongly NP-hard using a reduction from graph bisection. Note that Hayrapetyan et al. [6] study the version that bounds the edge costs of the cut and minimizes the node-cardinality of the cut, and show NP-hardness of that version via a reduction from max-clique. We provide a direct hardness proof for our version (omitted due to space limitation) by reducing from graph bisection.

▶ **Theorem 19.** *The node-cardinality constrained minimum cut containing a given source is strongly NP-hard.*

On the other hand, for the exact version of the problem where the side containing s must have exactly k nodes, an $O(\log n)$-approximation was given by Räcke [14] using his approach for the graph bisection problem.

4 Conclusion

Our results show that beyond the running time improvement, Karger's randomized contraction algorithm is sufficiently flexible to tackle efficiently budget constraints. An important open question is whether the exact algorithms of Nagamochi and Ibaraki [10] and Stoer and Wagner [15] can be extended in order to handle these budget constraints, since they are based on similar observations but have the potential to lead to better deterministic algorithms for the problems we study.

References

1 Amitai Armon and Uri Zwick. Multicriteria global minimum cuts. *Algorithmica*, 46(1):15–26, 2006.

2 Maurizio Bruglieri, Francesco Maffioli, and Matthias Ehrgott. Cardinality constrained minimum cut problems: complexity and algorithms. *Discrete Applied Mathematics*, 137(3):311–341, 2004.

3 Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

4 Matthias Ehrgott. *Multicriteria optimization*. Springer Science & Business Media, 2006.

5 Michel X. Goemans and José A. Soto. Algorithms for symmetric submodular function minimization under hereditary constraints and generalizations. *SIAM Journal on Discrete Mathematics*, 27(2):1123–1145, 2013.

6 Ara Hayrapetyan, David Kempe, Martin Pál, and Zoya Svitkina. Unbalanced graph cuts. In *Algorithms – ESA 2005: 13th Annual European Symposium, Proceedings*, pages 191–202, 2005.

7 D. R. Karger. Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'93, pages 21–30, 1993.

8 D. R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, 1996.

9 Angsheng Li and Peng Zhang. Unbalanced graph partitioning. In *Algorithms and Computation: 21st International Symposium, ISAAC 2010, Proceedings, Part I*, pages 218–229, 2010.

10 Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992. doi:10.1137/0405004.

11 Hiroshi Nagamochi, Kazuhiro Nishimura, and Toshihide Ibaraki. Computing all small cuts in an undirected network. *SIAM Journal on Discrete Mathematics*, 10(3):469–481.

12 M. Padberg and G. Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47(1):19–36, 1990.

13 Maurice Queyranne. Minimizing symmetric submodular functions. *Mathematical Programming*, 82(1):3–12, 1998.

14 Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 255–264. ACM, 2008.

15 Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997.

16 Rico Zenklusen. Connectivity interdiction. *Operations Research Letters*, 42(6):450–454, 2014.

17 Peng Zhang. A new approximation algorithm for the unbalanced min s–t cut problem. *Theoretical Computer Science*, 609:658–665, 2016.

Tight Bounds for Online Coloring of Basic Graph Classes*†

Susanne Albers[1] and Sebastian Schraink[2]

1 **Technical University of Munich, Garching, Germany**
albers@in.tum.de
2 **Technical University of Munich, Garching, Germany**
schraink@in.tum.de

Abstract

We resolve a number of long-standing open problems in online graph coloring. More specifically, we develop tight lower bounds on the performance of online algorithms for fundamental graph classes. An important contribution is that our bounds also hold for randomized online algorithms, for which hardly any results were known. Technically, we construct lower bounds for chordal graphs. The constructions then allow us to derive results on the performance of randomized online algorithms for the following further graph classes: trees, planar, bipartite, inductive, bounded-treewidth and disk graphs. It shows that the best competitive ratio of both deterministic and randomized online algorithms is $\Theta(\log n)$, where n is the number of vertices of a graph. Furthermore, we prove that this guarantee cannot be improved if an online algorithm has a lookahead of size $O(n/\log n)$ or access to a reordering buffer of size $n^{1-\epsilon}$, for any $0 < \epsilon \leq 1$. A consequence of our results is that, for all of the above mentioned graph classes except bipartite graphs, the natural *First Fit* coloring algorithm achieves an optimal performance, up to constant factors, among deterministic and randomized online algorithms.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases graph coloring, online algorithms, lower bounds, randomization

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.7

1 Introduction

Online graph coloring is a classical problem in graph theory and online computation. It has applications in job scheduling, dynamic storage allocation and resource management in wireless networks [19, 23, 24]. A problem instance is defined by an undirected graph $G = (V, E)$, consisting of a vertex set V and an edge set E. Let $|V| = n$. The vertices arrive one by one in a sequence $\sigma = v_1, \ldots, v_n$ that may be determined by an adversary. Whenever a new vertex v_t arrives, $1 \leq t \leq n$, its edges to previous vertices v_s with $s < t$ are revealed. An online algorithm $\mathcal{A}$ has to immediately assign a feasible color to v_t, i.e. a color that is different from those assigned to the neighbors of v_t presented so far. The goal is to minimize the total number of colors used.

For a graph G, let $\mathcal{A}(G)$ be the number of colors used by $\mathcal{A}$. Let $\chi(G)$ be the chromatic number of G, which is the minimum number of colors needed to color G offline. An online algorithm $\mathcal{A}$ is *c-competitive* if $\mathcal{A}(G) \leq c \cdot \chi(G)$ holds for every graph G [25]. If $\mathcal{A}$ is a

* A full version of this paper is available at https://arxiv.org/abs/1702.07172.
† Work supported by the European Research Council, Grant Agreement No. 691672, project APEG.

randomized algorithm, then $E[\mathcal{A}(G)]$ is the expected number of colors used by $\mathcal{A}$. The algorithm is c-competitive against oblivious adversaries if $E[\mathcal{A}(G)] \leq c \cdot \chi(G)$ holds for every G [5]. An oblivious adversary, when determining σ, does not know the outcome of the random choices made by $\mathcal{A}$. We always evaluate randomized online algorithms against this type of adversary. When considering specific graph classes, for a deterministic or randomized algorithm, the competitive factor of c must hold for every graph from the given class.

The framework defined above is the standard online one. It is also interesting to explore settings where an algorithm is given more power. An online algorithm $\mathcal{A}$ has *lookahead* l if, upon the arrival of vertex v_t, the algorithm also sees the next l vertices $v_{t+1}, \ldots, v_{t+l}$ along with their adjacencies to vertices in $\{v_1, \ldots, v_{t+l}\}$. Alternatively, an algorithm might have a *buffer of size* b in which vertices can be stored temporarily. The requirement is that at the end of step t the algorithm must have colored at least $t - b$ vertices. A buffer is more powerful than lookahead because it allows the algorithm to partially reorder the input sequence and delay coloring decisions. The value of a buffer has recently been explored for a variety of online problems, see e.g. [1, 11] and references therein.

Previous work: For general graphs, the competitive ratios are high compared to the trivial upper bound of n. Lovasz, Saks and Trotter [22] developed a deterministic online algorithm that achieves a competitive factor of $O(n/\log^* n)$. Vishwanathan [26] devised a randomized algorithm that attains a competitiveness of $O(n/\sqrt{\log n})$. This bound was improved to $O(n/\log n)$ by Halldorsson [16]. Halldorsson and Szegedy [17] proved that the competitive ratio of any deterministic online algorithm is $\Omega(n/\log^2 n)$. This lower bound also holds for randomized algorithms. Moreover, it holds if a randomized algorithm has a lookahead or a buffer of size $O(\log^2 n)$ [17].

There has also been considerable research interest in online coloring for various graph classes. An early and celebrated result proved by Bean [4] in 1976 is that, for trees, every deterministic online algorithm can be forced to use $\Omega(\log n)$ colors. The *First Fit* algorithm colors every tree with $O(\log n)$ colors [15]. The natural strategy *First Fit* assigns the lowest-numbered feasible color to each incoming vertex. Since trees have a chromatic number of 2, the best competitive ratio achievable by deterministic online algorithms is $\Theta(\log n)$. For bipartite graphs, there also exists a deterministic online algorithm that uses $O(\log n)$ colors [22], implying that the best competitiveness of deterministic strategies is again $\Theta(\log n)$. However, *First Fit* performs poorly, as there are bipartite graphs for which it requires $\Omega(n)$ colors. Kierstead and Trotter [20] proved that, for interval graphs, the best competitive ratio of deterministic online algorithms is equal to 3.

A paper directly related to our work is by Irani [18]. She examined d-inductive graphs, also referred to as d-degenerate graphs. They are defined as the graphs which admit a numbering of the vertices such that each vertex is adjacent to at most d higher-numbered vertices. Every planar graph is 5-inductive and every chordal graph G is $(\chi(G) - 1)$-inductive. Irani [18] proved that *First Fit* colors every d-inductive graph with $O(d \cdot \log n)$ colors. Furthermore, for every deterministic online algorithm $\mathcal{A}$, there exist graphs such that $\mathcal{A}$ uses $\Omega(d \cdot \log n)$ colors [18]. Since d-inductive graphs have a chromatic number of at most $d + 1$, the best competitive ratio achieved by deterministic online algorithms is $\Omega(\log n)$. For planar graphs a tight bound of $\Theta(\log n)$ holds because trees are planar. However, it was an open problem if a tight competitiveness of $\Theta(\log n)$ holds for general chordal graphs. In fact, Irani [18] raised the question if, for every deterministic online algorithm $\mathcal{A}$ and every d, there exists a chordal graph with chromatic number d such that $\mathcal{A}$ uses $\Omega(d \cdot \log n)$ colors. Finally, for d-inductive graphs, Irani [18] analyzed deterministic online algorithms with lookahead l and

showed that the best competitiveness is $\Theta(\min\{\log n, n/l\})$. A lower bound of $\Omega(\log\log n)$ on the competitive ratio of randomized online algorithms for d-inductive graphs was given by Leonardi and Vitaletti [21].

We address two further graph classes. Downey and McCartin [10] studied online coloring of bounded treewidth graphs. For an introduction to treewidth see [7]. For any graph of treewidth d, *First Fit* uses $O(d \cdot \log n)$ colors. This is a consequence of Irani's work [18] because a graph of treewidth d is d-inductive [10, 18]. Downey and McCartin [10] showed that, on graphs of treewidth d, *First Fit* can be forced to use $\Omega(\frac{d}{\log(d+1)} \log n)$ colors. Last but not least, a disk graph is the intersection graph of a set of disks in the Euclidean plane. Each vertex represents a disk; two vertices are adjacent if the two corresponding disks intersect. Online coloring of disk graphs has received quite some attention because it models frequency assignment problems in wireless communication networks, see [13] for a survey. The best competitiveness achieved by a deterministic online algorithm is $\Theta(\min\{\log n, \log \rho\})$, where ρ is the ratio of the largest to smallest disk radius [9, 12]. The result relies on the common assumption that an online algorithm does not use the disk representation, when making coloring decisions [9, 12, 13]. It has been repeatedly raised as an open problem if the bound of $\Theta(\min\{\log n, \log \rho\})$ can be improved using randomization [9, 12, 13].

Recent work on online graph coloring has studied scenarios where an online algorithm can query oracle information about future input [8, 6]. Moreover, online coloring of hypergraphs has been explored [2, 3].

Our Contribution: In this paper we settle the performance of online coloring algorithms for fundamental and widely studied graph classes. More precisely, we prove lower bounds on the performance of online algorithms. These bounds match the best upper bounds known in the literature. An important contribution is that our bounds also hold for randomized online algorithms, for which very few results were known.

First, in Sections 2 and 3 we investigate chordal graphs. They have been studied extensively, cf. textbook [27]. We remind the reader that a graph is chordal if every induced cycle with four or more vertices has a chord. For a chordal graph G, the chromatic number $\chi(G)$ is equal to the largest clique size $\omega(G)$. Interval graphs are a subfamily of chordal graphs. Chordal graphs in turn are perfect graphs, for which the offline coloring, maximum clique and independent set problems can be solved in polynomial time.

In Section 2 we examine deterministic online coloring algorithms. We prove that, for every deterministic algorithm $\mathcal{A}$ and every integer $d \geq 2$, there exists a family of chordal graphs G with $\chi(G) = d$ such that $\mathcal{A}$ uses $\Omega(d \cdot \log n)$ colors. This resolves the open problem raised by Irani [18]. In Section 3 we extend this result to randomized online algorithms. The statement is identical to the one for deterministic algorithms, except that a randomized online algorithm uses an expected number of $\Omega(d \cdot \log n)$ colors. Although the result for randomized algorithms is more general, we give proofs for both deterministic and randomized policies. Our lower bound construction for deterministic algorithms exhibits an adversarial strategy for generating worst-case graphs. Given this strategy, we show how to define a probability distribution on graphs so that Yao's principle [28] can be applied. *First Fit* colors every chordal graph G with $\chi(G) = d$ using $O(d \cdot \log n)$ colors. Hence, the optimal competitiveness of deterministic and randomized online algorithms is $\Theta(\log n)$.

In Section 4 we derive lower bounds for further graph classes, focusing on randomized online algorithms. For $d = 2$, our lower bound construction for chordal graphs generates trees. It follows that, for any randomized online algorithm $\mathcal{A}$, there exists a family of trees such that $\mathcal{A}$ needs an expected number of $\Omega(\log n)$ colors. This complements the fundamental and

early result by Bean [4] for deterministic algorithms. To the best of our knowledge, no lower bound on the performance of randomized online coloring algorithms for trees was previously known. Recall that trees have a chromatic number of 2. Vishwanathan [26] gave a lower bound of $\Omega(\log n)$ on the expected number of colors used by randomized online algorithms for graphs of chromatic number 2, i.e. bipartite graphs. However, the graphs in his construction have cycles. Thus, Vishwanathan's lower bound does not apply to trees. Obviously, trees are planar and bipartite. Hence, our result for trees directly implies that every randomized online algorithm can be forced to use $\Omega(\log n)$ colors in expectation for graphs of these two classes. The lower bounds are tight because known deterministic online algorithms color trees, planar and bipartite graphs with $O(\log n)$ colors [15, 18, 22].

Section 4 also addresses inductive and bounded-treewidth graphs. Since every chordal graph G is $(\chi(G)-1)$-inductive and has treewidth $\chi(G)-1$, we derive the following results. For every randomized online algorithm $\mathcal{A}$ and every $d \geq 1$, there exists a family of d-inductive graphs such that $\mathcal{A}$ uses $\Omega(d \cdot \log n)$ colors. The same statement holds for graphs of treewidth d. We further show that the statement also holds for strongly chordal graphs with chromatic number d. A chordal graph is strongly chordal if every cycle of even length consisting of at least six vertices has an odd chord, i.e. an edge connecting two vertices that have an odd distance from each other in the cycle [14]. *First Fit* colors any d-inductive graph and any graph of treewidth d using $O(d \cdot \log n)$ colors. We conclude that, for all the graph classes considered so far, $\Theta(\log n)$ is the best competitiveness of deterministic and randomized online algorithms. Finally, in Section 4 we study disk graphs. We prove that, for $d = 2$, every graph of the probability distribution defined in Section 3 translates to a disk graph. We then show that, for every randomized online algorithm $\mathcal{A}$ that does not use the disk representation, there exists a family of disk graphs forcing $\mathcal{A}$ to use an expected number of $\Omega(\min\{\log n, \log \rho\})$ colors, where ρ is again the ratio of the largest to smallest disk radius. Hence randomization does not improve the asymptotic performance of online coloring algorithms for disk graphs, cf. [9, 12, 13].

In Section 5 we explore the settings where an online algorithm has lookahead or is equipped with a reordering buffer. We show that a lookahead of size $O(n/\log n)$ does not improve the asymptotic performance of randomized online algorithms. We prove the result for chordal graphs and then derive analogous results for all the other graph classes. Irani [18] gave a similar result for deterministic algorithms, considering inductive graphs. As a final result of this paper we demonstrate that a reordering buffer of size $n^{1-\epsilon}$, for any $0 < \epsilon \leq 1$, does not yield an improvement in the asymptotic performance guarantees of deterministic online algorithms. Again, we develop the result for chordal graphs and derive corollaries for the other graph classes.

Our Proof Technique: We devise a technique for proving lower bounds that is relatively simple; we view this as a strength of our results. The main idea is to recursively construct trees of cliques, which in turn form forests. In a recursive step the construction combines forests by adding or not adding a new clique in a specific way. Our construction resembles the one by Bean [4] but differs in an important aspect that allows us to obtain lower bounds for randomized algorithms. The construction by Bean builds a tree T_k, $k \in \mathbb{N}$, by joining trees T_j, for $j < k$, so that any deterministic online algorithm must use a k-th new color for *some* vertex of T_k. This vertex then becomes the root of T_k. An oblivious adversary, playing against a randomized online algorithm, cannot identify with sufficiently high probability such vertices exhibiting a new color. Instead, our construction maintains the invariant that the root vertices of each forest use a large number of colors, given any deterministic online

algorithm. For randomized algorithms, a corresponding invariant holds with probability of at least $1/2$.

Convention: Unless otherwise stated, logarithms are base 2.

2 Deterministic online algorithms for chordal graphs

We establish a lower bound on the performance of any deterministic online coloring algorithm.

▶ **Theorem 1.** *Let $d \in \mathbb{N}$ with $d \geq 2$ be arbitrary. For every deterministic online algorithm $\mathcal{A}$ and every $n \in \mathbb{N}$ with $n \geq 2d^2$, there exists a n-vertex chordal graph G with chromatic number $\chi(G) = d$ such that $\mathcal{A}$ uses $\Omega(d \cdot \log n)$ colors to color G.*

The proof of Theorem 1 relies on Lemma 2, which we prove first.

▶ **Lemma 2.** *Let $d \in \mathbb{N}$ with $d \geq 2$ be arbitrary. For every deterministic online algorithm $\mathcal{A}$ and every $k \in \mathbb{N}$, there exists a chordal graph G_k having chromatic number $\chi(G_k) = d$ and consisting of $n_k \leq d2^k$ vertices such that $\mathcal{A}$ is forced to use at least $c_k \geq (d-1)k/4$ colors to color G_k.*

Proof. We describe how an adversary constructs a chordal graph G_k, $k \in \mathbb{N}$. Such a graph is built up recursively and consists of graphs G_j, where $j < k$. We assume that d is even. The construction of G_k can be adapted easily if d is odd; details will be given later. On a high level G_k is a forest, i.e. a collection of disjoint trees, each having a distinguished root node. In every tree T of G_k, each tree node represents a clique of size $d/2$ in G_k. If two tree nodes u_T and v_T are connected by a tree edge in T, then any two vertices $u \in u_T$ and $v \in v_T$ are connected by an edge in G_k. Hence u_T and v_T form a clique of size d in G_k. Since G_k is a forest, it consists of several connected components. One can add a final vertex and edges in order to connect the various trees; details will be given at the end of the proof.

We proceed with the concrete construction of G_k, for increasing values of $k \in \mathbb{N}$. As mentioned above, each tree T of G_k has a distinguished root node consisting of $d/2$ vertices in G_k. Let $r(T)$ be the set of these $d/2$ vertices. Moreover, let $r(G_k)$ be the union of these sets $r(T)$, taken over all T of G_k. We refer to the elements of $r(G_k)$ as the *root vertices of* G_k. They are important because the online algorithm $\mathcal{A}$ will be forced to use a large number of colors for $r(G_k)$. For any subset V' of the vertices of G_k, let $\mathcal{C}_{\mathcal{A}}(V')$ be the set of colors used by $\mathcal{A}$ to color V'.

The strategy of the adversary to generate a graph G_k is adaptive, i.e. the exact structure of the graph depends on the coloring decisions of $\mathcal{A}$. Nevertheless, during the bottom-up construction of G_k, for increasing $k \in \mathbb{N}$, the following invariants will be maintained.

(1) Algorithm $\mathcal{A}$ uses at least $\frac{d}{4} \cdot k$ colors for the root vertices of G_k, i.e. $|\mathcal{C}_{\mathcal{A}}(r(G_k))| \geq \frac{d}{4} \cdot k$.
(2) G_k is a union of connected components, each of which can be represented by a tree T. Each tree node is a clique of size $d/2$. Every tree T has a distinguished root node containing a set $r(T)$ of $d/2$ root vertices in G_k.
(3) G_k is chordal.
(4) The maximum clique size is $\omega(G_k) = d$.
(5) The number of vertices satisfies $n_k \leq \frac{d}{2} \cdot (2^{k+1} - 1)$.

Invariants (3) and (4) together imply that $\chi(G_k) = \omega(G_k) = d$ holds. In invariant (1) and the following technical exposition integer values are compared to expressions of the form $\frac{d}{4} \cdot k$, which might not be integer. We remark that the statements, comparisons and calculations hold without considering the rounded expressions.

Figure 1 The tree T representing G_1.

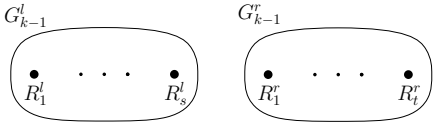


Figure 2 The general structure of G^l_{k-1} and G^r_{k-1} restricted to the root vertices.

Construction of the base graph G_1: G_1 is a clique of size d. The adversary may present the corresponding vertices in an arbitrary order. The set of root vertices $r(G_1)$ is an arbitrary subset R of size $d/2$ of the vertices of G_1. The remaining $d/2$ vertices form a second tree node. The resulting tree T is depicted in Figure 1. We can easily verify properties (1–5).

(1) Since $R = r(G_1)$ is a clique of size $d/2$, $\mathcal{A}$ uses $d/2$ colors for it, i.e. $|\mathcal{C}_{\mathcal{A}}(r(G_1))| \geq \frac{d}{4}$.

(2) G_1 consists of one connected component which represents a tree, as described above and shown in Figure 1.

(3) G_1 is a clique and thus chordal.

(4) The maximum clique size $\omega(G_1)$ is exactly d.

(5) There holds $n_1 = d \leq \frac{3}{2} \cdot d = \frac{d}{2} \cdot (2^{1+1} - 1)$.

Construction of the graph G_k, $k > 1$: Assume that the adversary can generate graphs G_j, for any $j < k$, satisfying invariants (1–5). The construction of G_k proceeds as follows. First the adversary recursively generates two independent graphs of type G_{k-1}, i.e. it twice executes the strategy for generating a graph G_{k-1}. Let G^l_{k-1} and G^r_{k-1} be these two graphs. They are created one after the other. We remark that G^l_{k-1} and G^r_{k-1} need not be identical because $\mathcal{A}$'s coloring decision in one graph can affect its decisions in the other one.

In the following we focus on the root vertices of G^l_{k-1} and G^r_{k-1}. In particular, we consider the colors used by $\mathcal{A}$. Invariant (1) implies that $\left|\mathcal{C}_{\mathcal{A}}(r(G^l_{k-1}))\right| \geq \frac{d}{4}(k-1)$ and $\left|\mathcal{C}_{\mathcal{A}}(r(G^r_{k-1}))\right| \geq \frac{d}{4}(k-1)$. We distinguish two cases depending on the total number of colors used, i.e. the cardinality of $\mathcal{C}_{\mathcal{A}}(r(G^l_{k-1}) \cup r(G^r_{k-1}))$. To this end we introduce some notation. Assume that G^l_{k-1} consists of s connected components, which we number in an arbitrary way. Each component/tree T^l_i has a distinguished root containing a set $r(T^l_i)$ of $d/2$ root vertices. We abbreviate $R^l_i = r(T^l_i)$, $1 \leq i \leq s$. Similarly, assume that G^r_{k-1} consists of t connected components. Set $r(T^r_j)$ is the set of root vertices in the component T^r_j. Let $R^l_j = r(T^r_j)$, $1 \leq j \leq t$. There holds $r(G^l_{k-1}) = \bigcup_{i=1}^{s} R^l_i$ and $r(G^r_{k-1}) = \bigcup_{j=1}^{t} R^r_j$. Figure 2 shows the general structure of G^l_{k-1} and G^r_{k-1} by focusing on the roots. The left-hand side of the figure depicts G^l_{k-1} as a union of connected components rooted at $R^l_1, \ldots, R^l_s$, respectively. The right-hand side shows G^r_{k-1} as a collection of components rooted at $R^r_1, \ldots, R^r_s$.

Case 1: Assume that $\left|\mathcal{C}_{\mathcal{A}}(r(G^l_{k-1}) \cup r(G^r_{k-1}))\right| \geq \frac{d}{4} \cdot k$. In this case the adversary defines G_k as the union of G^l_{k-1} and G^r_{k-1}. No further vertices or edges are added. It is easy to verify the five invariants because G^l_{k-1} and G^r_{k-1} satisfy them by inductive assumption.

(1) The condition of Case 1 ensures $|\mathcal{C}_{\mathcal{A}}(r(G_k))| = \left|\mathcal{C}_{\mathcal{A}}(r(G^l_{k-1}) \cup r(G^r_{k-1}))\right| \geq \frac{d}{4} \cdot k$.

(2) The invariant is satisfied since G_k is the union of G^l_k and G^r_k.

(3) G_k is chordal because G^l_k and G^r_k are, and no further vertices or edges have been added.

(4) There holds $\omega(G_k) = d$, as $\omega(G^l_{k-1}) = \omega(G^r_{k-1}) = d$.

(5) Let n^l_{k-1} and n^r_{k-1} be the number of vertices in G^l_{k-1} and G^r_{k-1}, respectively. There holds $n_k = n^l_{k-1} + n^r_{k-1} \leq 2 \cdot (\frac{d}{2} \cdot (2^k - 1)) = \frac{d}{2} \cdot (2^{k+1} - 2) \leq \frac{d}{2} \cdot (2^{k+1} - 1)$. The first inequality follows because (5) holds for n^l_{k-1} and n^r_{k-1}.

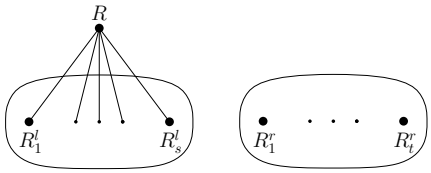


Figure 3 The graph G_k with the new addition of R.

Case 2: Next assume that $\left|\mathcal{C}_A(r(G_{k-1}^l) \cup r(G_{k-1}^r))\right| < \frac{d}{4} \cdot k$. In this case the adversary adds a set R of $d/2$ vertices that form a clique. Moreover, for every vertex of R there is an edge to every vertex in R_i^l, for $i = 1, \ldots, s$. In other words, every vertex of R has edges to all root vertices of $r(G_{k-1}^l)$. The vertices of R together with their adjacent edges may be presented by the adversary in an arbitrary order. The resulting structure is depicted in Figure 3. Set R and the connected components of G_{k-1}^l rooted at $R_1^l, \ldots, R_s^l$ form a single component rooted at R. There is a tree edge between R and every R_i^l, $1 \leq i \leq s$. The newly created component forms a tree rooted at R because the components of G_{k-1}^l represent trees rooted at $R_1^l, \ldots, R_s^l$. Graph G_k is the union of the new component and the components of G_{k-1}^r. The set of root vertices of G_k consists of R and the root vertices of G_{k-1}^r. Formally, $r(G_k) = R \cup R_1^r \cup \ldots, \cup R_t^r$. It remains to verify the five invariants.

(1) We analyze the number of colors that $\mathcal{A}$ uses for the root vertices in G_k. In a first step, among the colors $\mathcal{C}_\mathcal{A}(r(G_{k-1}^l)) \cup \mathcal{C}_\mathcal{A}(r(G_{k-1}^r))$ for the roots of G_{k-1}^l and G_{k-1}^r, we upper bound the number q of colors occurring in $\mathcal{C}_\mathcal{A}(r(G_{k-1}^r))$ only. By assumption $\left|\mathcal{C}_\mathcal{A}(r(G_{k-1}^l)) \cup \mathcal{C}_\mathcal{A}(r(G_{k-1}^r))\right| = \left|\mathcal{C}_\mathcal{A}(r(G_{k-1}^l) \cup r(G_{k-1}^r))\right| < \frac{d}{4} \cdot k$. There holds $\mathcal{C}_\mathcal{A}(r(G_{k-1}^l)) \geq \frac{d}{4}(k-1)$. We obtain $q = \left|\mathcal{C}_\mathcal{A}(r(G_{k-1}^r)) \setminus \mathcal{C}_\mathcal{A}(r(G_{k-1}^l))\right| = \left|\mathcal{C}_\mathcal{A}(r(G_{k-1}^r)) \cup \mathcal{C}_\mathcal{A}(r(G_{k-1}^l))\right| - \left|\mathcal{C}_\mathcal{A}(r(G_{k-1}^l))\right| < \frac{d}{4}$. Next consider the vertices in R. We upper bound the number of colors from $\mathcal{C}_\mathcal{A}(r(G_{k-1}^r))$ that $\mathcal{A}$ can use for R. Observe that $\mathcal{C}_\mathcal{A}(r(G_{k-1}^r))$ is the disjoint union of $\mathcal{C}_\mathcal{A}(r(G_{k-1}^l)) \cap \mathcal{C}_\mathcal{A}(r(G_{k-1}^r))$ and $\mathcal{C}_\mathcal{A}(r(G_{k-1}^r)) \setminus \mathcal{C}_\mathcal{A}(r(G_{k-1}^l))$. Every vertex of R is adjacent to every vertex in $r(G_{k-1}^l)$. Hence, $\mathcal{A}$ cannot apply a color occurring in $\mathcal{C}_\mathcal{A}(r(G_{k-1}^r)) \cap \mathcal{C}_\mathcal{A}(r(G_{k-1}^l))$ to a vertex in R. Only a color of $\mathcal{C}_\mathcal{A}(r(G_{k-1}^r)) \setminus \mathcal{C}_\mathcal{A}(r(G_{k-1}^l))$ is feasible, and the latter set has cardinality $q < d/4$. Since R is a clique of size $d/2$ algorithm $\mathcal{A}$ must use at least $d/2 - q > d/4$ colors not contained in $\mathcal{C}_\mathcal{A}(r(G_{k-1}^r))$ to color the vertices of R. As $r(G_k) = R \cup r(G_{k-1}^r)$, we conclude $|\mathcal{C}_\mathcal{A}(r(G_k))| = \left|\mathcal{C}_\mathcal{A}(R \cup r(G_{k-1}^r))\right| = \left|\mathcal{C}_\mathcal{A}(r(G_{k-1}^r))\right| + \left|\mathcal{C}_\mathcal{A}(R) \setminus \mathcal{C}_\mathcal{A}(r(G_{k-1}^r))\right| \geq \frac{d}{4}(k-1) + \frac{d}{4} = \frac{d}{4}k$.

(2) By construction G_k is a collection of connected components, forming trees rooted at R and $R_1^r, \ldots, R_t^r$, respectively.

(3) In G_k consider a simple cycle C with at least four vertices and assume that at least one vertex is in R. If three or more vertices of C are in R, then there is a chord because R is a clique. If C contains one or two vertices of R, then C can visit only one connected component of G_{k-1}^l. Suppose that it visits the one rooted at R_i^l. Cycle C must contain two vertices of R_i^l. Each of these two vertices has an edge to every vertex of R in C. Hence C has a chord. Since G_{k-1}^l and G_{k-1}^r, and thus the components rooted at $R_1^l, \ldots, R_s^l$ and $R_1^r, \ldots, R_t^r$, are chordal, so is G_k.

(4) Set R and each R_i^l, $1 \leq i \leq s$, form a clique of size d. The vertices of R are not connected to any vertices outside R_i^l, $1 \leq i \leq s$. Hence no other cliques are formed by the addition of R. Since $\omega(G_{k-1}^l) = \omega(G_{k-1}^r) = d$ it follows $\omega(G_k) = d$.

(5) Again, let n_{k-1}^l and n_{k-1}^r be the number of vertices in G_{k-1}^l and G_{k-1}^l. We have $n_k = n_{k-1}^l + n_{k-1}^r + \frac{d}{2} \leq 2 \cdot (\frac{d}{2} \cdot (2^k - 1)) + \frac{d}{2} = \frac{d}{2} \cdot (2^{k+1} - 2) + \frac{d}{2} = \frac{d}{2} \cdot (2^{k+1} - 1)$.

The construction and analysis of G_k is complete.

Graph G_k consists of several connected components if $k > 1$. The adversary can create a connected graph by adding a final vertex v_f that has an edge to exactly one root vertex in each of the components. The resulting graph remains chordal because there is no simple cycle containing v_f. By the addition of v_f the maximum clique size does not change. Including v_f the total number of vertices is upper bounded by $\frac{d}{2}(2^{k+1}-1)+1 \leq d2^k$ because $d \geq 2$. The lemma follows from invariants (1) and (3–5) because $\chi(G_k) = \omega(G_k) = d$.

We finally address the case that d is odd. In this case the adversary executes the graph construction described above for parameter $d-1$, which is even. In the end when G_k is generated for the desired k, the adversary adds a final vertex to each base graph G_1. This vertex has edges to every other vertex of the corresponding G_1. This increases the maximum clique size from $d-1$ to d. The new graph remains chordal. The number of colors used by algorithm $\mathcal{A}$ is at at least $\frac{d-1}{4}k$. We observe that the number of base graphs G_1 in G_k is 2^{k-1}. Hence, in the extended graph the total number of vertices is upper bounded by $\frac{d-1}{2}(2^{k+1}-1)+2^{k-1} \leq \frac{d}{2}(2^{k+1}-1)$. If $k > 1$, the adversary can add a final vertex to link the various components. Again the lemma follows. ◀

Proof of Theorem 1. Given d and n, let $k = \lfloor \log(n/d) \rfloor$. There holds $k \in \mathbb{N}$ because $n \geq 2d^2 > 2d$. For every deterministic online algorithm, by Lemma 2, there exists a chordal graph G_k with chromatic number $\chi(G_k) = d$ such that $\mathcal{A}$ uses at least $c_k \geq (d-1)k/4$ colors. Graph G_k has $n_k \leq d2^k$ vertices. By the choice of $k = \lfloor \log(n/d) \rfloor$, we have $n_k \leq n$. To G_k we add $n - n_k$ vertices, all of which have one edge to an arbitrary vertex of G_k. The resulting n-vertex graph remains chordal and $\chi(G) = d$. Since $d \geq 2$, there holds $c_k \geq dk/8$. We have $k \geq \log n - \log d - 1$. Inequality $n \geq 2d^2$ is equivalent to $d \leq \sqrt{n/2}$. Thus, $k \geq \log(n/2) - 1/2 \cdot \log(n/2) = 1/2 \cdot \log(n/2)$. As $n \geq 2d^2 \geq 4$, there holds $\log(n/2) \geq 1/2 \cdot \log n$. Hence, the number of colors used by $\mathcal{A}$ is at least $c_k \geq d \log n/32$. ◀

In Theorem 1 the lower bound on n can be reduced from $2d^2$ to $2d^{1+\epsilon}$, for any $0 < \epsilon < 1$. Then the number of colors used by $\mathcal{A}$ is $\Omega(\epsilon \cdot d \cdot \log n)$.

3 Randomized online algorithms for chordal graphs

We extend the result of Theorem 1 to randomized algorithms against oblivious adversaries.

▶ **Theorem 3.** *Let $d \in \mathbb{N}$ with $d \geq 2$ be arbitrary. For every randomized online algorithm $\mathcal{A}$ and every $n \in \mathbb{N}$ with $n \geq 12d^2$, there exists a n-vertex chordal graph G with chromatic number $\chi(G) = d$, presented by an oblivious adversary, such that the expected number of colors used by $\mathcal{A}$ to color G is $\Omega(d \cdot \log n)$.*

In order to prove Theorem 3 we resort to Yao's principle [28] and show the following Lemma 4.

▶ **Lemma 4.** *Let $d \in \mathbb{N}$ with $d \geq 2$ be arbitrary. For every $k \in \mathbb{N}$, there exists a probability distribution on a set $\mathcal{G}_k$ of chordal graphs with the following properties. For every $G_k \in \mathcal{G}_k$, $\chi(G_k) = d$ and the number of vertices is at most $d \cdot 12^k$. The expected number of colors used by any deterministic online algorithm to color a graph drawn according to the distribution is at least $(d-1)k/8$.*

Proof. For every $k \in \mathbb{N}$ we define a set $\mathcal{G}_k$ of chordal graphs G_k, each having a chromatic number of d. Moreover, we specify the order in which the vertices of any $G_k \in \mathcal{G}_k$ are presented to a deterministic online algorithm $\mathcal{A}$. The distribution on $\mathcal{G}_k$ is the uniform one, i.e. each $G_k \in \mathcal{G}_k$ is chosen with the same probability. We assume that d is even. The definition of $\mathcal{G}_k$ can be adapted easily if d is odd; details are given at the end of the proof.

The set $\mathcal{G}_k$ is built recursively based on $\mathcal{G}_{k-1}$. The construction of graphs $G_k \in \mathcal{G}_k$ is a generalization of the one presented in the proof of Lemma 2. A major difference is that any $G_k \in \mathcal{G}_k$ contains twelve graphs of $\mathcal{G}_{k-1}$, which are grouped into six pairs. For each pair a clique of size $d/2$ may or may not be added. As before, every $G_k \in \mathcal{G}_k$ is a union of connected components. Each such component can be represented by a tree with a distinguished root vertex. Every tree vertex is a set of $d/2$ vertices forming a clique in G_k. We reuse the notation of the proof of Lemma 2. Given $G_k \in \mathcal{G}_k$, for any component/tree T of G_k, $r(T)$ is the set of $d/2$ vertices in the root of T. Set $r(G_k)$ is the union of all $r(T)$, taken over all T of G_k. Finally $\mathcal{C}_{\mathcal{A}}(r(G_k))$ is the set of colors used by $\mathcal{A}$ for the vertices of $r(G_k)$.

During the recursive construction of $\mathcal{G}_k$, for increasing $k \in \mathbb{N}$, the following invariants are maintained. Compared to the proof of Lemma 2, (1) and (5) differ. Invariant (1) states that, for a randomly chosen G_k, every deterministic online algorithm needs, with probability greater than $1/2$, at least $dk/4$ colors for the root vertices $r(G_k)$. Invariant (5) gives an adjusted bound on the size of any G_k.

(1) If G_k is chosen uniformly at random from $\mathcal{G}_k$, then for any deterministic online algorithm $\mathcal{A}$, $\Pr[|\mathcal{C}_{\mathcal{A}}(r(G_k))| \geq dk/4] > 1/2$. This holds independently of other connected components $\mathcal{A}$ might have already colored.

(2) Every $G_k \in \mathcal{G}_k$ is a union of connected components, each of which can be represented by a tree T. Each tree node is a clique of size $d/2$. Every tree T has a distinguished root containing a set $r(T)$ of $d/2$ root vertices in G_k.

(3) Every $G_k \in \mathcal{G}_k$ is chordal.

(4) For every $G_k \in \mathcal{G}_k$, the maximum clique size is $\omega(G_k) = d$.

(5) For every $G_k \in \mathcal{G}_k$, the number n_k of vertices satisfies $n_k \leq d(12^k - 1)$.

Graph set $\mathcal{G}_1$: The set only contains G_1, the base graph used in the proof of Lemma 2, which is a clique of size d. The vertices of G_1 may be presented in any order to a deterministic online algorithm. Again, the set $r(G_1)$ of root vertices is an arbitrary subset of size $d/2$ of the vertices of G_1. The remaining $d/2$ vertices form a second tree node. Every deterministic online algorithm, with probability 1, needs $d/2$ colors for $r(G_1)$, which implies (1). Invariants (2–4) are obvious. As for (5), there holds $n_1 = d \leq d(12 - 1)$.

Graph set $\mathcal{G}_k$, $k > 1$: Assume that the set $\mathcal{G}_{k-1}$ satisfying (1–5) has been constructed. First, in order to build $\mathcal{G}_k$, all possible 12-tuples of graphs of $\mathcal{G}_{k-1}$ are formed. In assigning tuple entries, graphs of $\mathcal{G}_{k-1}$ are selected with replacement. Hence, a total of $|\mathcal{G}_{k-1}|^{12}$ tuples are built. For each tuple, 2^6 graphs are added to $\mathcal{G}_k$ in the following way. Let τ be any fixed tuple. Six graph pairs are formed. For $i = 1, \ldots, 6$, let $G_{k-1}^{i,l}$ and $G_{k-1}^{i,r}$ be the graphs in tuple entries $2i - 1$ and $2i$, respectively. To the i-th pair a clique R_i of size $d/2$ may or may not be added. The possible additions, over the six pairs, can be represented by a bit vector $\vec{b} = (b_1, \ldots, b_6)$. More specifically, given τ and any such bit vector $\vec{b}$, a graph G_k is constructed as follows. For $i = 1, \ldots, 6$, a subgraph G_k^i is generated. If $b_i = 0$, then G_k^i is the union of $G_{k-1}^{i,l}$ and $G_{k-1}^{i,r}$. The set $r(G_k^i)$ of root vertices is the union of $r(G_k^{i,l})$ and $r(G_k^{i,r})$. If $b_i = 1$, then a clique R_i of size $d/2$ is added to $G_{k-1}^{i,l}$ and $G_{k-1}^{i,r}$. Every vertex of R_i has an edge to every vertex of $r(G_{k-1}^{i,l})$. Subgraph G_k^i consists of the newly created component rooted at R_i and $r(G_{k-1}^{i,r})$, i.e. $r(G_{k-1}^i) = R_i \cup r(G_{k-1}^{i,r})$. Graph G_k is the union of the G_k^i and the set $r(G_k)$ is the union of the $r(G_k^i)$, $1 \leq i \leq 6$. When G_k is presented to $\mathcal{A}$, the subgraphs G_k^i are revealed one by one, $1 \leq i \leq 6$. For each G_k^i the graphs $G_{k-1}^{i,l}$ and $G_{k-1}^{i,r}$

are presented recursively. Finally, the vertices of R_i, if they exist, are shown. It remains to verify the invariants.

(1) Let G_k be a graph drawn uniformly at random from $\mathcal{G}_k$. Consider any subgraph G_k^i, $1 \leq i \leq 6$, containing $G_k^{i,l}$ and $G_k^{i,r}$. By the construction of $\mathcal{G}_k$, both $G_k^{i,l}$ and $G_k^{i,r}$ represent graphs drawn uniformly at random from $\mathcal{G}_{k-1}$. Let $\mathcal{A}$ be any deterministic online algorithm. Invariant (1) for $k-1$ implies $\Pr[|\mathcal{C}_{\mathcal{A}}(r(G_{k-1}^{i,l}))| \geq d(k-1)/4] > 1/2$ and $\Pr[|\mathcal{C}_{\mathcal{A}}(r(G_{k-1}^{i,r}))| \geq d(k-1)/4] > 1/2$. Moreover it implies $\Pr[|\mathcal{C}_{\mathcal{A}}(r(G_{k-1}^{i,l}))| \geq d(k-1)/4$ and $|\mathcal{C}_{\mathcal{A}}(r(G_{k-1}^{i,r}))| \geq d(k-1)/4] > 1/4$. Let $\mathcal{E}^i$ be the latter event that $|\mathcal{C}_{\mathcal{A}}(r(G_{k-1}^{i,l}))| \geq d(k-1)/4$ and $|\mathcal{C}_{\mathcal{A}}(r(G_{k-1}^{i,r}))| \geq d(k-1)/4$ hold.

Assume that $\mathcal{E}^i$ holds. There are two cases, which correspond to those analyzed in the proof of Lemma 2. If $|\mathcal{C}_{\mathcal{A}}(r(G_{k-1}^{i,l}) \cup r(G_{k-1}^{i,l}))| \geq dk/4$, then $|\mathcal{C}_{\mathcal{A}}(r(G_k^i))| \geq dk/4$ if R_i is not added to $G_k^{i,l}$ and $G_k^{i,r}$, which happens with probability $1/2$. On the other hand, if $|\mathcal{C}_{\mathcal{A}}(r(G_{k-1}^{i,l}) \cup r(G_{k-1}^{i,r}))| < dk/4$, then the addition of R_i ensures that $|\mathcal{C}_{\mathcal{A}}(r(G_k^i))| \geq dk/4$. Again, R_i is added with probability $1/2$. In either case, given $\mathcal{E}^i$, $\Pr[|\mathcal{C}_{\mathcal{A}}(r(G_k^i))| \geq dk/4] \geq 1/2$. We obtain $\Pr[|\mathcal{C}_{\mathcal{A}}(r(G_k^i))| \geq dk/4] \geq \Pr[|\mathcal{C}_{\mathcal{A}}(r(G_k^i))| \geq dk/4 \mid \mathcal{E}^i] \cdot \Pr[\mathcal{E}^i] \geq \frac{1}{2} \cdot \frac{1}{4} = \frac{1}{8}$. Equivalently, $\Pr[|\mathcal{C}_{\mathcal{A}}(r(G_k^i))| < dk/4] \leq 7/8$. If $|\mathcal{C}_{\mathcal{A}}(r(G_k))| < dk/4$, then $|\mathcal{C}_{\mathcal{A}}(r(G_k^i))| < dk/4$ must hold true for $i = 1, \ldots, 6$. The latter event occurs with probability at most $(7/8)^6$. We conclude $\Pr[|\mathcal{C}_{\mathcal{A}}(r(G_k))| \geq dk/4] \geq 1 - (7/8)^6 > 1/2$. This holds independently of $\mathcal{A}$'s coloring decisions made in other components.

Invariants (2–4) are immediate, based on the arguments given in the proof of Lemma 2. As for the number of vertices of any $G_k \in \mathcal{G}_k$, we observe that it is upper bounded by $12 \cdot d \cdot (12^{k-1} - 1) + 6 \cdot d/2 < d \cdot (12^k - 1)$.

If d is odd, the above construction of sets $\mathcal{G}_k$, $k \geq 1$, is performed for parameter $d-1$. In $\mathcal{G}_1$, graph G_1 is extended by a single vertex having edges to all other vertices in G_1. Invariant (5) holds because any graph $G_k \in \mathcal{G}_k$ contains 12^{k-1} copies of G_1.

The lemma follows from (1) and (3–5). In particular, (1) implies that the expected number of colors used by any deterministic online algorithm is at least $1/2 \cdot (d-1)k/4 = (d-1)k/8$. ◀

Proof of Theorem 3. For the given d and n, choose $k = \lfloor \log(n/d) \rfloor$. In this proof, logarithms are base 12. There holds $k \in \mathbb{N}$, because $n \geq 12d^2 > 12d$. By Lemma 4, there exists a probability distribution on a set $\mathcal{G}_k$ of chordal graphs with chromatic number d such that the expected number of colors used by every deterministic online algorithm is at least $(d-1)k/8$. The number of vertices of any graph in $\mathcal{G}_k$ is at most $d12^k$. Hence, by the choice of k, it is upper bounded by n. For every $G_k \in \mathcal{G}_k$, we add a suitable number of vertices so that the total number of vertices is equal to n. Every new vertex has one edge to an arbitrary vertex in the original graph G_k. Hence, there exists a probability distribution on a set of n-vertex graphs with chromatic number d such that the expected number of colors used by any deterministic online algorithm is at least $(d-1)k/8$. By Yao's principle [28], for every randomized online algorithm, there exists an n-vertex chordal graph G with $\chi(G) = d$ such that the expected number of color is $c_k \geq (d-1)k/8 \geq dk/16$. We have $k \geq \log n - \log d - 1 = \log(n/12) - \log d \geq 1/2 \cdot \log(n/12)$, because $12d^2 \leq n$, and hence $d \leq \sqrt{n/12}$. Since $12d^2 \leq n$, we have $\log(n/12) \geq 1/3 \cdot \log n$ and thus $c_k \in \Omega(d \cdot \log n)$. ◀

Again, in Theorem 3 we can reduce the lower bound on n from $12d^2$ to $12d^{1+\epsilon}$, for any $0 < \epsilon < 1$. The expected number of colors used by $\mathcal{A}$ is $\Omega(\epsilon \cdot d \cdot \log n)$.

4 Further graph classes

Given Theorem 3, we can derive lower bounds on the performance of randomized online coloring algorithms for other important graph classes.

4.1 Trees, planar, bipartite, d-inductive and bounded-treewidth graphs

▶ **Corollary 5.** *For every randomized online algorithm $\mathcal{A}$ and every $n \in \mathbb{N}$ with $n \geq 48$, there exists a n-vertex tree T, presented by an oblivious adversary, such that the expected number of colors used by $\mathcal{A}$ to color T is $\Omega(\log n)$.*

The proof is given in the full version of the paper. Since trees are planar and bipartite graphs, we obtain the following two corollaries.

▶ **Corollary 6.** *For every randomized online algorithm $\mathcal{A}$ and every $n \in \mathbb{N}$ with $n \geq 48$, there exists a n-vertex planar graph G, presented by an oblivious adversary, such that the expected number of colors used by $\mathcal{A}$ to color G is $\Omega(\log n)$.*

▶ **Corollary 7.** *For every randomized online algorithm $\mathcal{A}$ and every $n \in \mathbb{N}$ with $n \geq 48$, there exists a n-vertex bipartite graph G, presented by an oblivious adversary, such that the expected number of colors used by $\mathcal{A}$ to color G is $\Omega(\log n)$.*

Every chordal graph G is $(\chi(G) - 1)$-inductive and has treewidth $\omega(G) - 1 = \chi(G) - 1$ [7]. Hence, Theorem 3 gives the following two results.

▶ **Corollary 8.** *Let $d \in \mathbb{N}$ be an arbitrary positive integer. For every randomized online algorithm $\mathcal{A}$ and every $n \in \mathbb{N}$ with $n \geq 12d^2$, there exists a n-vertex d-inductive graph G, presented by an oblivious adversary, such that the expected number of colors used by $\mathcal{A}$ to color G is $\Omega(d \cdot \log n)$.*

▶ **Corollary 9.** *Let $d \in \mathbb{N}$ be an arbitrary positive integer. For every randomized online algorithm $\mathcal{A}$ and every $n \in \mathbb{N}$ with $n \geq 12d^2$, there exists a n-vertex graph G of treewidth d, presented by an oblivious adversary, such that the expected number of colors used by $\mathcal{A}$ to color G is $\Omega(d \cdot \log n)$.*

The graphs used in the proof of Theorem 3 are strongly chordal, which yields the following corollary. The proof can be found in the full version of the paper.

▶ **Corollary 10.** *Let $d \in \mathbb{N}$ be an arbitrary positive integer. For every randomized online algorithm $\mathcal{A}$ and every $n \in \mathbb{N}$ with $n \geq 12d^2$, there exists a n-vertex strongly chordal graph G with chromatic number $\chi(G) = d$, presented by an oblivious adversary, such that the expected number of colors used by $\mathcal{A}$ to color G is $\Omega(d \cdot \log n)$.*

4.2 Disk graphs

A disk graph is the intersection graph of disks in the Euclidean plane. Every vertex corresponds to a disk; two vertices are connected by an edge if the respective disks intersect. The following theorem implies that it is not possible to improve on the performance of deterministic online coloring algorithms by using randomization. We use the common assumption that when an online algorithm makes coloring decisions, it does not use the disk representation [9, 12, 13]. The proof of Theorem 11 is presented in the full version of the paper.

▶ **Theorem 11.** *Let $\mathcal{A}$ be an arbitrary randomized online algorithm. For every $n \in \mathbb{N}$ and $\rho \in \mathbb{R}$ with $\min\{n, \rho\} \geq 25$, there exists a n-vertex disk graph G with chromatic number $\chi(G) = 2$, presented by an oblivious adversary, in which the ratio of the largest to smallest disk radius is ρ, such that the expected number of colors used by $\mathcal{A}$ is $\Omega(\min\{\log n, \log \rho\})$.*

5 Lookahead and buffer reordering

Lookahead: We first assume that a randomized online coloring algorithm $\mathcal{A}$ has lookahead l. Theorem 12 below shows that, for chordal graphs, a lookahead of size $O(n/\log n)$ leads to no improvement. The proof is given in the full version of the paper.

▶ **Theorem 12.** *Let $d \in \mathbb{N}$ and $c \in \mathbb{R}$ be arbitrary numbers with $d \geq 2$ and $c \geq 1$. For every randomized online algorithm $\mathcal{A}$ with lookahead l and every $n \in \mathbb{N}$ with $n \geq \max\{12d^2, d \cdot 12^{2c}\}$ and $l \leq cn/\log(n/d)$, there exists a n-vertex chordal graph G with chromatic number $\chi(G) = d$, presented by an oblivious adversary, such that the expected number of colors used by $\mathcal{A}$ to color G is $\Omega(\frac{1}{c} \cdot d \cdot \log n)$.*

Based on Theorem 12 we can derive analogous results for all the other graph classes considered in Section 4. Loosely speaking, a lookahead of size $O(n/\log n)$ is of no help. The next Corollary 13 addresses trees. Exactly the same statement holds for planar and bipartite graphs, respectively. For brevity, we omit the corresponding corollaries.

▶ **Corollary 13.** *Let $c \geq 1$ be an arbitrary real number. For every randomized online algorithm $\mathcal{A}$ with lookahead l and every $n \in \mathbb{N}$ with $n \geq \max\{48, 2 \cdot 12^{2c}\}$ and $l \leq cn/\log(n/2)$, there exists a n-vertex tree G, presented by an oblivious adversary, such that the expected number of colors used by $\mathcal{A}$ to color G is $\Omega(\frac{1}{c} \cdot \log n)$.*

For d-inductive graphs, graphs of treewidth d and strongly chordal graphs with chromatic number d, the formulation of Theorem 12 directly carries over. In fact, the result holds for all integers $d \geq 1$. For disk graphs, Theorems 11 and 12 give the following corollary.

▶ **Corollary 14.** *Let $c \in \mathbb{R}$ with $c \geq 1$ be arbitrary. For every randomized online algorithm $\mathcal{A}$ with lookahead l, every $n \in \mathbb{N}$ and $\rho \in \mathbb{R}$ with $\min\{n, \rho\} \geq 2 \cdot 12^{2c}$ and $l \leq cn/\log(n/2)$, there exists a n-vertex disk graph G with chromatic number $\chi(G) = 2$, presented by an oblivious adversary, in which the ratio of the largest to smallest disk radius is ρ, such that the expected number of colors used by $\mathcal{A}$ to color G is $\Omega(\frac{1}{c} \cdot \log n)$.*

Buffer reordering: Next we examine the setting in which a deterministic online coloring algorithm $\mathcal{A}$ has a reordering buffer. We prove that a buffer of size $n^{1-\epsilon}$, for any $0 < \epsilon \leq 1$, does not improve the asymptotic performance of the algorithms.

▶ **Theorem 15.** *Let $d \in \mathbb{N}$ and $\epsilon \in \mathbb{R}$ be arbitrary numbers with $d \geq 2$ and $0 < \epsilon \leq 1$. For every deterministic online algorithm $\mathcal{A}$ having a buffer of size b and every $n \in \mathbb{N}$ with $b \leq n^{1-\epsilon}$ and $n \geq \max\{2d^2, 2^{7/\epsilon}\}$, there exists a n-vertex chordal graph G with chromatic number $\chi(G) = d$ such that the number of colors used by $\mathcal{A}$ is $\Omega(\epsilon \cdot d \cdot \log n)$.*

The proof of Theorem 15 is presented in the full version of the paper. Given Theorem 15, we derive analogous results for the other graph classes. Corollary 16 shows a result for trees. Identical statements hold for planar and bipartite graphs. Again, for brevity, we omit the corresponding corollaries.

▶ **Corollary 16.** *Let $\epsilon \in \mathbb{R}$ with $0 < \epsilon \leq 1$ be arbitrary. For every deterministic online algorithm $\mathcal{A}$ having a buffer of size b and every $n \in \mathbb{N}$ with $b \leq n^{1-\epsilon}$ and $n \geq 2^{7/\epsilon}$, there exists a n-vertex tree G such that the number of colors used by $\mathcal{A}$ is $\Omega(\epsilon \cdot \log n)$.*

For d-inductive graphs, graphs of treewidth d and strongly chordal graphs with chromatic number d, the statement of Theorem 15 directly carries over. In this case it holds for any $d \geq 1$. The corollaries are omitted here. Finally, we give a result for disk graphs.

▶ **Corollary 17.** *Let $\mathcal{A}$ be an arbitrary deterministic online algorithm having a buffer of size b and let $\epsilon \in \mathbb{R}$ be an arbitrary real number with $0 < \epsilon \leq 1$. For every $n \in \mathbb{N}$ and $\rho \in \mathbb{R}$ with $b \leq \min\{n^{1-\epsilon}, \rho^{1-\epsilon}\}$ and $\min\{n, \rho\} \geq 2^{7/\epsilon}$, there exists a n-vertex disk graph G with chromatic number $\chi(G) = 2$, in which the ratio of the largest to smallest disk radius is ρ, such that the number of colors used by $\mathcal{A}$ is $\Omega(\epsilon \cdot \min\{\log n, \log \rho\})$.*

Acknowledgments. We thank anonymous referees for their valuable comments.

References

1. N. Avigdor-Elgrabli and Y. Rabani. An optimal randomized online algorithm for reordering buffer management. In *Proc. 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1–10, 2013.
2. A. Bar-Noy, P. Cheilaris, S. Olonetsky, and S. Smorodinsky. Online conflict-free colouring for hypergraphs. *Combinatorics, Probability & Computing*, 19(4):493–516, 2010.
3. A. Bar-Noy, P. Cheilaris, and S. Smorodinsky. Deterministic conflict-free coloring for intervals: From offline to online. *ACM Trans. Algorithms*, 4(4):44:1–44:18, 2008.
4. D. Bean. Effective coloration. *J. Symbolic Logic*, 41(2):469–480, 1976.
5. S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11(1):2–14, 1994.
6. M. P. Bianchi, H.-J. Böckenhauer, J. Hromkovic, and L. Keller. Online coloring of bipartite graphs with and without advice. *Algorithmica*, 70(1):92–111, 2014.
7. H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2):1–21, 1993.
8. E. Burjons, J. Hromkovic, X. Muñoz, and W. Unger. Online graph coloring with advice and randomized adversary. In *Proc. 42nd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'16)*, pages 229–240. Springer LNCS 9587, 2016.
9. I. Caragiannis, A. V. Fishkin, C. Kaklamanis, and E. Papaioannou. A tight bound for online colouring of disk graphs. *Theoretical Computer Science*, 384(2):152–160, 2007.
10. R. G. Downey and C. McCartin. Online promise problems with online width metrics. *Journal of Computer and System Sciences*, 73(1):57–72, 2007.
11. M. Englert, D. Özmen, and M. Westermann. The power of reordering for online minimum makespan scheduling. *SIAM J. Comput.*, 43(3):1220–1237, 2014.
12. T. Erlebach and J. Fiala. On-line coloring of geometric intersection graphs. *Computational Geometry*, 23(2):243–255, 2002.
13. T. Erlebach and J. Fiala. Independence and coloring problems on intersection graphs of disks. In E. Bampis, K. Jansen, and C. Kenyon, editors, *Efficient Approximation and Online Algorithms: Recent Progress on Classical Combinatorial Optimization Problems and New Applications*, pages 135–155. Springer LNCS 3484, 2006.
14. Martin Farber. Characterizations of strongly chordal graphs. *Discrete Mathematics*, 43(2-3):173–189, 1983.
15. A. Gyárfás and J. Lehel. On-line and first fit colorings of graphs. *Journal of Graph Theory*, 12(2):217–227, 1988.
16. M. M. Halldórsson. Parallel and on-line graph coloring. *J. Algorithms*, 23(2):265–280, 1997.
17. M. M. Halldórsson and M. Szegedy. Lower bounds for on-line graph coloring. *Theoretical Computer Science*, 130(1):163–174, 1994.
18. S. Irani. Coloring inductive graphs on-line. *Algorithmica*, 11(1):53–72, 1994. Preliminary version in *FOCS'90*.
19. H. A. Kierstead. Coloring graphs on-line. In A. Fiat and G. J. Woeginger, editors, *Online Algorithms*, pages 281–305. Springer LNCS 1442, 1998.

20 H. A. Kierstead and W. A. Trotter. An extremal problem in recursive combinatorics. *Congressus Numerantium*, 33:143–153, 1981.

21 S. Leonardi and A. Vitaletti. Randomized lower bounds for online path coloring. In *Proc. 2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM'98)*, pages 232–247. Springer LNCS 1518, 1998.

22 L. Lovász, M. Saks, and W. T. Trotter. An on-line graph coloring algorithm with sublinear performance ratio. *Annals of Discrete Mathematics*, 43:319–325, 1989.

23 D. Marx. Graph colouring problems and their applications in scheduling. *Periodica Polytechnica, Electrical Engineering*, 48(1–2):11–16, 2004.

24 L. Narayanan. Channel assignment and graph multicoloring. *Handbook of Wireless Networks and Mobile Computing*, pages 71–94, 2004.

25 D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

26 S. Vishwanathan. Randomized online graph coloring. *J. Algorithms*, 13(4):657–669, 1992. Preliminary version in *FOCS'90*.

27 D. B. West. *Introduction to Graph Theory, 2nd Edition.* Pearson, 2001.

28 A. C. C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proc. 18th Annual Symposium on Foundations of Computer Science*, pages 222–227, 1977.

Combinatorics of Local Search: An Optimal 4-Local Hall's Theorem for Planar Graphs*

Daniel Antunes[1], Claire Mathieu[2], and Nabil H. Mustafa[3]

1 **Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, Equipe A3SI, ESIEE, Paris, France**
daniel.antunes@esiee.fr

2 **Department of Computer Science, CNRS, École Normale Supérieure and PSL Research University, Paris, France**
Claire.Mathieu@ens.fr

3 **Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, Equipe A3SI, ESIEE, Paris, France**
mustafan@esiee.fr

Abstract

Local search for combinatorial optimization problems is becoming a dominant algorithmic paradigm, with several papers using it to resolve long-standing open problems. In this paper, we prove the following '4-local' version of Hall's theorem for planar graphs: given a bipartite planar graph $G = (B, R, E)$ such that $|N(B')| \geq |B'|$ for all $|B'| \leq 4$, there exists a matching of size at least $\frac{|B|}{4}$ in G; furthermore this bound is tight. Besides immediately implying improved bounds for several problems studied in previous papers, we find this variant of Hall's theorem to be of independent interest in graph theory.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Combinatorial Optimization, Planar Graphs, Local Search, Hall's Theorem, Expansion

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.8

1 Introduction

One of the exciting developments in the field of geometric algorithms in recent years has been the use of *local search* techniques to resolve several open problems in combinatorial optimization. Remarkably, all these following NP-hard problems are approximately solved by the same meta-algorithm:

1. *Minimum hitting set problem for pseudo-disks*[1] [16]. Given a set X of points and a set $\mathcal{D}$ of pseudo-disks in the plane, compute a minimum size subset of X that hits all pseudo-disks in $\mathcal{D}$.
2. *Maximum independent set in the intersection graph of pseudo-disks* [1, 8]. Given a set $\mathcal{D}$ of pseudo-disks in the plane, compute a maximum size pairwise disjoint subset of $\mathcal{D}$.
3. *Terrain guarding problem* [10]. Given a 1.5D terrain[2] T and two subsets $X, G \subseteq T$, compute a minimum size subset of G such that every point of X is visible from some point of G.

* This work was supported by the grant ANR SAGA (JCJC-14-CE25-0016-01).

[1] A set of geometric objects in the plane are called pseudo-disks if the boundary of every pair of objects intersect at most twice.

[2] A 1.5D terrain T is an x-monotone chain of line segments in $\mathbb{R}^2$.

4. *Minimum dominating set in disk intersection graphs* [11]. Given a set $\mathcal{D}$ of disks in the plane, compute a minimum size subset $\mathcal{D}' \subseteq \mathcal{D}$ such that each $D \in \mathcal{D}$ is either in $\mathcal{D}'$ or intersects some disk in $\mathcal{D}'$.
5. *Minimum dominating set in pseudo-disk intersection graphs* [12]. Given a set $\mathcal{D}$ of pseudo-disks in the plane, compute a minimum size subset $\mathcal{D}'$ of $\mathcal{D}$ such that each $D \in \mathcal{D}$ is either in $\mathcal{D}'$ or intersects some pseudo-disk in $\mathcal{D}'$.
6. *Minimum set-cover problem for disks in the plane* [7, 15]. Given a set of points X and a set of disks $\mathcal{D}$ in the plane, compute the minimum sized subset of $\mathcal{D}$ that covers all the points of X. This problem can be reduced to the minimum hitting set problem for disks.

The Meta-Algorithm: Local Search

The meta-algorithm can be parameterized by an integer k representing the search radius. Abstractly, let X be a set of given base elements, and $\Pi : 2^X \longrightarrow \{0, 1\}$ be a function that assigns feasibility to each subset of X with respect to the specific problem. Then the goal is to find a minimum/maximum sized subset of X for which $\Pi(\cdot)$ is feasible. The local-search algorithm proceeds as follows: start with any feasible solution $\mathcal{S} \subseteq X$, and iteratively improve $\mathcal{S}$ by changing[3] subsets of $\mathcal{S}$ of size at most k, as long as the new solution is also feasible. We restrict the discussion below to instances of minimization problems; the maximization case is similar.

Local-Search Method With Search Radius k (minimization instance).

Let $\mathcal{S} \subseteq X$ be any feasible solution.
while *there exists* $\mathcal{S}'$ *with* $\pi(\mathcal{S}')$ *feasible and where* $|\mathcal{S}' \setminus \mathcal{S}| < |\mathcal{S} \setminus \mathcal{S}'| \leq k$ **do**
  set $\mathcal{S} = \mathcal{S}'$.
return $\mathcal{S}$

The analysis of the approximation factor of a local search algorithm, assuming the problem has some planar features, usually proceeds as follows.

Recall that for a graph $G = (V, E)$ and a subset V' of V, $N_G(V') = \{v \in V : \exists u \in V', \{u, v\} \in E\}$ denotes the set of neighbors of V' in G.

▶ **Definition 1.** Let $k \geq 1$ be given. A bipartite graph $G = (B, R, E)$ satisfies a *local expansion property* if, for every subset B' of B of cardinality at most k, we have $|N_G(B')| \geq |B'|$. Then G is called a k-expanding graph. If $k = |B|$ then G is called an expanding graph.

▶ **Lemma 2** ([8, 16])**.** *There is an absolute constant c_0 such that any planar bipartite k-expanding graph $G = (B, R, E)$ satisfies $|R| \geq \big(1 - \frac{c_0}{\sqrt{k}}\big)|B|$.*

The analysis of local-search algorithm with search radius k proceeds by first constructing a certain bipartite planar graph $G = (\mathcal{S}, \mathcal{O}, E)$ on $\mathcal{S}$ and $\mathcal{O}$, where $\mathcal{S}$ is the local-search solution with radius k and $\mathcal{O}$ is an (unknown) optimal solution, such that G is k-expanding.

Now setting $k = \Theta(\frac{1}{\epsilon^2})$ and applying Lemma 2 to G implies that the local optimum $\mathcal{S}$ has size $\big(1 + O(\epsilon)\big)$ times the optimal size $|\mathcal{O}|$, hence near-optimality. A straightforward implementation of the local-search algorithm gives a running time of $n^{O(\frac{1}{\epsilon^2})}$, so this is a PTAS (polynomial-time approximation scheme). Note that as most of the problems listed earlier are $W[1]$-hard [13, 14], it is unlikely that algorithms exist that do not have a dependency on $1/\epsilon$ in the exponent of n.

[3] In case of a minimization problem, replace some k elements of $\mathcal{S}$ with some $k - 1$ elements of X; for a maximization problem replace some k elements of $\mathcal{S}$ with some $k + 1$ elements of X.

Combinatorics of Local Search: Hall's Theorem for Planar Graphs

The reader will notice the resemblance between the Local Expansion Property and pre-conditions of Hall's theorem – Local Expansion Property is simply the pre-condition of Hall's theorem restricted to subsets of size at most k. And indeed, the statement of Lemma 2 can be re-cast as a 'local' version of Hall's theorem for planar graphs, as follows. One of the cornerstones of graph theory, Hall's theorem, can be rephrased as:

▶ **Theorem 3** (Hall's Theorem). *Let $G = (B, R, E)$ be a $|B|$-expanding bipartite graph. Then there exists a matching in G of size $|B|$.*

Note that if we restrict the expanding subsets to be of size at most k for some integer k, then the theorem fails, as one cannot guarantee a matching of size more than k – e.g., take G to be the complete bipartite graph $K_{|B|,k}$. Interestingly, Lemma 2 implies that unlike the general graph case, a 'local' version of Hall's theorem is indeed true for planar graphs. We first observe that Lemma 2 can be used to get a local variant of Hall's theorem for planar graphs:

▶ **Theorem 4** (k-local Hall's Theorem for Planar Graphs). *Let $G = (B, R, E)$ be a k-expanding bipartite planar graph. Then there exists a matching in G of size at least $\left(1 - \frac{c_0}{\sqrt{k}}\right)|B|$.*

Proof. Let $B' \subseteq B$ for any subset of B. Observing that the subgraph of G induced by $B' \cup N_G(B')$ is planar, bipartite and k-expanding, we have $|N_G(B')| \geq \left(1 - \frac{c_0}{\sqrt{k}}\right)|B'|$ by Lemma 2. Let S be a new set of $\frac{c_0|B|}{\sqrt{k}}$ dummy vertices. Construct a bipartite graph $G' = (B, R \cup S, E \cup E')$, where E' is the set of all $|B| \cdot |S|$ edges between B and S. Then G' satisfies the conditions of Hall's theorem, as for any $B' \subseteq B$, we have

$$|N_{G'}(B')| = |N_G(B')| + |S| \geq \left(1 - \frac{c_0}{\sqrt{k}}\right)|B'| + \frac{c_0|B|}{\sqrt{k}} \geq |B'|.$$

Thus there is a matching of size $|B|$ in G' by Hall's theorem. Removing the vertices of S from this matching still leaves a matching of size at least $\left(1 - \frac{c_0}{\sqrt{k}}\right)|B|$. ◀

Note that Theorem 4 is more general than Lemma 2, so it can be interpreted as a strengthening of Lemma 2. Summarizing this discussion, the above local version of Hall's theorem for planar graphs is the key combinatorial reason why local-search works for a wide variety of geometric optimization problems. The proof of Lemma 2 relies on separators in planar graphs, and there has been work in generalizing these ideas to classes of non-planar graphs which still have small separators (see [6, 2, 5]).

Our Results

While local-search with search radius $k = \Theta(\frac{1}{\epsilon^2})$ theoretically gives the best possible result in terms of approximation factors, these problems are far from being solved satisfactorily:

- As stated earlier, most of these problems are $W[1]$-hard [13, 14]: therefore unless $W[1] = FTP$, there is no efficient polynomial-time approximation scheme for most of the listed problems; i.e., algorithms with running time $O(n^c)$, where c is a constant independent of $\frac{1}{\epsilon}$. This effectively restricts local search to small constant values of k.
- Furthermore, local-search is often the *only* approach known for these problems that yields good approximations. For example, the best approximation ratio for the hitting set problem for disks without using local-search is 13.4 [4] via the theory of ϵ-nets (see the chapter [17] for details); or $O(\log n)$-approximation for dominating sets in disk intersection

graphs [11]. Any effective solution to these problems entails examining closely the limits of efficiency and quality of local search for small values of k.

- While the construction of the graph is specific to the problem at hand, all these algorithms rely on the same Local Expansion Property of planar graphs, and thus the quantitative approximation bounds are the same across all the problems. The constants involved in Theorem 4 unfortunately make this result inefficient even for small values of k; e.g., the current best work shows that setting k to get a 3-approximation implies a running time of $\Omega(n^{66})$ for the hitting set problem for disks [9].

Thus the natural way forward is to explore the limits of local search for small values of k. In this paper, we will consider the combinatorial aspect, and evaluate the quality of local-search – alternatively, the precise statement of local Hall's theorem for planar graphs:

- $\mathbf{k = 1, 2}$. The local Hall's theorem *fails* (and so does local search) for the same reason as for general graphs – $K_{|B|,2}$ is a 2-expanding planar graph, but with a matching of size only 2.
- $\mathbf{k = 3}$. An optimal local Hall's theorem was shown in [3] by a short argument: any planar bipartite 3-expanding graph has a matching of size $\frac{|B|}{8}$ and this is tight.

The next fundamental case of local search that is open is for $k = 4$; the previous-best bound was $\frac{|B|}{5}$ and the resolution of the optimal bound was the main problem left open in [3]. In this paper we settle this question by presenting an optimal bound for local Hall's theorem for 4-expanding planar graphs.

▶ **Theorem 5** (Main Theorem). *Let $G = \big(B, R, E\big)$ be a bipartite planar graph on vertex sets R and B, such that G is 4-expanding; i.e., for all $B' \subseteq B$ with $|B'| \leq 4$, $|N_G(B')| \geq |B'|$. Then there exists a matching in G of size at least $\frac{|B|}{4}$. Furthermore, this bound is tight up to lower-order terms.*

▶ **Corollary 6.** *The local search algorithm with parameter $k = 4$ gives a 4-approximation to these problems in geometric combinatorial optimization:*

1. *Minimum hitting set problem for pseudo-disks in the plane.*
2. *Maximum independent set problem in the intersection graph of pseudo-disks.*
3. *Terrain guarding problem.*
4. *Minimum dominating set in the pseudo-disk intersection graphs.*
5. *Minimum set-cover problem for disks in the plane.*

Tightness

The optimality of the bound follows from the example shown in Figure 1, where R consists of n vertices of a $\sqrt{n} \times \sqrt{n}$ grid, and each 'grid cell' contains 4 vertices of B connected to the four red vertices of that cell. It is easy to verify that there is no matching of size greater than $\frac{|B|}{4} + O(\sqrt{|B|})$ (this is trivial, as $|B| = 4n - O(\sqrt{n})$), and the graph is planar and bipartite.

Finally, the fact that it is 4-expanding follows from the observation that, except at the grid boundary, any set of two vertices of B of degree 3 or any set of three vertices of B of degree 2 has at least 4 neighbors in R.

The proof of the upper-bound relies on the following key lemma, presented in Section 2:

▶ **Lemma 7.** *Let $G = \big(B, R, E\big)$ be a bipartite planar graph on vertex sets R and B, such that G is 4-expanding. Then $|R| \geq \frac{|B|}{4}$.*

Lemma 7 can be seen as a version of Lemma 2 for $k = 4$ and $c_0 = \frac{3}{2}$, leading to the Main Theorem via an argument identical to the proof of Theorem 4.

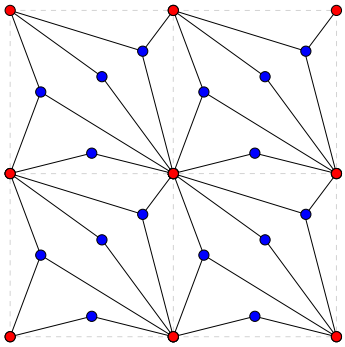


Figure 1 A lower-bound construction for 4-expanding bipartite planar graphs.

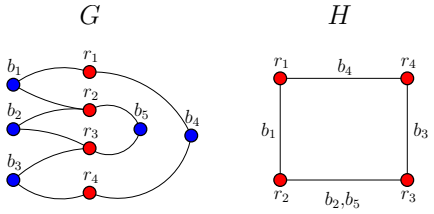


Figure 2 A bipartite planar graph $G(B, R, E)$ and its corresponding graph $H(R, E)$.

2 Proof of Lemma 7

The proof, at its core, uses the *discharging method* [18] of combinatorial geometry. Henceforth, a graph satisfying 4-expanding property is said to satisfy 4L.

First note that no vertex in B can have degree zero, as otherwise the neighborhood of such a vertex would violate 4L. Moreover, it can be assumed that every vertex in B has degree at least two, since it is always possible to add edges to all vertices of B which have degree one in G while maintaining the planarity and bipartiteness of the graph (as any such vertex v must lie in a face which has at least two vertices of R, at least one of which is not adjacent to v).

Let $B_{=i} \subseteq B$ be the subset of vertices of B of degree exactly i, and $B_{\geq i} \subseteq B$ the set of vertices of degree at least i.

For the remainder of the proof, we fix a planar embedding of G.

Let $H(R, E)$ be a planar graph on R constructed from G as follows: two vertices $r_1 \in R$ and $r_2 \in R$ are adjacent in H iff there is at least one vertex $b \in B_{=2}$ which is adjacent to both r_1 and r_2 in G. Note that H is planar since G is planar, and the edges between r_1 and r_2 can be routed via one such vertex b. Note also that vertices in $B_{=3}$ lie in the interior of faces of H. Vertices of R will be called the *red* vertices, and vertices of B the *blue* vertices.

Note that for a fixed pair $\{r_1, r_2\} \subseteq R$, there cannot be three distinct vertices b_1, b_2, $b_3 \in B_{=2}$ adjacent to both r_1 and r_2, since in this case the neighborhood of set $\{b_1, b_2, b_3\}$ is of size two and the graph G would violate 4L. Therefore, each edge of H corresponds to one or two vertices in $B_{=2}$. Edges corresponding to a single vertex in $B_{=2}$ are called *single edges* and the set of all such edges is denoted by E_1, while edges mapped to two vertices in $B_{=2}$ are called *double edges* and its set is denoted by E_2. In Figure 2, $\{r_1, r_2\}$ is a single edge and $\{r_2, r_3\}$ is a double edge. In later figures, the numbers 1 and 2 will be used to indicate whether an edge is single or double. When referring to a particular face f, ∂f will denote its set of edges while E_1^f and E_2^f will denote the set of single and double edges of f, respectively.

For the rest of the proof, fix an embedding of H as well as the counter-clockwise ordering on $\partial_V f$ for each $f \in F$, where $\partial_V f$ denotes the vertices of f. Let F_i be the set of faces of H with exactly i edges on its boundary, and let F be the set of all faces of H. A face in F_3 will be called a *triangular* face and a face in F_4 a *rectangular* face. If ∂f is a cycle then f is called a *face cycle*. An edge e on the boundary of two different faces is called a *boundary edge*; it is called a *cut edge* otherwise.

In proceeding with the proof, we now encounter a technical difficulty: H need not be 2-connected, and so the structure of the faces can be arbitrarily complex. We first prove, in the next subsection, Lemma 7 for the case when H is 2-connected. Then we show how to handle the general case by reducing it to the 2-connected case.

2.1 Case: $H(R, E)$ is 2-connected

If H is 2-connected then all its faces are face cycles; in particular, each edge of H is a boundary edge, and there are no cut edges.

2.1.1 Structural properties of H

▶ **Claim 8.** *For $i \geq 4$, let $f \in F_i$. Then $|E_2^f| \leq \lfloor \frac{i}{2} \rfloor$. A triangular face has no double edges.*

Proof. Let f be a triangular face with vertices $\{r_1, r_2, r_3\}$, and with, say, $\{r_1, r_2\} \in E_2^f$. Recall that edges of H are associated with vertices of $B_{=2}$. Thus the two single edges and one double edge of f correspond to a set $B' \subseteq B$ of four vertices, with $N(B') = \{r_1, r_2, r_3\}$, violating 4L. For $i \geq 4$, if a face $f \in F_i$ has $|E_2^f| > \lfloor \frac{i}{2} \rfloor$, then there must exist two double edges incident to the same vertex of f and 4L is again violated. ◀

For a face $f \in F_i$, f is called a *full face* if $|E_2^f| = \lfloor \frac{i}{2} \rfloor$. Let $B_{=3}^f$ denote the set of $B_{=3}$ vertices lying in the interior of f. Note that due to planarity, for a fixed face f in the embedding of H, each vertex $v \in B_{=3}^f$ can be written uniquely (up to rotation) as an ordered triple $v = (r_1, r_2, r_3)$, where $r_1, r_2, r_3 \in R$ are vertices of f in counter-clockwise order with $\{v, r_i\} \in E(G)$ for $i = 1, 2, 3$.

▶ **Claim 9.** *For $i \geq 4$, let $f \in F_i$. Then $|B_{=3}^f| \leq (i-2)$.*

Proof. Note that we can assume that $|E_2^f| = 0$, as a double edge can only make it harder to 'pack' more vertices of $B_{=3}$ into f. Define a chain τ of f to be a consecutive set of vertices of $\partial_V f$. The size $|\tau|$ of a chain is equal to its number of vertices, and define $B_{=3}^\tau$, in the natural way, as the set of vertices of $B_{=3}^f$ with edges only to vertices of τ. We show that for a chain τ of size n, $|B_{=3}^\tau| \leq n-2$. The proof will be by induction on the size of τ. For $|\tau| = 2$, $|B_{=3}^\tau| = 0$, trivially. For $|\tau| = j$, any fixed $v \in B_{=3}^\tau$ divides τ into three distinct sub-chains, τ_1, τ_2, τ_3, with $|\tau_1| + |\tau_2| + |\tau_3| = j + 3$. Applying the induction hypothesis on each sub-chain,

$$|B_{=3}^\tau| \leq (|\tau_1| - 2) + (|\tau_2| - 2) + (|\tau_3| - 2) + 1 = j + 3 - 6 + 1 = j - 2.$$ ◀

For the next steps, we will need the list of 'forbidden' substructures in graphs satisfying 4L.

▶ **Claim 10.** *H satisfies* 4L *if and only if it does not contain the structures shown in Figure 3.*

For the next claim, we will need the following independent property for planar graphs.

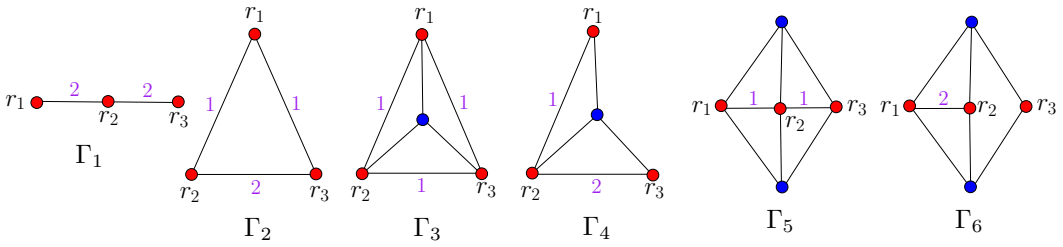


Figure 3 Forbidden structures for a graph H satisfying 4L.

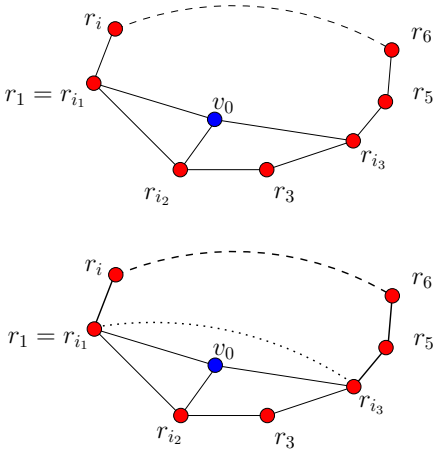


Figure 4 The vertex v_0 divides the graph in two regions.

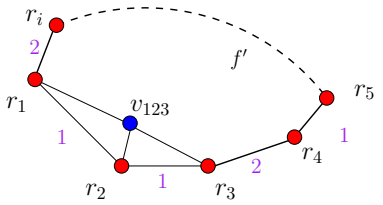


Figure 5 An odd full face. There exist two consecutive single edges.

▶ **Claim 11.** *Let G be a planar graph consisting of one (external) cycle $C = \langle r_1, \ldots, r_i \rangle$ of i vertices and a set V of internal vertices, such that each vertex of V has exactly three neighbors, all in C, with these three neighbors not being consecutive vertices of C. Then $|V| \leq i - 4$.*

Proof. The proof is inductive. For $i = 4$, we have $|V| = 0 = i - 4$, as there cannot exist a vertex not adjacent to three consecutive vertices of C. Consider the case where $i \geq 5$. By an extremal argument, there must exist a vertex $v_0 \in V$, say connected to $\{r_{i_1}, r_{i_2}, r_{i_3}\}$ where we can assume without loss of generality that $1 = i_1 < i_2 < i_3$, such that the two regions – one with boundary vertices $\langle v_0, r_{i_1}, r_{i_1+1}, \ldots, r_{i_2} \rangle$ and the other with boundary vertices $\langle v_0, r_{i_2}, r_{i_2+1}, \ldots, r_{i_3} \rangle$ – are both empty of vertices of V (see Figure 4). Furthermore, by the assumption that v does not have edges to three consecutive vertices of C, we have $(i_3 - i_1) \geq 3$. If there exists a vertex, other than v_0, in V with edges to both r_{i_1} and r_{i_3}, call it v_1 (note that due to planarity, there can exist only one such vertex). Consider a new cycle $C' = \langle r_1, r_{i_3}, r_{i_3+1}, \ldots, r_i \rangle$ of size $i - (i_3 - i_1) + 1 \leq (i - 2)$, and set $V' = V \setminus \{v_0, v_1\}$ to be a subset of vertices lying inside C'. It is easy to see that no vertex of V' can have edges to three consecutive vertices of C', and thus by induction, we have $|V'| \leq |C'| - 4 \leq (i - 2) - 4$, and thus $|V| \leq |V'| + 2 \leq (i - 4)$. ◀

▶ **Claim 12.** *For $i \geq 4$, let $f \in F_i$ be a full face. If i is even then $|B^f_{=3}| \leq (i - 4)$. If i is odd then $|B^f_{=3}| \leq (i - 3)$.*

Proof. Let $f \in F_i$ and $|E^f_{=2}| = \lfloor \frac{i}{2} \rfloor$. Label the vertices of $\partial_V f$ as $\langle r_1, \ldots, r_i \rangle$ in the assumed counter-clockwise ordering.

***i* is even:** Note that as f is full, the edges around f alternate between single and double edges. Therefore 4L implies that there does not exist a $v \in B^f_{=3}$ with edges to three consecutive vertices of $\partial_V f$. Claim 11 applied to f shows that $|B^f_{=3}| \leq i - 4$.

***i* is odd:** For $i \geq 5$, as f is a full face, the edges around f alternate between single and double edges – with one exception where two adjacent edges are both single. Say these adjacent single edges are $\{r_1, r_2\}$ and $\{r_2, r_3\}$ (see Figure 5). 4L implies that there does not exist a $v \in B^f_{=3}$ with edges to three consecutive vertices of $\partial_V f$, *except* possibly there could exist a single vertex $v_{123} \in B^f_{=3}$ with edges to $\{r_1, r_2, r_3\}$. Claim 11 applied to f shows that $|B^f_{=3} \setminus \{v_{123}\}| \leq i - 4$, and thus $|B^f_{=3}| \leq i - 3$. ◀

2.1.2 Bounding $|B|$

We first observe that to bound the size of B, it suffices to bound the number of vertices of degree 2 and 3 in B. We will need the following fact on planar graphs.

▶ **Fact 13.** *Let $G = (V, E)$ be a simple, connected, planar bipartite graph. Then $|E| \leq 2|V| - 4$.*

▶ **Claim 14.** $|B| \leq |B_{=2}| + \frac{|B_{=3}|}{2} + |R|$.

Proof. We count the number of edges in G in two ways – first by summing up the degrees of the vertices in B (recall that G is a bipartite graph), and secondly by using the upper-bound on the number of edges of planar bipartite graphs from Fact 13:

$$2 \cdot |B_{=2}| + 3 \cdot |B_{=3}| + \sum_{i=4} i \cdot |B_{=i}| = \quad |E(G)| \quad \leq 2\big(|R| + |B|\big) - 4.$$

Simplifying,

$$2 \cdot |B_{=2}| + 3 \cdot |B_{=3}| + \sum_{i=4} i \cdot |B_{=i}| \leq 2 \cdot \Big(|R| + |B_{=2}| + |B_{=3}| + \sum_{i=4} |B_{=i}| \Big).$$

Re-arranging the terms,

$$\sum_{i=4} (i-2) \cdot |B_{=i}| \leq 2|R| - |B_{=3}| \implies 2 \sum_{i=4} |B_{=i}| \leq 2|R| - |B_{=3}|$$

$$|B_{\geq 4}| \leq |R| - \frac{|B_{=3}|}{2}. \tag{1}$$

Now one can get an upper-bound on $|B|$ from inequality (1):

$$|B| = |B_{=2}| + |B_{=3}| + |B_{\geq 4}| \leq |B_{=2}| + |B_{=3}| + |R| - \frac{|B_{=3}|}{2} = |B_{=2}| + \frac{|B_{=3}|}{2} + |R|. \quad \blacktriangleleft$$

Thus it remains to bound $|B_{=2}| + \frac{|B_{=3}|}{2}$. Towards this, a charging intuition leads one to classify the contribution of a face $f \in F$ as $2 \cdot |E^f_2| + |E^f_1| + \frac{|B^f_{=3}|}{2}$. It turns out that the right discharging function is slightly different; define the *weight* of a face $f \in F$ to be

$$w(f) = |E^f_2| + \frac{|E^f_1|}{2} + \frac{|B^f_{=3}|}{2},$$

and the *weight* of the graph H to be

$$w(H) = |B_{=2}| + \frac{|B_{=3}|}{2}.$$

Note that as each edge is part of the boundary of precisely two faces and each vertex of $B_{=3}$ lies in precisely one face, we have

$$\sum_{f\in F} w(f) = \sum_{f\in F}\Big(|E_2^f| + \frac{|E_1^f|}{2} + \frac{|B_{=3}^f|}{2}\Big) = 2\cdot|E_2| + |E_1| + \frac{|B_{=3}|}{2} = |B_{=2}| + \frac{|B_{=3}|}{2} = w(H). \quad (2)$$

▶ **Claim 15.** $w(H) \le \frac{1}{4}\sum_{i\ge 3}(5i-6)|F_i| - \frac{1}{4}\sum_{i \text{ is odd}}|F_i| - \frac{1}{2}|F_4| - \frac{1}{2}|F_3|$.

Proof. Let $I_f \in \{0,1\}$ be an indicator variable such that $I_f = 1$ if and only if f is a full face. For $f \in F_i$ and i an even number, by applying the upper bounds in Claims 8, 9 and 12,

$$w(f) = |E_2^f| + \frac{|E_1^f|}{2} + \frac{|B_{=3}^f|}{2} \le \Big(\frac{i}{2} - 1 + I_f\Big) + \frac{i - \Big(\frac{i}{2} - 1 + I_f\Big)}{2} + \frac{(i-2) - 2I_f}{2}$$
$$= \frac{5i - 6 - 2I_f}{4} \le \frac{5i-6}{4}.$$

For $i = 4$, a better bound is possible. For a face $f \in F_4$, let $\alpha^f = |E_2^f|$. Then

$$w(f) \le \alpha^f + \frac{4-\alpha^f}{2} + \frac{(4-2)-\alpha^f}{2} = \frac{4\cdot 4 - 4}{4} = \frac{5\cdot 4 - 6}{4} - \frac{1}{2} = \frac{5i-6}{4} - \frac{1}{2}. \quad (3)$$

For i an odd number,

$$w(f) \le \Big(\frac{i-1}{2} - 1 + I_f\Big) + \frac{i - \Big(\frac{i-1}{2} - 1 + I_f\Big)}{2} + \frac{(i-2) - I_f}{2} = \frac{5i-7}{4}. \quad (4)$$

For $i = 3$, note that for a face $f \in F_3$, $|E_2^f| = 0$, $|E_1^f| = 3$ and $|B_{=3}^f| = 0$, since f cannot have neither a $B_{=3}$ vertex in its interior nor a double edge, as otherwise the forbidden structures Γ_3 or Γ_2 would be present. Then,

$$w(f) = |E_2^f| + \frac{1}{2}|E_1^f| + \frac{|B_{=3}^f|}{2} = \frac{3}{2} = \frac{5i-7}{4} - \frac{1}{2}. \quad (5)$$

By Equations (2)–(5),

$$w(H) = \sum_{f\in F} w(f) = \sum_{f\in F_3} w(f) + \sum_{f\in F_4} w(f) + \sum_{\substack{i\ge 5\\ i \text{ is odd}}}\sum_{f\in F_i} w(f) + \sum_{\substack{i\ge 6\\ i \text{ is even}}}\sum_{f\in F_i} w(f)$$
$$\le \Big(\frac{5\cdot 3 - 7}{4} - \frac{1}{2}\Big)|F_3| + \Big(\frac{5\cdot 4 - 6}{4} - \frac{1}{2}\Big)|F_4| + \sum_{\substack{i\ge 5\\ i \text{ is odd}}}\Big(\frac{5i-7}{4}\Big)|F_i| + \sum_{\substack{i\ge 6\\ i \text{ is even}}}\Big(\frac{5i-6}{4}\Big)|F_i|$$
$$= \frac{1}{4}\sum_{\substack{i\ge 3\\ i \text{ is odd}}}\Big(5i-7\Big)|F_i| + \frac{1}{4}\sum_{\substack{i\ge 4\\ i \text{ is even}}}\Big(5i-6\Big)|F_i| - \frac{1}{2}|F_4| - \frac{1}{2}|F_3|$$
$$= \frac{1}{4}\sum_{i\ge 3}(5i-6)|F_i| - \frac{1}{4}\sum_{i \text{ is odd}}|F_i| - \frac{1}{2}|F_4| - \frac{1}{2}|F_3|. \quad ◀$$

Finally we can bound the number of vertices of B of degree 2 and 3.

▶ **Lemma 16.** $w(H) = |B_{=2}| + \dfrac{|B_{=3}|}{2} \leq 3|R|$.

Proof. Let F^{odd} be the set of faces of H with an odd number of edges. By Claim 15,

$$\begin{aligned} w(H) &\leq \frac{1}{4}\sum_{i\geq 3}\big(5i-6\big)|F_i| - \frac{1}{4}\sum_{i \text{ is odd}}|F_i| - \frac{1}{2}|F_4| - \frac{1}{2}|F_3| \\ &= \frac{5}{4}\sum_{i\geq 3} i|F_i| - \frac{3}{2}\sum_{i\geq 3}|F_i| - \frac{1}{4}\sum_{i \text{ is odd}}|F_i| - \frac{1}{2}|F_4| - \frac{1}{2}|F_3| \\ &= \frac{5}{2}|E| - \frac{3}{2}|F| - \frac{1}{4}|F^{\text{odd}}| - \frac{1}{2}|F_4| - \frac{1}{2}|F_3| = \hat{w}(H). \end{aligned} \tag{6}$$

Now note that the last quantity – $\hat{w}(H)$ as defined in Equation (6) – is maximized when H is a triangulation. To see this, consider an index i and a face $f \in F_i$ of H. Then decompose f into a face $f' \in F_{i-1}$ and a triangular face, resulting in a graph H'. Then comparing the bounds of Equation (6) for H and H':

- Case $i = 4$: $\hat{w}(H') \geq \hat{w}(H) + 1 - \frac{2}{4} + \frac{1}{2} - \frac{2}{2} = \hat{w}(H)$.
- Case $i \geq 5$ and i is odd: $\hat{w}(H') \geq \hat{w}(H) + 1 - \frac{1}{2} - \frac{1}{2} = \hat{w}(H)$.
- Case $i \geq 6$ and i is even: $\hat{w}(H') \geq \hat{w}(H) + 1 - \frac{2}{4} - \frac{1}{2} = \hat{w}(H)$.

Consider any triangulation H' of H. Then,

$$\begin{aligned} w(H) \leq \hat{w}(H) \leq \hat{w}(H') &= \frac{5}{2}|E_{H'}| - \frac{3}{2}|F_{H'}| - \frac{1}{4}|F_{H'}| - \frac{1}{2}|F_{H'}| = \frac{5}{2}|E_{H'}| - \frac{9}{4}|F_{H'}| \\ &= \frac{5}{2}|E_{H'}| - \frac{9}{4}\cdot\frac{2}{3}|E_{H'}| = \frac{5}{2}|E_{H'}| - \frac{3}{2}|E_{H'}| = |E_{H'}|. \end{aligned}$$

By using Euler's formula for planar graphs,

$$|R| - |E_{H'}| + \frac{2}{3}|E_{H'}| = 2 \implies |R| = 2 + \frac{1}{3}|E_{H'}|.$$

Therefore,

$$\frac{w(H)}{|R|} \leq \frac{|E_{H'}|}{2 + \frac{1}{3}|E_{H'}|} \leq 3,$$

implying that $w(H) \leq 3|R|$ and we're done. ◀

Now, Claim 14 and Lemma 16 imply the proof of the required Lemma 7.

2.2 Case: $H(R, E)$ is not 2-connected

Now we deal with the case when H is not 2-connected. The general idea will be to transform each such planar graph H to a 2-connected planar graph H' while respecting the 4L property as well as planarity. Consider a straight-line embedding of H in the plane. If H is not 2-connected, there exists a *cut edge* e, say $e = \{r_{i_1}, r\}$. Let $I = \{r_{i_1}, r_{i_2}, \ldots\}$ be the vertices in the connected component of r_{i_1} once e is removed. These vertices are called the *inner vertices*. Let $O = \{r, r_{o_1}, r_{o_2}, \ldots, r_{o_m}\}$ be the vertices in the connected component of r. These vertices are called the *outer vertices*. Further assume that $r_{o_1} \in O$ is the first vertex after r_{i_1}, in the clockwise order, that is adjacent to r (see Figure 6).

Our goal is to connect an inner vertex in I to an outer vertex in O iteratively until H becomes 2-connected. In order to achieve that, we will apply the following transformation:

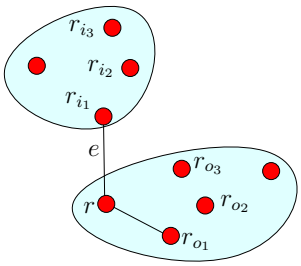


Figure 6 Inner vertices component and outer vertices component connected by cut edge e.

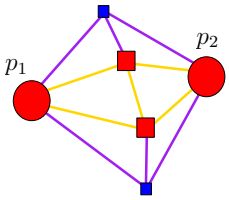


Figure 7 Gadget used for the clustering operation.

Clustering operation on $\{p_1, p_2\}$, where p_1 is an inner vertex and p_2 is an outer vertex: Add a set Q of two new red vertices to H. Furthermore, add sets B_2'' of 5 new degree-2 and B_3'' of 2 new degree-3 blue vertices. Connect these vertices as shown in Figure 7. Note that p_1 and p_2 are not adjacent in H.

We are going to argue that it is always possible to execute this while respecting planarity and 4L.

First we show that upper-bounding $w(\cdot)$ after a clustering operation gives an upper-bound for the original problem.

▶ **Claim 17.** *Let $H'(B', R', E')$ be the graph resulting from an application of the clustering operation on a graph $H(B, R, E)$. If $w(H') \leq 3|R'|$ then $w(H) \leq 3|R|$.*

Proof. More generally, assume we add b_2 new degree-two vertices to H', b_3 degree-three vertices and r red vertices. Then from assumption, we have

$$w(H') = |B_{=2}| + b_2 + \frac{|B_{=3}|}{2} + \frac{b_3}{2} \leq 3(|R| + r),$$

which implies that

$$w(H) = |B_{=2}| + \frac{|B_{=3}|}{2} \leq 3|R| + 3r - b_2 - \frac{b_3}{2} \leq 3|R|,$$

assuming $3r \leq b_2 + \frac{b_3}{2}$. This condition is satisfied for the clustering operation, where $r = 2$, $b_2 = 5$ and $b_3 = 2$. ◀

Next we show that a clustering operation does not violate the 4L condition.

▶ **Claim 18.** *The clustering operation preserves the* 4L *property.*

Proof. Let p_1 be any inner and p_2 be any outer vertex. Then add a set Q of two red vertices, a set B_2'' of 5 blue degree-2 vertices and a set B_3'' of 2 blue degree-3 vertices (see Figure 8). Let $B' \cup B''$ be any subset of size at most 4, where $B' \subseteq B$ and $B'' \subseteq B_2'' \cup B_3''$. We need to show that then $|N(B' \cup B'')| \geq |B' \cup B''|$.

1. $|B''| = 0$. Then $|N(B' \cup B'')| = |N(B')| \geq |B'|$, as H satisfies 4L.
2. $|B''| = 1$. As any vertex of B'' has at least one neighbor in Q, we have $|N(B' \cup B'')| \geq |N(B')| + 1 \geq |B'| + 1 = |B' \cup B''|$.
3. $|B''| = 2, 3$. As any two vertices of B'' have at least three neighbors in $Q \cup \{p_1, p_2\}$, and any vertex of B' must have at least one neighbor not in $Q \cup \{p_1, p_2\}$ (recall that p_1 and p_2 are not adjacent in H!), we get that $|N(B' \cup B'')| \geq |N(B'')| + 1 \geq 4$.
4. $|B''| = 4$. It can be verified that any set of 4 vertices of B'' have the set $Q \cup \{p_1, p_2\}$ of size 4 as its neighbor. ◀

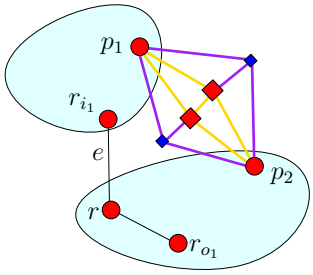


Figure 8 Clustering operation on p_1 and p_2.

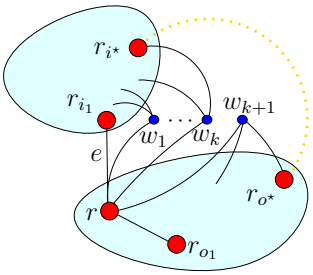


Figure 9 A planar path between $r_{i^\star}$ to $r_{o^\star}$.

Finally, we show that there exists an inner and an outer vertex which can be connected via a clustering operation while maintaining planarity.

▶ **Claim 19.** *It is always possible to find an inner vertex $r_{i^\star}$ and an outer vertex $r_{o^\star}$ such that there exists a path that connects them without violating planarity.*

Proof. Denote by $B^r_{=3}$ the set of degree-3 vertices adjacent to vertex r. If $B^r_{=3}$ is empty, then clearly there exists a path from the inner vertex r_{i_1} to the outer vertex r_{o_1}. Similarly, if there exists a vertex $w \in B^r_{=3}$ with one edge to an inner vertex and one to an outer vertex (other than the edge to r), then there exists a planar path between these inner and outer vertices by following the path along the edges of w.

Otherwise, sort the vertices in $B^r_{=3}$ clockwise by the order of their edges around r, say labeled $w_1, \ldots, w_t$. If w_1 has both edges (other than to r) to outer vertices, then clearly there is a planar path from r_{i_1} to one of these outer vertices (see Figure 9). Similarly, if w_t has both edges (other than to r) to inner vertices, then there is a planar path from r_{o_1} to one of these inner vertices. Now by a parity argument, there must exist two vertices, say w_k and w_{k+1}, such that w_k has both neighbors to inner vertices, and w_{k+1} has both neighbors to outer vertices. Then there exists a path from one of inner vertices adjacent to w_k to one of the outer vertices adjacent to w_{k+1}. ◀

▶ **Lemma 20.** *Let H be a 1-connected planar graph. Then $w(H) \leq 3|R|$.*

Proof. Claim 19 implies that – as long as the current graph H is not 2-connected – it is always possible to do a clustering operation between an inner vertex and an outer vertex while maintaining planarity. By Claim 18, the resulting graph H' still satisfies the condition 4L. Crucially, note that each new edge introduced by the clustering operation is not a cut edge in the derived graph H', and further, the edge e which was a cut edge in H is no longer a cut edge in H'. Thus the clustering operation reduces the total number of cut edges, and so the process terminates after a finite number of steps. Apply this iteratively to get a 2-connected graph H', which, by Lemma 16, satisfies $w(H') \leq 3|R'|$. Then $w(H) \leq 3|R|$ follows by Claim 17. ◀

References

1 Pankaj K. Agarwal and Nabil H. Mustafa. Independent set of intersection graphs of convex objects in 2D. *Comput. Geom.*, 34(2):83–95, 2006.

2 Rom Aschner, Matthew J. Katz, Gila Morgenstern, and Yelena Yuditsky. Approximation schemes for covering and packing. In *Proceedings of the 7th International Workshop on Algorithms and Computation (WALCOM)*, pages 89–100, 2013.

3 Norbert Bus, Shashwat Garg, Nabil H. Mustafa, and Saurabh Ray. Limits of local search: Quality and efficiency. *Discrete & Computational Geometry*, 57(3):607–624, 2017.

4 Norbert Bus, Nabil H. Mustafa, and Saurabh Ray. Geometric hitting sets for disks: Theory and practice. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA)*, pages 903–914, 2015.

5 Sergio Cabello and David Gajser. Simple PTAS's for families of graphs excluding a minor. *Discrete Applied Mathematics*, 189:41–48, 2015.

6 Timothy M. Chan. Polynomial-time approximation schemes for packing and piercing fat objects. *J. Algorithms*, 46(2):178–189, 2003.

7 Timothy M. Chan and Elyot Grant. Exact algorithms and APX-hardness results for geometric packing and covering problems. *Comput. Geom.*, 47(2):112–124, 2014.

8 Timothy M. Chan and Sariel Har-Peled. Approximation algorithms for maximum independent set of pseudo-disks. *Discrete & Computational Geometry*, 48(2):373–392, 2012.

9 Robert Fraser. *Algorithms for Geometric Covering and Piercing Problems*. PhD thesis, University of Waterloo, 2012.

10 Matt Gibson, Gaurav Kanade, Erik Krohn, and Kasturi R. Varadarajan. Guarding terrains via local search. *JoCG*, 5(1):168–178, 2014.

11 Matt Gibson and Imran A. Pirwani. Algorithms for dominating set in disk graphs: Breaking the log$n$ barrier. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA)*, pages 243–254, 2010.

12 Sathish Govindarajan, Rajiv Raman, Saurabh Ray, and Aniket Basu Roy. Packing and covering with non-piercing regions. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA)*, pages 47:1–47:17, 2016.

13 Dániel Marx. Efficient approximation schemes for geometric problems? In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA)*, pages 448–459, 2005.

14 Dániel Marx. Parameterized complexity of independence and domination on geometric graphs. In *Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC)*, pages 154–165, 2006.

15 Nabil H. Mustafa, Rajiv Raman, and Saurabh Ray. Quasi-polynomial time approximation scheme for weighted geometric set cover on pseudodisks and halfspaces. *SIAM J. Comput.*, 44(6):1650–1669, 2015.

16 Nabil H. Mustafa and Saurabh Ray. Improved results on geometric hitting set problems. *Discrete & Computational Geometry*, 44(4):883–895, 2010.

17 Nabil H. Mustafa and K. Varadarajan. Epsilon-approximations and Epsilon-nets. In J. E. Goodman, J. O'Rourke, and C. D. Tóth, editors, *Handbook of Discrete and Computational Geometry*. CRC Press LLC, 2017.

18 Radoš Radoičić and Géza Tóth. The discharging method in combinatorial geometry and the Pach-Sharir conjecture. In *Contemporary Mathematics: Surveys on Discrete and Computational Geometry*, pages 319–342. American Mathematical Society, 2008.

In-Place Parallel Super Scalar Samplesort (IPS^{4}o)*

Michael Axtmann[1], Sascha Witt[2], Daniel Ferizovic[3], and Peter Sanders[4]

1 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
michael.axtmann@kit.edu
2 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
sascha.witt@kit.edu
3 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
4 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
sanders@kit.edu

Abstract

We present a sorting algorithm that works in-place, executes in parallel, is cache-efficient, avoids branch-mispredictions, and performs work $\mathcal{O}(n \log n)$ for arbitrary inputs with high probability. The main algorithmic contributions are new ways to make distribution-based algorithms in-place: On the practical side, by using coarse-grained block-based permutations, and on the theoretical side, we show how to eliminate the recursion stack. Extensive experiments show that our algorithm IPS^{4}o scales well on a variety of multi-core machines. We outperform our closest in-place competitor by a factor of up to 3. Even as a sequential algorithm, we are up to 1.5 times faster than the closest sequential competitor, BlockQuicksort.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases shared memory, parallel sorting, in-place algorithm, comparison-based sorting, branch prediction

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.9

1 Introduction

Sorting an array $A[1..n]$ of n elements according to a total ordering of their keys is a fundamental subroutine used in many applications. Sorting is used for index construction, for bringing similar elements together, or for processing data in a "clever" order. Indeed, often sorting is the most expensive part of a program. Consequently, a huge amount of research on sorting has been done. In particular, algorithm engineering has studied how to make sorting practically fast in presence of complex features of modern hardware like multi-core (e.g., [30, 29, 5, 28]), instruction parallelism (e.g., [27]), branch prediction (e.g., [27, 19, 18, 10]), caches (e.g., [27, 7, 11, 5]), or virtual memory (e.g., [24, 17]). In contrast, the sorting algorithms used in the standard libraries of programming languages like Java or C++ still use variants of quicksort – an algorithm that is more than 50 years old. A reason seems to be that you have to outperform quicksort in every respect in order to replace it. This is less easy than it sounds since quicksort is a pretty good algorithm – it needs $\mathcal{O}(n \log n)$ expected work, it can be parallelized [30, 29], it can be implemented to avoid branch mispredictions [10], and it is reasonably cache-efficient. Perhaps most importantly,

* A full version of the paper is available at https://arxiv.org/abs/1705.02257.

quicksort works (almost) in-place[1] which is of crucial importance for very large inputs. This feature rules out many contenders. Further algorithms are eliminated by the requirement to work for arbitrary data types and input distributions. This makes integer sorting algorithms like radix sort (e.g., [21]) or using specialized hardware (e.g., GPUs or SIMD instructions) less attractive, since these algorithms cannot be used in a reusable library where they have to work for arbitrary data types. Another portability issue is that the algorithm should use no code specific to the processor architecture or the operating system like non-temporal writes or overallocation of virtual memory (e.g. [26]). One aspect of making an algorithm in-place is that such "tricks" are not needed. Hence, this paper focuses on portable comparison-based algorithms and also considers how the algorithms can be made robust for arbitrary inputs, e.g., with a large number of repeated keys.

The main contribution of this paper is to propose a new algorithm – ***I****n-place* ***P****arallel* ***S****uper* ***S****calar* ***S****ample****so****rt* (IPS^{4}o)[2] – that combines enough advantages to become an attractive replacement of quicksort. Our starting point is *super scalar samplesort* (s^3-sort) [27] which already provides a very good sequential non-in-place algorithm that is cache-efficient, allows considerable instruction parallelism, and avoids branch mispredictions. s^3-sort is a variant of samplesort, which in turn is a generalization of quicksort to multiple pivots. The main operation is distributing elements of an input sequence to k output buckets of about equal size. We parallelize this algorithm using t threads and make it more robust by taking advantage of inputs with many identical keys. Our main innovation is to make the algorithm in-place. The first phase of IPS^{4}o distributes the elements to k *buffer* blocks. When a buffer becomes full, it is emptied into a block of the input array that has already been distributed. Subsequently, the memory blocks are permuted into the globally correct order. A cleanup step handles empty blocks and half-filled buffer blocks. The distribution phase is parallelized by assigning disjoint pieces of the input array to different threads. The block permutation phase is parallelized using atomic fetch-and-add operations for each block move. Once subproblems are small enough, they can be solved independently in parallel.

After discussing related work in Section 2 and introducing basic tools in Section 3, we describe our new algorithm IPS^{4}o in Section 4. Section 5 makes an experimental evaluation. An overall discussion and possible future work is given in Section 6. The full paper [3] gives further experimental data and proofs.

2 Related Work

Variants of Hoare's quicksort [15, 23] are generally considered some of the most efficient general purpose sorting algorithms. Quicksort works by selecting a *pivot* element and partitioning the array such that all elements smaller than the pivot are in the left part and all elements larger than the pivot are in the right part. The subproblems are solved recursively. A variant of quicksort (with a fallback to heapsort to avoid worst case scenarios) is currently used in the C++ standard library of GCC [23]. Some variants of quicksort use two or three pivots [31, 22] and achieve improvements of around 20% in running time over the single-pivot case. Dual-pivot quicksort [31] is the default sorting routine in Oracle Java 7 and 8. The basic principle of quicksort remains, but elements are partitioned into three or four subproblems

[1] In algorithm theory, an algorithm works in-place if it uses only constant space in addition to its input. We use the term *strictly in-place* for this case. In algorithm engineering, one is sometimes satisfied if the additional space is sublinear in the input size. We adopt this convention but use the term *almost in-place* when we want to make clear what we mean. Quicksort needs logarithmic additional space.

[2] The Latin word "ipso" means "by itself", referring to the in-place feature of IPS^{4}o.

instead of two. Increasing the number of subproblems (from now on called *buckets*) even further leads to samplesort [6, 5]. Unlike single- and dual-pivot quicksort, samplesort is usually not in-place, but it is well-suited for parallelization and more cache-efficient.

Super scalar samplesort [27] (s^3-sort) improves on samplesort by avoiding inherently hard-to-predict conditional branches linked to element comparisons. Branch mispredictions are very expensive because they disrupt the pipelined and instruction-parallel operation of modern processors. Traditional quicksort variants suffer massively from branch mispredictions [19]. By replacing conditional branches with conditionally executed machine instructions, branch mispredictions can be largely avoided. This is done automatically by modern compilers if only a few instructions depend on a condition. As a result, s^3-sort is up to two times faster than quicksort (`std::sort`), at the cost of $\mathcal{O}(n)$ additional space. BlockQuicksort [10] applies similar ideas to single-pivot quicksort, resulting in a very fast in-place sorting algorithm.

Super scalar samplesort has also been adapted for efficient parallel string sorting [4]. Our implementation is influenced by that work with respect to parallelization and handling equal keys. Moreover, we were also influenced by an implementation of s^3-sort written by Lorenz Hübschle-Schneider. A prototypical implementation of sequential non-blocked in-place s^3-sort in a student project by our student Florian Weber motivated us to develop IPS^{4}o.

The best practical comparison-based multi-core sorting algorithms we have found are based on multi-way mergesort [29] and samplesort [28], respectively. The former algorithm is used in the parallel mode of the C++ standard library of GCC. Parallel in-place algorithms are based on quicksort so far. Intel's Thread Building Blocks library [25] contains a variant that uses only sequential partitioning. The MCSTL library [29] contains two implementations of the more scalable parallel quicksort by Tsigas and Zhang [30].

There is a considerable amount of work by the theory community on (strictly) in-place sorting (e.g., [11, 12]). However, there are few – mostly negative – results on transferring these results into practice. Katajainen and Teuhola [20] report that in-place mergesort is slower than heapsort, which is quite slow for big inputs due to its cache-inefficiency. Chen [8] reports that in-place merging takes about six times longer than non-in-place merging. There is previous work on (almost) in-place multi-way merging or data distribution. However, few of these papers seem to address parallelism. There are also other problems. For example, the multi-way merger in [14] needs to allocate very large blocks to become efficient. In contrast, the block size of IPS^{4}o does not depend on the input size. In-place data distribution, e.g., for radix sort [9], is often done element by element. Using this for samplesort would require doing the expensive element classification twice and would also make parallelization difficult.

3 Preliminaries

(Super Scalar) Samplesort. Samplesort [13] can be viewed as a generalization of quicksort which uses multiple pivots to split the input into k *buckets* of about equal size. A robust way for determining the pivots is to sort $\alpha k - 1$ randomly sampled input elements. The pivots $s_1, \ldots s_{k-1}$ are then picked equidistantly from the sorted sample. Element e goes to bucket b_i if $s_{i-1} \leq e < s_i$ (with $s_0 = -\infty$ and $s_k = \infty$). The main contribution of s^3-sort [27] is to eliminate branch mispredictions for element classification. Assuming k is a power of two, the pivots are stored in an array a representing a complete binary search tree: $a_1 = s_{k/2}$, $a_2 = s_{k/4}$, $a_3 = s_{3k/4}, \ldots$ More generally, the left successor of a_i is a_{2i} and its right successor is a_{2i+1}. Thus, navigating this tree is possible by performing a conditional instruction for incrementing an array index. We adopt (and refine) this approach to element classification but change the organization of buckets in order to make the algorithm in-place.

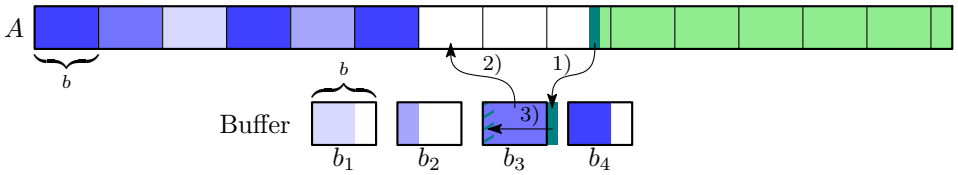


Figure 1 Local classification. Blue elements have already been classified, with different shades indicating different buckets. Unprocessed elements are green. Here, the next element (in dark green) has been determined to belong to bucket b_3. As that buffer block is already full, we first write it into the array A, then write the new element into the now empty buffer.

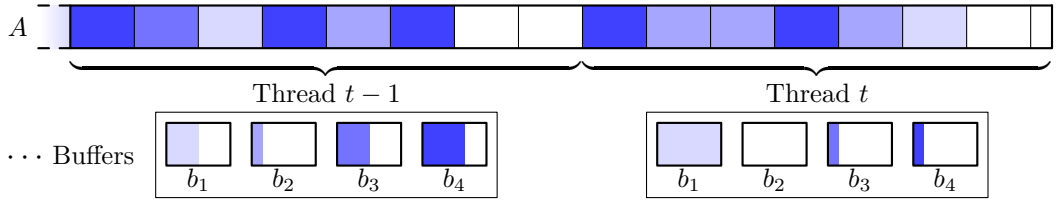


Figure 2 Input array and block buffers of the last two threads after local classification.

4 In-Place Parallel Super Scalar Samplesort (IPS^{4}o)

IPS^{4}o is based on the ideas of s^3-sort. It is a recursive algorithm, where each step divides the input into k buckets, such that each element of bucket b_i is smaller than all elements of b_{i+1}. As long as problems with at least $\beta\frac{n}{t}$ elements exist, we partition those problems one after another with t threads in parallel. Here, β is a tuning parameter. Then we assign remaining problems in a balanced way to threads, which sort them sequentially.

The partitioning consists of four phases. **Sampling** determines the bucket boundaries. **Local classification** groups the input into blocks such that all elements in each block belong to the same bucket. **Block permutation** brings the blocks into the globally correct order. Finally, we perform some **cleanup** around the bucket boundaries. The following sections will explain each of these phases in more detail.

Sampling. The sampling phase is similar to the sampling in s^3-sort. The main difference is that we swap the sample to the front of the input array to keep the in-place property even if the oversampling factor α depends on n.

4.1 Local Classification

The input array A is viewed as an array of blocks each containing b elements (except possibly for the last one). For parallel processing, we divide the blocks of A into t stripes of equal size – one for each thread. Each thread works with a local array of k buffer blocks – one for each bucket. A thread then scans its stripe. Using the search tree created in the previous phase, each element in the stripe is classified into one of the k buckets, then moved into the corresponding local buffer block. If this buffer is already full, it is first written back into the local stripe, starting at the front. It is clear that there is enough space to write b elements into the local stripe, since at least b more elements have been scanned from the stripe than have been written back – otherwise no full buffer could exist.

In this way, each thread creates blocks of b elements belonging to the same bucket. Figure 1 shows a typical situation during this phase. To achieve the in-place property, we

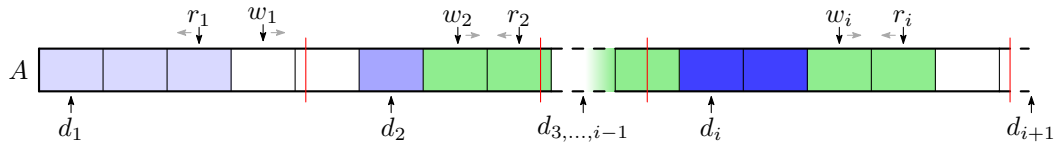


Figure 3 Invariant during block permutation. In each bucket b_i, blocks in $[d_i, w_i)$ are already correct (blue), blocks in $[w_i, r_i]$ are unprocessed (green), and blocks in $[\max(w_i, r_i + 1), d_{i+1})$ are empty (white).

do not track which bucket each block belongs to. However, we do keep count of how many elements are classified into each bucket, since we need this information in the following phases. This information can be obtained almost for free as a side effect of maintaining the buffer blocks. Figure 2 depicts the input array after local classification. Each stripe contains a number of full blocks, followed by a number of empty blocks. The remaining elements are still contained in the buffer blocks.

4.2 Block Permutation

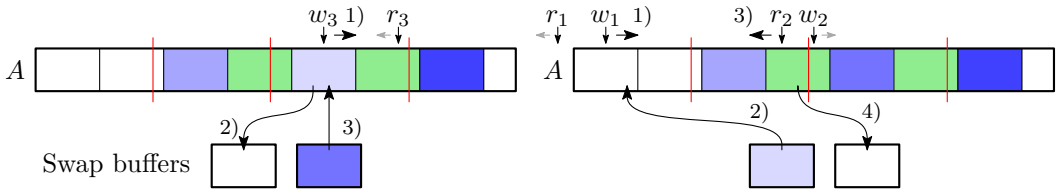
In this phase, the blocks in the input array will be rearranged such that they appear in the correct order. From the previous phase we know, for each stripe, how many elements belong to each bucket. We perform a prefix sum operation to compute the exact boundaries of the buckets in the input array. In general, these will not coincide with the block boundaries. For the purposes of this phase, we will ignore this: We mark the beginning of each bucket b_i with a delimiter pointer d_i, rounded up to the next block. We similarly mark the end of the last bucket b_k with a delimiter pointer d_{k+1}. Adjusting the boundaries may cause a bucket to "lose" up to $b - 1$ elements; this doesn't affect us, since this phase only deals with full blocks, and any elements not constituting a full block remain in the buffers. Additionally, if the input size is not a multiple of b, some of the d_is may end up outside the bounds of A. To avoid overflows, we allocate a single empty *overflow block* which the algorithm will use instead of writing to the final (partial) block.

For each b_i, a write pointer w_i and a read pointer r_i is introduced; these will be set such that all unprocessed blocks, i.e., blocks that still need to be moved into the correct bucket, are found between w_i and r_i. During the block permutation, we maintain the following invariant for each bucket b_i, visualized in Figure 3:

- Blocks to the left of w_i (exclusive) are correctly placed, i.e., contain only elements belonging to b_i.
- Blocks between w_i and r_i (inclusive) are unprocessed, i.e., may need to be moved.
- Blocks to the right of $\max(w_i, r_i + 1)$ (inclusive) are empty.

In other words, each bucket follows the pattern of correct blocks followed by unprocessed blocks followed by empty blocks, with w_i and r_i determining the boundaries. In the parallel case, we may need to establish this invariant by moving some empty blocks to the end of a bucket (see the full paper [3] for details); in the sequential algorithm, the result of the classification phase already has this pattern. The read pointers r_i are then set to the first non-empty block in each bucket, or $d_i - 1$ if there are none.

We are now ready to start the block permutation. Each thread maintains two local swap buffers. We define a *primary* bucket b_p for each thread; whenever both its buffers are empty, a thread tries to read an unprocessed block from its primary bucket. To do so, it decrements the read pointer r_p (atomically) and reads the block it pointed to into one of its swap buffers.



(a) Swapping a block into its correct position.

(b) Moving a block into an empty position, followed by refilling the swap buffer.

Figure 4 Block permutation examples.

If b_p contains no more unprocessed blocks (i.e., $r_p < w_p$), it switches its primary bucket to the next bucket (cyclically). If it completes a whole cycle and arrives back at its initial primary bucket, there are no more unprocessed blocks and this phase ends. The starting points for the threads are distributed across that cycle to reduce contention.

Once it has a block, each thread classifies the first element of that block to find its destination bucket b_{dest}. There are now two possible cases, visualized in Figure 4:

- As long as $w_{\text{dest}} \leq r_{\text{dest}}$, write pointer w_{dest} still points to an unprocessed block in bucket b_{dest}. In this case, the thread increases w_{dest}, reads the unprocessed block into its empty swap buffer, and writes the other one into its place.
- If $w_{\text{dest}} > r_{\text{dest}}$, no unprocessed block remains in bucket b_{dest} but w_{dest} now points to an empty block. In this case, the thread increases w_{dest}, writes its swap buffer to the empty block and then reads a new unprocessed block from its primary bucket.

We repeat these steps until all blocks are processed. We can skip unprocessed blocks which are already correctly placed: We simply classify blocks *before* reading them into a swap buffer, and skip as needed. We omitted this from the above description for the sake of clarity. In some cases, this reduces the number of block moves significantly.

It is possible that one thread wants to write to a block that another thread is currently reading from (when the reading thread has just decremented the read pointer, but has not yet finished reading the block into its swap buffer). To avoid data races, we keep track of how many threads are reading from each bucket. Threads are only allowed to write to empty blocks if no other threads are currently reading from the bucket in question, otherwise they wait. Note that this situation occurs at most once for each bucket, namely when w_{dest} and r_{dest} cross each other. In addition, we store each w_i and r_i in a single 128-bit word which we read and modify atomically. This ensures a consistent view of both pointers for all threads.

4.3 Cleanup

After the block permutation, some elements may still be in incorrect positions. This is due to the fact that we only moved blocks, which may span bucket boundaries. We call the partial block at the beginning of a bucket its *head* and the partial block at its end its *tail*.

We assign consecutive buckets evenly to threads; if $t > k$, some threads will not receive any buckets, but those that do only need to process a single bucket each. Each thread reads the head of the first bucket of the next thread into one of its swap buffers. Then, each thread processes its buckets from left to right, moving incorrectly placed elements into empty array entries. The incorrectly placed elements of bucket b_i consist of the elements in the head of b_{i+1} (or the swap buffer, for the last bucket), the partially filled buffers from the local classification phase (of all threads), and, for the corresponding bucket, the overflow

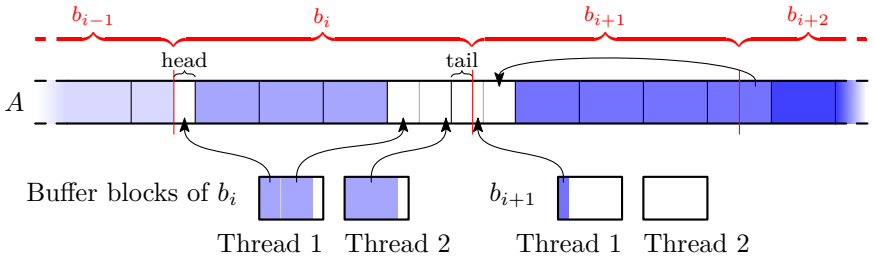


Figure 5 An example of the steps performed during cleanup.

buffer. Empty array entries consist of the head of b_i and any (empty) blocks to the right of w_i (inclusive). Although the concept is relatively straightforward, the implementation is somewhat involved, due to the many parts that have to be brought together. Figure 5 shows an example of the steps performed during this phase. Afterwards, all elements are back in the input array and correctly partitioned, ready for recursion.

4.4 The Case of Many Identical Keys

Having inputs with many identical keys can be a problem for samplesort, since this might move large fractions of the keys through many levels of recursion. We turn such inputs into *easy* instances by introducing separate buckets for elements identical to pivots (keys occurring more then $\frac{n}{k}$ times are likely to become pivots). Finding out whether an element has to go into an equality bucket (and which one) can be implemented using a single additional comparison [4] and, once more, without a conditional branch. Equality buckets can be skipped during recursion and thus are not a load balancing problem.

4.5 Analysis

Algorithm IPS^{4}o inherits from s^3-sort that it has virtually no branch mispredictions (this includes the comparisons for placing elements into equality buckets discussed in subsection 4.4). More interesting is the parallel complexity. Here, the main issue is the number of accesses to main memory. We analyze this aspect in the parallel external memory (PEM) model [1], where each of the t threads has a private cache of size M and access to main memory happens in blocks of size B. In the full paper [3], we prove:

▶ **Theorem 1.** *Assuming $b = \Theta(tB)$ (buffer block size), $M = \Omega(ktB)$, $n_0 = \mathcal{O}(M)$ (base case size), $\alpha \in \Omega(\log t) \cap \mathcal{O}(t)$ (oversampling factor), and $n = \Omega\big(\max(k,t)t^2B\big)$, IPS^{4}o has an I/O-complexity of $\mathcal{O}\Big(\frac{n}{tB}\log_k \frac{n}{n_0}\Big)$ block transfers with high probability.*

Basically, Theorem 1 tells us that IPS^{4}o is asymptotically I/O efficient if certain rather steep assumptions on cache size and input size hold. In particular, the blocks need to have size $b = \Theta(tB)$ in order to amortize contention on shared block pointers. Lifting those could be an interesting theoretical question and we would have to see how absence of branch mispredictions and the in-place property can be combined with previous techniques [1, 5]. However, it is likely that the constant factors involved are much larger than for our simple implementation. Thus, the constant factors will be the main issue in bringing theory and practice further together. To throw some light on this aspect, let us compare the constant factors in I/O-volume (i.e., data flow between cache and main memory) for the sequential algorithms IS^{4}o (IPS^{4}o with $t = 1$) and s^3-sort. To simplify the discussion, we assume a single

level of recursion, $k = 256$ and 8-byte elements. In the full paper [3], we show that IS^{4}o needs about $48n$ bytes of I/O volume, whereas s^3-sort needs (more than) $86n$ – almost twice that of IS^{4}o. This is surprising since on first glance, the partitioning algorithm of IS^{4}o writes the data twice, whereas s^3-sort does this only once. However, this is more than offset by "hidden" overheads of s^3-sort like memory management, allocation misses, and associativity misses.

Finally, we consider the memory overhead of IPS^{4}o. In the full paper [3], we show:

▶ **Theorem 2.** *IPS^{4}o requires additional space* $\mathcal{O}\left(kbt + \log_k \frac{n}{n_0}\right)$.

In practice, the term $\mathcal{O}(kbt)$ (mostly for the distribution buffers) will dominate. However, for a strictly in-place algorithm in the sense of algorithm theory, we need to get rid of the $\mathcal{O}(\log n)$ term which depends on the input size. We discuss this separately in subsection 4.6.

4.6 From Almost In-Place to Strictly In-Place

We now explain how the space consumption of IPS^{4}o can be made independent of n in a rather simple way. We can restrict ourselves to the sequential case, since only $\mathcal{O}(\log_k t)$ levels of parallel recursion are needed to arrive at subproblems that are solved sequentially. We require the partitioning operation to mark the beginning of each bucket by storing the largest element of a bucket in its first entry. By searching the next larger element, we can then find the end of the bucket. Note that this is possible in time logarithmic in the bucket size using exponential/binary search. We assume that the corresponding function *searchNextLargest* returns $n + 1$ if no larger elements exists – this happens for the last bucket. The following pseudocode uses this approach to emulate recursion in constant space for sequential IS^{4}o.

```
i := 1                                                    -- first element of current bucket
j := n + 1                                                -- first element of next bucket
while i < n do
    if j − i < n_0 then smallSort(a, i, j − 1);  i := j   -- base case
    else partition(a, i, j − 1)                           -- partition first unsorted bucket
    j := searchNextLargest(A[i], A, i + 1, n)             -- find beginning of next bucket
```

4.7 Implementation Details

The strategy for handling identical keys described in subsection 4.4 is enabled conditionally: After the splitters have been selected from the initial sample, we check for and remove duplicates. Equality buckets are only used if there were duplicate splitters.

For buckets under a certain base case size n_0, we stop the recursion and fall back on insertion sort. Additionally, we use an adaptive number of buckets on the last two levels of the recursion, such that the expected size of the final buckets remains reasonable. For example, instead of performing two 256-way partitioning steps to get 2^{16} buckets of 2 elements, we might perform two 64-way partitioning steps to get 2^{12} buckets of about 32 elements. Furthermore, on the last level, we perform the base case sorting immediately after the bucket has been completely filled in the cleanup phase, before processing the other buckets. This is more cache-friendly, as it eliminates the need for another pass over the data.

IPS^{4}o has several parameters that can be used for tuning and adaptation. We performed our experiments using (up to) $k = 256$ buckets, an oversampling factor of $\alpha = 0.2 \log n$, an overpartitioning factor of $\beta = 1$, a base case size of $n_0 = 16$ elements, and a block size of about 2 KiB, or $b = \max\left(1, 2^{\lfloor 11 - \log_2 s \rfloor}\right)$ elements, where s is the size of an element in bytes. In the sequential case, we avoid the use of atomic operations on pointers. All algorithms are written in C++ and compiled with version 6.2.0 of the GNU compiler collection, using

the optimization flags "`-march=native -O3`". For parallelization, we employ OpenMP. Our implementation can be found at `https://github.com/SaschaWitt/ips4o`.

5 Experimental Results

We present the results of our in-place parallel sorting algorithm IPS^{4}o. We compare the results of IPS^{4}o with its in-place competitors, parallel sort from the Intel® TBB library [25] (TBB), parallel unbalanced quicksort from the GCC STL library (MCSTLubq), and parallel balanced quicksort from the GCC STL library (MCSTLbq). We also give results on the parallel non-in-place sorting algorithms, parallel samplesort from the problem based benchmark suite [28] (PBBS) and parallel multiway mergesort from the GCC STL library [29] (MCSTLmwm). We also ran sequential experiments and present the results of IS^{4}o, the sequential implementation of IPS^{4}o. We compare the results of IS^{4}o with its sequential competitors, a recent implementation [16] of non-in-place Super Scalar Samplesort [27] (s^3-sort) optimized for modern hardware, BlockQuicksort [10] (BlockQ), Dual-Pivot Quicksort [31] (DualPivot), and introsort from the GCC STL library (std-sort).

We ran benchmarks with nine input distributions: Uniformly distributed (*Uniform*), exponentially distributed (*Exponential*), and almost sorted (*AlmostSorted*), proposed by Shun et. al. [28]; *RootDup*, *TwoDup*, and *EightDup* from Edelkamp et. al. [10]; and *Sorted* (sorted Uniform input), *ReverseSorted*, and *Ones* (just ones). The input distribution RootDup sets $A[i] = i \mod \lfloor\sqrt{n}\rfloor$, TwoDup sets $A[i] = i^2 + \frac{n}{2} \mod n$, and EightDup sets $A[i] = i^8 + \frac{n}{2} \mod n$. We ran benchmarks with 64-bit floating point elements and *Pair*, *Quartet*, and *100Bytes* data types. Pair (Quartet) consists of one (three) 64-bit floating point elements as key and one 64-bit floating point element of associated information. 100Bytes consists of 10 bytes as key and 90 bytes of associated information. Quartet and 100Bytes are compared lexicographically. For $n < 2^{30}$, we perform each measurement 15 times and for $n \geq 2^{30}$, we perform each measurement twice. Unless stated otherwise, we report the average over all runs and use 64-bit floating point elements.

We ran our experiments on machines with one AMD Ryzen +1800 8-core processor (*AMD1S*), two Intel Xeon E5-2683 v4 16-core processors (*Intel2S*), and four Intel Xeon E5-4640 8-core processors (*Intel4S*). Intel2S and Intel4S are equipped with 512 GiB of memory, AMD1S is equipped with 32 GiB of memory. We use the `taskset` tool to set the CPU affinity for speedup benchmarks. We tested all parallel algorithms on Uniform input with and without hyper-threading. Hyper-threading did not slow down any algorithm. Thus, we give results of all algorithms with hyper-threading. Overall, we executed more than 12 000 combinations of different algorithms, input distributions and sizes, data types and machines. We now present a selection of our measurements and discuss our results. For the remaining (detailed) running time and hardware counter measurements, we refer to the full paper [3].

Sequential Algorithms. Figure 6 shows the running times of sequential algorithms on Uniform input executed on machine Intel2S. We see that IS^{4}o is faster than its closest competitor, BlockQ, by a factor of 1.14 for $n = 2^{32}$. On machine Intel4S (AMD1S), IS^{4}o outperforms BlockQ even by a factor of 1.22 (1.57). DualPivot and std-sort, which do not avoid branch mispredictions, are at least a factor of 1.86 slower than IS^{4}o for $n = 2^{32}$. The number of branch mispredictions of these algorithms for this input size is about 10 times larger than that of IS^{4}o. s^3-sort is the slowest sequential sorting algorithm avoiding branch mispredictions and has fluctuations in running time for varying input sizes. Due to the initial overhead, IS^{4}o is slower than BlockQ for $n \leq 2^{15}$.

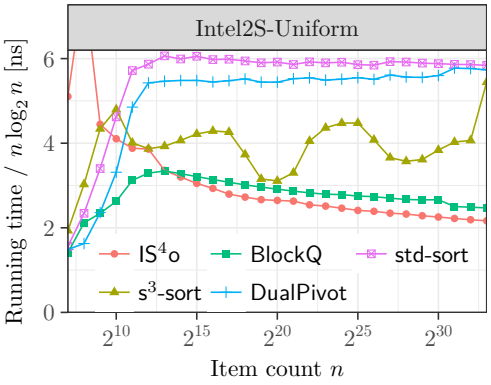


Figure 6 Running times of sequential algorithms on input distribution Uniform executed on machine Intel2S.

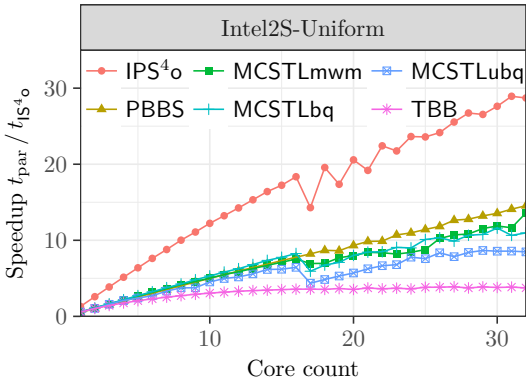


Figure 7 Speedup of parallel algorithms with different number of cores relative to our sequential implementation IS^{4}o on Intel2S, sorting 2^{30} elements of input distribution Uniform.

As expected, the running times for inputs with a moderate number of different keys (TwoDup) are similar to the running times for Uniform. When the number of different keys decreases (Exponential, EightDup, and RootDup in decreasing order), IS^{4}o becomes even faster by a factor of up to two on all machines. The running times of the competitors also decrease. However, only DualPivot on Intel2S with RootDup distributed input comes close for $n \geq 2^{28}$. Only input Ones and (almost) sorted input are hard for IS^{4}o; for example, DualPivot outperforms IS^{4}o on AlmostSorted input by a factor of 1.70 for $n = 2^{32}$ (Intel2S).

Parallel Algorithms. Figure 8 (a–c) presents experiments of parallel algorithms on different machines for Uniform input. We see that IPS^{4}o outperforms its closest competitors, e.g., for $n = 2^{32}$ on Intel2S (AMD1S) by a factor of 2.13 (1.75), and all but TBB and IPS^{4}o fail to sort this input size on AMD1S due to memory limitations. For $n \geq 2^{26}$, IPS^{4}o outperforms its closest non-in-place competitors on Intel2S (AMD1S) on average by a factor of 2.26 (1.69) and its closest in-place competitors by a factor of 2.78 (1.98). For the same input sizes, IPS^{4}o outperforms its closest competitors on Intel4S in average just by a factor of 1.41. We believe that the small difference in running time between IPS^{4}o and its competitors on Intel4S is caused by two factors: The slower memory modules (DDR4 vs. DDR3), and the long load delays due to a ring interconnect between four sockets.

In Figure 8 (d–e), we present running times of parallel algorithms on input distributions with duplicates (TwoDup and RootDup) on machine Intel2S. For $n \geq 2^{26}$ and a moderate number of different keys (TwoDup), IPS^{4}o still outperforms its in-place competitors on average by a factor of at least 2.88 and its non-in-place competitors on average by a factor of at least 1.91. Experiments have shown that the running times on EightDup and Exponential are similar to the running times on TwoDup. We also see that the non-in-place algorithms become almost as fast as IPS^{4}o if we sort inputs which contain few different keys (RootDup). However, IPS^{4}o still outperforms its in-place competitors by a factor of at least 3.43 on this input for $n \geq 2^{20}$. Figure 8 (f) depicts the running times of parallel algorithms on AlmostSorted distributions on Intel2S. On AlmostSorted and ReverseSorted, the fastest non-in-place algorithm, PBBS, performs similarly to IPS^{4}o for large input sizes. Only on Sorted and Ones, IPS^{4}o is outperformed by TBB, an in-place competitor. This is because TBB detects these pre-sorted input distributions and terminates immediately. Further benchmarks on machines

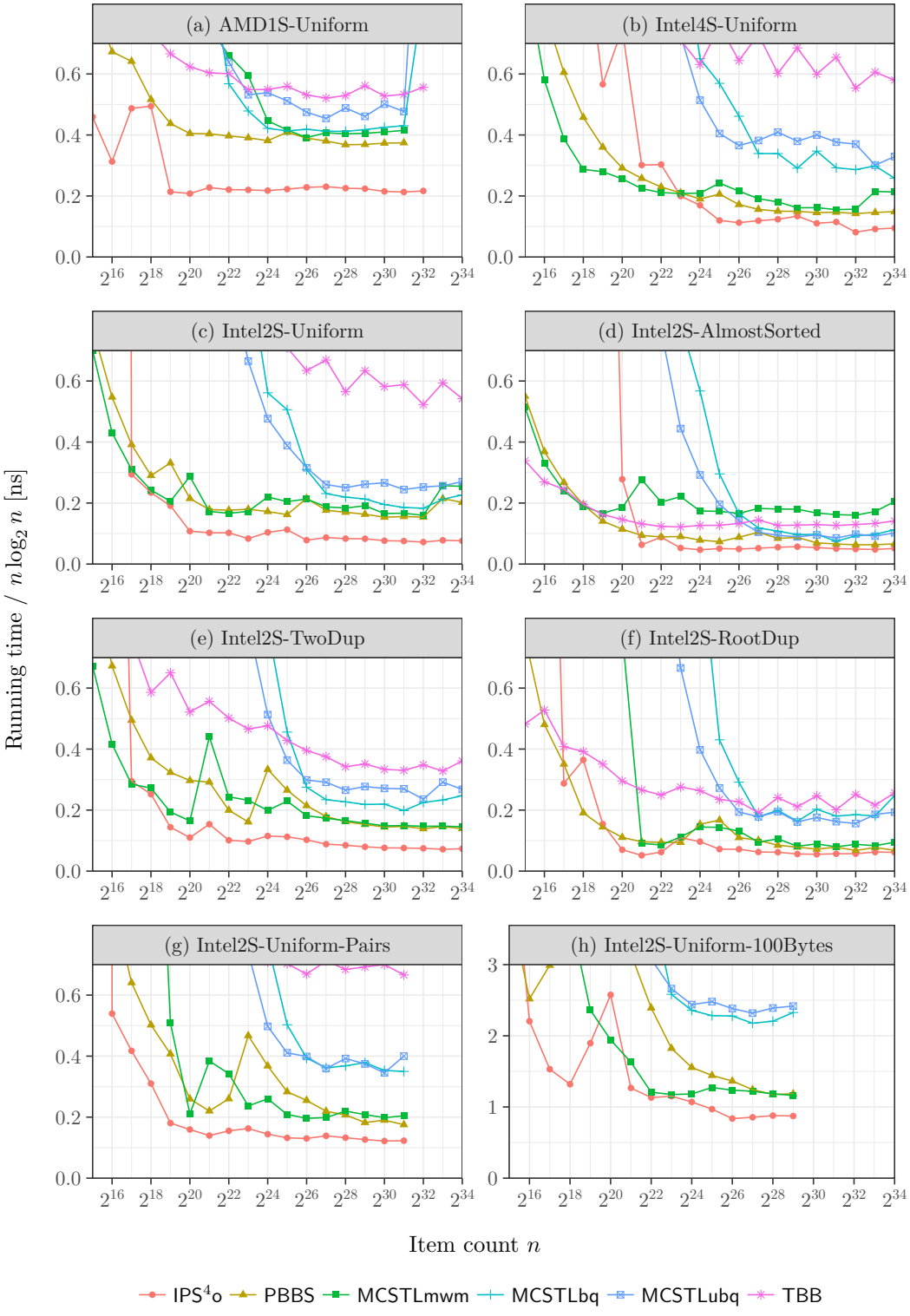


Figure 8 Running times of parallel algorithms on different input distributions executed on different machines.

Intel4S and AMD1S show that IPS^{4}o also outperforms its non-in-place competitors on any machine and that IPS^{4}o is much faster than its in-place competitors except in the case of Sorted and Ones inputs.

In Figure 8 (g–h), we give running times of Pair and 100Bytes data types on machine Intel2S with uniformly distributed keys. We see that IPS^{4}o outperforms its competitors, e.g., by a factor of 1.33 (non-in-place competitor) and by a factor of 2.67 (its in-place competitor) for 2^{29} 100Bytes elements. Further benchmarks on machines Intel4S and AMD1S show similar running times.

Figure 7 depicts the speedup of parallel algorithms executed on different numbers of cores relative to our sequential implementation IS^{4}o on Intel2S, sorting Uniform input ($n = 2^{30}$). We see that IPS^{4}o outperforms its competitors on any number of cores. IPS^{4}o outperforms IS^{4}o on 32 cores by a factor of 28.71, whereas its fastest non-in-place competitor, PBBS, outperforms IS^{4}o just by a factor of 14.54. The in-place algorithms, MCSTLubq and MCSTLbq, scale similarly to PBBS up to 16 cores but begin lagging behind for larger numbers of cores. Further measurements show that IPS^{4}o scales similarly on AMD1S. On Intel4S, IPS^{4}o scales well on the first processor. However, as the input data is stored in the memory of the first processor, adding the second, third and fourth processors speeds up IPS^{4}o by an additional factor of only 1.45; again caused by the slower memory modules (DDR4 vs. DDR3) and the long load delays due to a ring interconnect between four sockets.

6 Conclusion and Future Work

In-place super scalar samplesort (IPS^{4}o) is among the fastest comparison-based sorting algorithms both sequentially and on multi-core machines. The algorithm can also be used for data distribution and local sorting in distributed memory parallel algorithms (e.g., [2]). Somewhat surprisingly, there is even an advantage over non-in-place algorithms because IPS^{4}o saves on overhead for memory allocation, associativity misses and write allocate misses. Compared to previous parallel in-place algorithms, improvements by more than a factor of two are possible. The main case where IPS^{4}o is slower than the best competitors (s^3-sort and BlockQuicksort) is for sequentially sorting large objects (Quartet and 100Bytes, see the full paper [3]) because IPS^{4}o moves elements twice in one distribution step. In this case, the overhead for the oracle information of s^3-sort is small and we could try an almost-in-place variant of s^3-sort with element-wise in-place permutation.

Several improvements of IPS^{4}o can be considered. Besides careful adaptation of parameters like k, b, α, and the choice of base case algorithm, one would like to avoid contention on the bucket pointers in the block permutation phase when t is large. Perhaps the most important improvement would be to make IPS^{4}o aware of non-uniform memory access costs (NUMA) depending on the memory module holding a particular piece of data. This can be done by preferably assigning pieces of the input array to "close-by" cores both for local classification and when switching to sequential sorting. In situations with little NUMA effects, we could ensure that our data blocks correspond to pages of the virtual memory. Then, one can replace block permutation with relabelling the virtual memory addresses of the corresponding pages.

Coming back to the original motivation for an alternative to quicksort variants in standard libraries, we see IPS^{4}o as an interesting candidate. The main remaining issue is the code complexity. When code size matters (e.g., as indicated by a compiler flag like `-Os`), quicksort should still be used. Formal verification of the correctness of the implementation might help to increase trust in the remaining cases.

Acknowledgements. We would like to thank the authors of [28, 10] for sharing their code for evaluation. Timo Bingmann and Lorenz Hübschle-Schneider [16] kindly provided code that was used as a starting point for our implementation.

References

1 Lars Arge, Michael T Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *20th Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 197–206. ACM, 2008.

2 Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. Practical massively parallel sorting. In *27th ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA)*, 2015. doi:10.1145/2755573.2755595.

3 Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-place parallel super scalar samplesort (IPSSSSo). *arXiv preprint arXiv:1705.02257*, 2017. URL: https://arxiv.org/abs/1705.02257.

4 Timo Bingmann and Peter Sanders. Parallel string sample sort. In *European Symposium on Algorithms*, pages 169–180. Springer, 2013.

5 Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 189–199. ACM, 2010.

6 Guy E. Blelloch, Charles E. Leiserson, Bruce M Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 3–16. ACM, 1991.

7 G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In *6th Workshop on Algorithm Engineering and Experiments*, 2004.

8 Jing-Chao Chen. A simple algorithm for in-place merging. *Information Processing Letters*, 98(1):34–40, 2006. doi:10.1016/j.ipl.2005.11.018.

9 Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and Ruchir Puri. PARADIS: an efficient parallel algorithm for in-place radix sort. *Proceedings of the VLDB Endowment*, 8(12):1518–1529, 2015.

10 Stefan Edelkamp and Armin Weiss. BlockQuicksort: Avoiding branch mispredictions in quicksort. In *24th European Symposium on Algorithms (ESA)*, volume 57 of *LIPIcs*, 2016.

11 Gianni Franceschini. Proximity mergesort: Optimal in-place sorting in the cache-oblivious model. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'04, pages 291–299, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=982792.982833.

12 Gianni Franceschini and Viliam Geffert. An in-place sorting with O(N log N) comparisons and O(N) moves. *J. ACM*, 52(4):515–537, July 2005. doi:10.1145/1082036.1082037.

13 W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, July 1970. doi:10.1145/321592.321600.

14 Viliam Geffert and Jozef Gajdoš. Multiway in-place merging. In Mirosław Kutyłowski, Witold Charatonik, and Maciej Gębala, editors, *17th Symposium on Fundamentals of Computation Theory (FCT)*, volume 5699 of *LNCS*, pages 133–144. Springer, 2009. doi:10.1007/978-3-642-03409-1_13.

15 Charles A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

16 Lorenz Hübschle-Schneider. Super scalar sample sort. https://github.com/lorenzhs/ssssort, retrieved September 15, 2016.

17 Tomasz Jurkiewicz and Kurt Mehlhorn. On a model of virtual address translation. *Journal of Experimental Algorithmics (JEA)*, 19, 2015.

18 K. Kaligosi and P. Sanders. How branch mispredictions affect quicksort. In *14th European Symposium on Algorithms (ESA)*, volume 4168 of *LNCS*, pages 780–791, 2006.

19 Kanela Kaligosi and Peter Sanders. How branch mispredictions affect quicksort. In *European Symposium on Algorithms*, pages 780–791. Springer, 2006.

20 Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. Practical in-place mergesort. *Nord. J. Comput.*, 3(1):27–40, 1996.

21 Marek Kokot, Sebastian Deorowicz, and Maciej Dlugosz. Even faster sorting of (not only) integers. *arXiv preprint arXiv:1703.00687*, 2017.

22 Shrinu Kushagra, Alejandro López-Ortiz, J. Ian Munro, and Aurick Qiao. Multi-pivot quicksort: Theory and experiments. In *Meeting on Algorithm Engineering & Experiments (ALENEX)*, pages 47–60, Philadelphia, PA, USA, 2014. AMS. URL: `http://dl.acm.org/citation.cfm?id=2790174.2790180`.

23 David R. Musser. Introspective sorting and selection algorithms. *Softw., Pract. Exper.*, 27(8):983–993, 1997.

24 N. Rahman. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, chapter Algorithms for Hardware Caches and TLB, pages 171–192. Springer, 2003.

25 James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

26 Peter Sanders and Jan Wassenberg. Engineering a multi-core radix sort. In *17th Euro-Par Conference*, volume 6853 of *LNCS*, pages 160–169. Springer, 2011.

27 Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *12th European Symposium on Algorithms (ESA)*, volume 3221 of *LNCS*, pages 784–796. Springer, 2004.

28 Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 68–70. ACM, 2012.

29 J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In *13th Euro-Par Conference*, volume 4641 of *LNCS*, pages 682–694. Springer, 2007. `doi:10.1007/978-3-540-74466-5_72`.

30 P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN Enterprise 10000. In *PDP*, pages 372–381. IEEE Computer Society, 2003. `doi:10.1109/EMPDP.2003.1183613`.

31 Vladimir Yaroslavskiy. Dual-pivot quicksort. *Research Disclosure*, 2009.

Online Bin Packing with Cardinality Constraints Resolved

János Balogh[1], József Békési[2], György Dósa[3], Leah Epstein[4], and Asaf Levin[5]

1 **Department of Applied Informatics, Gyula Juhász Faculty of Education, University of Szeged, Hungary**
balogh@jgypk.u-szeged.hu
2 **Department of Applied Informatics, Gyula Juhász Faculty of Education, University of Szeged, Hungary**
bekesi@jgypk.u-szeged.hu
3 **Department of Mathematics, University of Pannonia, Veszprém, Hungary**
dosagy@almos.vein.hu
4 **Department of Mathematics, University of Haifa, Haifa, Israel**
lea@math.haifa.ac.il
5 **Faculty of Industrial Engineering and Management, The Technion, Haifa, Israel**
levinas@ie.technion.ac.il

Abstract

Cardinality constrained bin packing or bin packing with cardinality constraints is a basic bin packing problem. In the online version with the parameter $k \geq 2$, items having sizes in $(0, 1]$ associated with them are presented one by one to be packed into unit capacity bins, such that the capacities of bins are not exceeded, and no bin receives more than k items. We resolve the online problem in the sense that we prove a lower bound of 2 on the overall asymptotic competitive ratio. This closes the long standing open problem of finding the value of the best possible overall asymptotic competitive ratio, since an algorithm of an absolute competitive ratio 2 for any fixed value of k is known. Additionally, we significantly improve the known lower bounds on the asymptotic competitive ratio for every specific value of k. The novelty of our constructions is based on full adaptivity that creates large gaps between item sizes. Thus, our lower bound inputs do not follow the common practice for online bin packing problems of having a known in advance input consisting of batches for which the algorithm needs to be competitive on every prefix of the input. Last, we show a lower bound strictly larger than 2 on the asymptotic competitive ratio of the online 2-dimensional vector packing problem, and thus provide for the first time a lower bound larger than 2 on the asymptotic competitive ratio for the vector packing problem in any fixed dimension.

1998 ACM Subject Classification F.2.2 Sequencing and Scheduling, G.2.1 Combinatorial Algorithms

Keywords and phrases Online algorithms, bin packing, cardinality constraints, lower bounds

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.10

1 Introduction

Bin packing with cardinality constraints (CCBP, also called cardinality constrained bin packing) is a well-known variant of bin packing [18, 19, 17, 9, 10, 11, 15]. In this problem, a parameter k is given. Items of indices $1, 2, \ldots, n$, where item i has a size $s_i \in (0, 1]$ are to be

split into subsets called bins, such that the total size of items packed into each bin is at most 1, and no bin has more than k items. In the standard bin packing problem, only the first condition is required.

CCBP is a special case of vector packing (VP) [14]. In VP with dimension $d \geq 2$, a set of items, where every item is a non-zero d-dimensional vector whose components are rational numbers in $[0, 1]$, are to be split into subsets (called bins in this case as well) such that the vector sum of every subset does not exceed 1 in any component. Given an input for CCBP, an input for VP is created as follows. For every item, let the first component be $\frac{1}{k}$, the second component is s_i, and the remaining components are equal to zero (or to $\frac{1}{k}$).

In this paper we study online algorithms, which receive input items one by one, and pack each new item irrevocably before the next item is presented, into an empty (new) bin or non-empty bin. Such algorithms receive an input as a sequence, while offline algorithms receive an input as a set. By the definition of CCBP, an item i can be packed into a non-empty bin B if the packing is feasible both with respect to the total size of items already packed into that bin and with respect to the number of packed items (i.e., the bin contains items of total size at most $1 - s_i$ and it contains at most $k - 1$ items). An optimal offline algorithm, which uses a minimum number of bins for packing the items, is denoted by OPT. For an input L and algorithm A, we let $A(L)$ denote the number of bins that A uses to pack L. We also use $OPT(L)$ to denote the number of bins that OPT uses for a given input L. The absolute competitive ratio of an algorithm A is the supremum ratio over all inputs L between the number of its bins $A(L)$ and the number of the bins of OPT, $OPT(L)$. The asymptotic approximation ratio is the limit of absolute approximation ratios R_K when K tends to infinity and R_K takes into account only inputs for which OPT uses at least K bins, that is the asymptotic competitive ratio of A is

$$\lim_{K \to \infty} \sup_{OPT(L) \geq K} \frac{A(L)}{OPT(L)} .$$

The term *competitive ratio* is used for online algorithms instead of *approximation ratio* and it is equivalent. In this paper we mostly deal with the asymptotic competitive ratio, and also refer to it by the term competitive ratio. When we discuss the absolute competitive ratio, we use this last term explicitly.

In this paper, we resolve the long standing open problem of online CCBP, in the sense that we find the best overall asymptotic competitive ratio and the best overall absolute competitive ratio. An algorithm with an asymptotic competitive ratios of 2 has been designed by Babel et al. [4], and a similar algorithm was shown to have an absolute competitive ratio of 2 [6] (earlier, it was known that the competitive ratio of a suitable variant of First Fit is below 2.7 for any k [18]). However, prior to this work, all lower bounds were strictly smaller than the best lower bounds for standard bin packing [23, 5]. With the exception of the case $k = 2$ for which simple algorithms have competitive ratios of 1.5 [18, 10], and a more sophisticated algorithm has a competitive ratio of at most 1.44721 [4], all lower bounds on the competitive ratio were implied by partial inputs of ones used to prove lower bounds for standard bin packing [24, 23, 5] (such lower bounds can be used for $k \geq \frac{1}{\delta}$ when all items have sizes no smaller than δ, for a fixed value $\delta > 0$), and modifications of such inputs [4, 12, 6]. That is, all lower bounds had the form where a number of lists may be presented, each list has a large number of items of a certain size (the sequence of sizes of the different lists is increasing, and the numbers of items in the lists are not necessarily equal). The unknown factor is the number of presented lists, that is, the input can stop after any of the lists. See Table 1 for values of previously known lower bounds (and note that for $k = 3, 4, 5, 6$ algorithms with competitive ratios strictly below 2 are known [10]).

Table 1 Bounds for $2 \leq k \leq 10$. The middle column contains the previously best known asymptotic lower bounds on the asymptotic competitive ratio for CCBP with parameter k. The right column contains our improved lower bounds.

Value of k	previous lower bound	new lower bound
2	1.42764 [12]	$\frac{10}{7} \approx 1.42857$
3	$\frac{3}{2} = 1.5$ [4]	1.55642
4	$\frac{3}{2} = 1.5$ [12]	1.63330
5	$\frac{3}{2} = 1.5$ [6]	1.69776
6	$\frac{3}{2} = 1.5$ [24]	1.74093
7	$\frac{217}{143} \approx 1.51748$ [6]	1.77223
8	$\frac{32}{21} \approx 1.52381$ [6]	1.79634
9	$\frac{189}{124} \approx 1.524194$ [6]	1.81563
10	$\frac{235}{154} \approx 1.52597$ [6]	1.83148
200000	1.54037 [5]	1.99999
$k \to \infty$	$\frac{248}{161} \approx 1.54037$ [5]	**2**

In this work, we take a different approach for proving lower bounds, where many of the item sizes are based on the complete and precise action of the algorithm up to the time it is presented. While some ingredients of our approach were used for the very limited special case of $k = 2$ in the past [7, 4, 12], it was unclear how and if it could be used for $k > 2$. In a nutshell, in these lower bound sequences for $k = 2$, sub-inputs were constructed such that items packed in certain ways (for example, as the second item of a bin) had much larger sizes than items of the same sub-input packed in other ways. Here, we generalize the approach for larger values of k by defining careful constructions where *sufficiently large multiplicative gaps* are created. This requires much more delicate procedures where item sizes are defined.

Additionally, we improve the lower bounds for all values of k, and in particular, prove lower bounds above the best known lower bound on the competitive ratio for standard online bin packing, 1.54037 [5] for $k \geq 3$. Already for $k = 3$ our lower bound is above 1.55, and already for $k = 4$, our lower bound is above the competitive ratio of many algorithms for standard online bin packing (see for example [21, 22]).

Our result for CCBP provides, in particular, a lower bound of 2 for the asymptotic competitive ratio of VP in two dimensions. The previously known lower bounds for VP are as follows. The best results for constant dimensions are fairly low, and tending to 2 as the dimension d grows to infinity [13, 8, 7], while a lower bound of $\Omega(d^{1-\varepsilon})$ was given by Azar et al. [2] for the case where both d and the optimal cost are functions of a common parameter n that grow to infinity when n grows to infinity, and thus this result does not give any lower bound on the competitive ratio for constant values of d (see also [1, 3] for results on vectors with small components). In particular, the best lower bound for $d = 2$ prior to this work was 1.67117 [13, 8, 7]. An upper bound of $d + 0.7$ on the competitive ratio is known [14]. We conclude this work by establishing a lower bound strictly larger than 2 on the competitive ratio of 2-dimensional VP, and thus we show here for the first time that the 2-dimensional VP is provably harder for online algorithms than its special case of CCBP.

Note that the offline CCBP problem is NP-hard in the strong sense, and approximation schemes are known for it [9, 11, 15]. We note that for online CCBP, it is sometimes the

case that the competitive ratio for some specific algorithms for CCBP is larger by 1 with comparison to that of the corresponding algorithms for standard bin packing [18, 16, 20, 10]. Interestingly, this is not the case with respect to the results shown in this paper.

1.1 Paper outline

We discuss general properties in Section 2, and we define procedures for constructing sub-inputs in Section 3. Our main result, an overall lower bound of 2 on the competitive ratio of any online algorithm for CCBP is proved in Section 4, and improved lower bounds for fixed values of k are given in Section 5. Our result for VP is established in Section 6. Omitted proofs appear in the full version of this work.

2 Preliminaries

The analysis of the lower bounds on the asymptotic competitive ratio of online algorithms will be based on the following lemma that basically allows us to disregard a constant number of bins in the costs of the optimal solution and the solution returned by the algorithm.

▶ **Lemma 1.** *Consider an algorithm ALG, such that the asymptotic competitive ratio of the algorithm ALG is at most R, where $R \geq 1$ is a fixed value, and let $f(n)$ denote a positive function such that $f(n) = o(n)$ and for any input, $ALG(I) \leq R \cdot OPT(I) + f(OPT(I))$. Let $C_a \geq 0$, $C_b \geq 0$ be constants. Assume that for a given integer N_0, for any integer $n \geq N_0$ there is an input I^n for which $OPT(I^n) = \Omega(n)$, then we have*

$$R \geq \limsup_{n\to\infty} \frac{ALG(I^n) + C_a}{OPT(I^n) - C_b} .$$

Proof. We have

$$\frac{ALG(I^n) + C_a}{n} \leq R \cdot \frac{OPT(I^n) - C_b}{n} + \frac{C_a + R \cdot C_b}{n} + \frac{f(OPT(I^n))}{n}$$

for any $n \geq N_0$.

Since $ALG(I^n) + C_a \geq OPT(I^n) - C_b$ and $OPT(I^n) - C_b = \Omega(n)$ while $C_a + R \cdot C_b + f(OPT(I^n)) = o(n)$, letting n grow to infinity implies that

$$R \geq \limsup_{n\to\infty} \frac{ALG(I^n) + C_a}{OPT(I^n) - C_b} .$$ ◀

In what follows, we will use Lemma 1 as follows. We construct inputs whose size depends on a parameter N, so that the costs of optimal solutions increase with the input size. We will compare the cost of a fixed online algorithm ALG plus a suitable non-negative constant to the optimal cost minus a suitable non-negative constant by considering their ratio.

3 Constructions of sub-inputs

In this section we introduce the core of our lower bound constructions. In such constructions, we *adaptively* present inputs that are based on the behavior of the algorithm. More specifically, we define several procedures that construct sub-inputs according to certain conditions. Similarly to [4, 12, 7] (and other work on online problems), a new input item is presented at each time, where its size is based on the action of the algorithm on previous items. For example, if the previous item was packed into an empty bin, the size of the next item is

different from the size that would be used if the previous item is added to a non-empty bin. In order to ensure that the properties are satisfied, we will define invariants, and we will prove the specific properties that we need in the sequel via induction. The constructions use k as a parameter since they are defined to be used for CCBP. However, they can be used for any packing problem of items into bins and the property that k is the cardinality constraint is not used in the constructions of sub-inputs (it is used later in the analysis of inputs constructed using these sub-inputs). Thus, if the constructions are used for other problems like we do for VP, the parameter k should be specified.

In the first procedure, the most important property is that there will be a gap between two types of items constructed by applying the procedure, in the sense that the procedure creates items that will be called small and items that will be called large, any large item is larger than any small item, and there is a requirement on the size ratio that will be satisfied (a multiplicative gap between the size of the smallest large item and the largest small item). Such constructions differ from previous work [4, 12, 7] where only an additive gap was created. The gap was always positive, but it could be arbitrarily small. In particular, one limitation was that it was unknown how such an approach could be used for CCBP with parameter $k > 2$.

We will also use this method to construct sub-inputs with large items, such that there is a multiplicative gap in the differences between 1 and the items sizes. This new method will allow us to provide a tight overall result for CCBP, new and significantly improved lower bounds on the asymptotic competitive ratio for CCBP with fixed values of k, and our improved lower bound for VP.

3.1 Procedure SMALL

In this first procedure called SMALL, a rational value $0 < \varepsilon \leq 1$, and an integer upper bound N on the number of items to be presented are given. The goal is to present (at most) N items of sizes in $(0, \varepsilon]$, such that every item will be seen as either a small item or a large item, and such that any large item is more than k times larger than any small item. In fact, a stronger requirement on the item sizes will hold. Moreover, all item sizes will be rational. Given two logical conditions, C_1 and C_2 specified for each construction (such that for every packed item, exactly one of them holds), a new item will be defined as *small* if C_1 holds and it will be defined as *large* if C_2 holds. There is a third condition C_3 that is based on the packing of the prefix of items introduced so far, and the sub-input is stopped if C_3 holds.

Let N be an upper bound on the number of items that will be created by the procedure. Let $N' \leq N$ be the actual number of items (where N is known in advance and used for the sequence construction, while N' is not necessarily known in advance and it becomes known when C_3 holds for the first time). The item sizes $a_1, a_2, \ldots, a_{N'}$ will be defined based on another sequence $x_1, x_2, \ldots, x_{N'}$, such that $a_i = \varepsilon \cdot k^{-x_i}$ for $1 \leq i \leq N'$. The values x_i will be integral in order to ensure that the values a_i will be rational. There will also be two sequences of values $\tau_1, \ldots, \tau_{N'}$ and $\rho_1, \ldots, \rho_{N'}$, representing thresholds on item sizes of further items.

Let $\tau_0 = 2^{N+2}$, $\rho_0 = 2^{N+3}$, and $i = 1$. The process is defined as follows for any given value of i (such that $1 \leq i \leq N'$). Let $x_i = \frac{\tau_{i-1}+\rho_{i-1}}{2}$ (we will show that these values are integers). After the algorithm packs item i, if C_1 holds, let $\tau_i = \tau_{i-1}$ and $\rho_i = x_i$ and if C_2 holds, let $\tau_i = x_i$ and $\rho_i = \rho_{i-1}$. If C_3 holds or $i = N$, stop and otherwise increase i by 1.

Intuitively, the process is as follows. The interval (τ_i, ρ_i) contains the x_j values of all further items (with $j > i$), and for $j \leq i$, all items satisfying C_1 have x_j values in $[\rho_i, \rho_0)$ and all items satisfying C_2 have x_j values in $(\tau_0, \tau_i]$. In each iteration i, the new values τ_i, ρ_i are

defined such that these requirements are satisfied. In particular, the x_i values of any item satisfying C_1 are larger than those of items satisfying C_2. Next, we establish the invariants of this procedure.

▶ **Lemma 2.** *Let N' be the number of items. For any i such that $1 \leq i \leq N'$, $\rho_i \leq \rho_{i-1}$ and $\tau_i \geq \tau_{i-1}$. Additionally, we have $\rho_i - \tau_i = 2^{N+2-i}$, all x_i values are integral, if item i satisfies C_1, $x_i \geq \rho_{N'}$ and otherwise $x_i \leq \tau_{N'}$.*

Proof. We start with showing that the x_i values as well as ρ_i and τ_i are integral and $\rho_i - \tau_i = 2^{N+2-i}$. We prove this by induction. Indeed $\rho_0 = 2^{N+3}$ that is integral, $\tau_0 = 2^{N+2}$ that is an integer as well. Furthermore, $\rho_0 - \tau_0 = 2^{N+2}$, and $x_1 = 3 \cdot 2^{N+1}$ that is an integer, and no matter if the first item satisfies C_1 or C_2, we have that both ρ_1 and τ_1 are integers, and $\rho_1 - \tau_1 = 2^{N+1}$. Thus, the cases $i = 0$ and $i = 1$ for the induction claim hold. Assume that $\rho_{i-1} - \tau_{i-1} = 2^{N+3-i}$ holds for some i where $1 \leq i \leq N' - 1$. Then,

$$x_i = \frac{\tau_{i-1} + \rho_{i-1}}{2} = \tau_{i-1} + \frac{\rho_{i-1} - \tau_{i-1}}{2} = \tau_{i-1} + 2^{N+3-i} \ ,$$

which is an integer for $1 \leq i \leq N$, since τ_{i-1} is an integer. Moreover, if $\tau_i = \tau_{i-1}$ and $\rho_i = x_i$, then $\rho_i - \tau_i = x_i - \tau_{i-1} = \frac{\rho_{i-1} - \tau_{i-1}}{2}$, and otherwise $\tau_i = x_i$ and $\rho_i = \rho_{i-1}$, then $\rho_i - \tau_i = \rho_{i-1} - x_i = \frac{\rho_{i-1} - \tau_{i-1}}{2}$. In both cases, $\rho_i - \tau_i = 2^{N+2-i}$ and both τ_i and ρ_i are integers. Since, in particular, for any i, $\rho_i > \tau_i$ holds and x_{i+1} is their average, we find $\tau_i < x_{i+1} < \rho_i$. Thus, $\rho_i \leq \rho_{i-1}$ and $\tau_i \geq \tau_{i-1}$ holds for any i.

Finally, since in the case that item i satisfies C_1, we let $\rho_i = x_i$, and in the case that item i satisfies C_2, we let $\tau_i = x_i$, we get $x_i = \rho_i \geq \rho_{i+1} \geq \ldots \geq \rho_{N'}$ in the first case, and $x_i = \tau_i \leq \tau_{i+1} \leq \ldots \leq \tau_{N'}$ in the second case. ◀

▶ **Corollary 3.** *For any item i, $a_i \in \left(\varepsilon \cdot k^{-2^{N+3}}, \varepsilon \cdot k^{-2^{N+2}}\right)$, and in particular $a_i \leq \frac{1}{k^4}$. For any item i_1 satisfying C_1 and any item i_2 satisfying C_2, it holds that $\frac{a_{i_2}}{a_{i_1}} > k$.*

Note that it is possible that the constructed input is such that there are only items satisfying C_1 or only items satisfying C_2.

Proof. The first claim holds by definition. Since we have $x_{i_1} \geq \rho_{N'}$ and $x_{i_2} \leq \tau_{N'}$, we get $\frac{a_{i_2}}{a_{i_1}} > k^{\rho_{N'} - \tau_{N'}}$, Using $\rho_{N'} - \tau_{N'} = 2^{N+2-N'} \geq 4$ as $N' \leq N$, we find $\frac{a_{i_2}}{a_{i_1}} \geq k^4 > k$. ◀

3.2 Procedure LARGE

The second type of input is such that all items have sizes in $(1 - \varepsilon, 1)$ for a given value $\varepsilon > 0$. The construction is the same as before, but the size of the ith item is $b_i = 1 - a_i$. The terms "small" and "large" refer to the difference between the size of the item and 1.

▶ **Corollary 4.** *All b_i for $1 \leq i \leq N$ are in $(1 - \varepsilon \cdot k^{-2^{N+2}}, 1 - \varepsilon \cdot k^{-2^{N+3}})$. The sizes of any small item i_s and any large item i_l satisfy $1 - b_{i_l} > k \cdot (1 - b_{i_s})$.*

3.3 Procedure SMALLandLARGE

We will also use a procedure where the conditions C_1 and C_2 are not fixed, and they are based on additional properties of the packing and the input that has been presented so far. Moreover, in this case the size of each item is based on a_i, but it is fixed for each item separately (it will be either a_i or $1 - a_i$). In this construction the sub-input will be decomposed into *parts* where for an item of an odd indexed part the size of the item will be $1 - a_i$, whereas for an item of an even indexed part the size of the item will be a_i. The definitions of C_1 and C_2 will also depend on the parity of the index of the part containing the item. This procedure is called SMALLandLARGE.

4 A lower bound of 2 for CCBP

The general structure of inputs constructed in this section is as follows. There are a large number of very small items, such that the first item packed into a bin by the algorithm is significantly larger than small items packed as a second item or later. Afterwards, there are two cases. In the first case there are very large items (of sizes almost 1) that can be combined with $k-1$ items that arrived earlier, but only with those that are smaller. Thus, an optimal solution can pack all items densely except for those items that are first in their bins (for the algorithm). The algorithm cannot use its previously packed bins again to pack new items, and therefore the best approach is to pack a large number of items into each bin (otherwise the percentage of larger small items is larger, which makes the optimal packing more sparse, but the algorithm has an even larger number of bins, and the effect of the last property is more significant). Another option is that instead of the very large items, items slightly larger than $\frac{1}{2}$ will arrive, in which case it turns out that the algorithm should have packed $k-1$ items into each bin (so that a new item could be still packed there). For very large values of k, the two values $k-1$ and k are not very different, and since the algorithm does not know which items will arrive, packing $k-1$ items into each bin (if k is very large) is a good strategy. The result of packing $k-1$ items into each bin is that in the first case the very large items increase the number of bins roughly by a factor of 2, while an optimal solution has relatively few bins with k small items. Note that the order of options in the construction below is reversed for the sake of convenience.

Let N be a large integer. Apply procedure SMALL with $\varepsilon = 1$ for the construction of N items (i.e., condition C_3 never happens). The condition C_2 is that the item is packed as the first item of some bin (into an empty bin), and the condition C_1 is that the item is packed into a non-empty bin. The item sizes are no larger than $\frac{1}{k^4}$. The multiplicative gap between the smallest large item and the largest small item is larger than k. The N items presented so far will be called the first phase items. Let $\delta > 0$ denote the largest size of any first phase item packed not as a first item of a bin (the largest small item). Let $\alpha = k \cdot \delta$. Any first phase item that is packed as the first item of a bin (a large item) has size strictly above α. Let $\Delta < \frac{1}{k^3}$ be the largest size of any first phase item. Obviously, $1 - k\Delta > 1 - \frac{1}{k^2} > \frac{1}{2}$.

For the first phase items, let X_k denote the number of bins packed by the algorithm that contain k items, and let Y denote the number of other bins (such that there are $X_k + Y$ bins in total after N items have been presented).

The first phase items are followed by another set of items called the second phase items. This set of items is selected out of two possible options. The first option is that $\lceil \frac{N}{k-1} \rceil$ items of size $1 - k\Delta$ arrive, and the second option is that $\lceil \frac{N - X_k - Y}{k-1} \rceil$ items of size $1 - \alpha = 1 - k\delta$ arrive. In both cases it is possible to create an offline solution such that each bin (except for possibly two bins) has k items. In the first case, an offline solution has $\lceil \frac{N}{k-1} \rceil$ bins, each with one item of size $1 - k\Delta$ and an arbitrary subset of $k-1$ first phase items (the last bin may have a smaller number of such items). Such a solution is optimal. In the second case, an offline solution has $\lceil \frac{N - X_k - Y}{k-1} \rceil$ bins, each with one item of size $1 - k\delta$ and $k-1$ small first phase items, and $\lceil \frac{X_k + Y}{k} \rceil$ bins with k large first phase items (for each one of these two bin types, the last bin may have a smaller number of such items). Indeed the last solution is an optimal solution though we will only use that it is a feasible solution.

In the first case, the algorithm cannot use the bins that already have k items for packing second phase items, and its cost is at least $X_k + \lceil \frac{N}{k-1} \rceil \geq X_k + \frac{N}{k-1}$. In the second case, the algorithm cannot use any of its bins to pack any second phase item, as each bin has a large

first phase item of size above α, so its cost is

$$X_k + Y + \left\lceil \frac{N - X_k - Y}{k-1} \right\rceil \geq X_k + Y + \frac{N - X_k - Y}{k-1} \ .$$

We call the two inputs (of the two cases) I_1 and I_2. Obviously, since each input consists of more than N items, $OPT(I_1) = \Omega(\frac{N}{k})$ and $OPT(I_2) = \Omega(\frac{N}{k})$ hold. Letting $N = kn$ provides an input I^n as required. By Lemma 1, we will analyze modified competitive ratios of the form $\frac{ALG(I)+C_a}{OPT(I)-C_b}$ for fixed constants C_a and C_b.

For the input I_1, $OPT(I_1) - 1 \leq \frac{N}{k-1}$ and $ALG(I_1) \geq X_k + \frac{N}{k-1}$. For the input I_2, $OPT(I_2) - 2 \leq \frac{N-X_k-Y}{k-1} + \frac{X_k+Y}{k}$ and $ALG(I_2) \geq X_k + Y + \frac{N-X_k-Y}{k-1}$.

First, we analyze the competitive ratio r for input I_2 and show that it tends to 2 as k grows to infinity. Let $Z = X_k + Y$. We have $OPT(I_2) - 2 \leq \frac{N-Z}{k-1} + \frac{Z}{k}$ and $ALG(I_2) \geq Z + \frac{N-Z}{k-1}$. Thus,

$$r \geq \frac{kZ(k-1) + k(N-Z)}{k(N-Z) + (k-1)Z} = \frac{Z(k^2-2k) + kN}{kN - Z} \ .$$

Since $Z \geq \frac{N}{k}$ and the last lower bound on r is a ratio between an increasing function of Z and a decreasing function of Z, we conclude that by substituting $\frac{N}{k}$ instead of Z in the last bound, we achieve a valid lower bound on r. Thus, we have $r \geq \frac{N(k-2)+kN}{kN-\frac{N}{k}} = \frac{2-2/k}{1-1/(k^2)} = \frac{2k}{k+1}$ and the last bound tends to 2 when k grows to infinity. By Lemma 1, the overall (asymptotic) competitive ratio is at least 2. Since there is a 2-competitive algorithm for any value of k [4] (even for the absolute competitive ratio [6]), we establish the following.

▶ **Theorem 5.** *The overall best possible asymptotic and absolute competitive ratios for bin packing with cardinality constraints are equal to* 2.

To obtain a better lower bound on the asymptotic competitive ratio r for a fixed value of $k \geq 3$, we use I_1 as well. By $r \geq \frac{ALG(I_1)}{OPT(I_1)-1} \geq \frac{X_k+N/(k-1)}{N/(k-1)}$ we have $(k-1)X_k \leq (r-1) \cdot N$. By counting arguments, $N \leq kX_k + (k-1)Y$ holds, and we get $X_k \geq N - (k-1)Z$, and $(r-1)N \geq (k-1)X_k \geq (k-1)(N-(k-1)Z) = (k-1)N - (k-1)^2 \cdot Z$. Rearranging gives

$$Z \geq \frac{(k-r)N}{(k-1)^2}.$$

As we saw earlier, by using I_2 we have $r \geq \frac{Z(k^2-2k)+kN}{kN-Z}$, which is equivalent to

$$Z(k^2 - 2k + r) \leq kN(r-1).$$

Combining the lower bound and upper bound on Z results in

$$\frac{(k-r)N(k^2-2k+r)}{(k-1)^2} \leq kN(r-1),$$

or equivalently

$$r^2 + r(k^3 - k^2 - 2k) - (2k^3 - 4k^2 + k) \geq 0.$$

Since $k^3 - k^2 - 2k \geq 0$ holds for $k \geq 2$ and $2k^3 - 4k^2 + k > 0$ holds for $k \geq 2$, it is sufficient to find the (unique) positive root which is equal to $\frac{2k+k^2-k^3+\sqrt{(k^3-k^2-2k)^2+4(2k^3-4k^2+k)}}{2}$. The last expression is a lower bound on r and thus the following holds.

▶ **Theorem 6.** *For any $k \geq 3$, the asymptotic competitive ratio for bin packing with cardinality constraints is at least*

$$\frac{2k + k^2 - k^3 + \sqrt{k^6 - 2k^5 - 3k^4 + 12k^3 - 12k^2 + 4k}}{2}.$$

The last lower bound is equal to approximately 1.54983 for $k = 3$, 1.63330 for $k = 4$, 1.69047 for $k = 5$, 1.73214 for $k = 6$, 1.76388 for $k = 7$, 1.78888 for $k = 8$, 1.80909 for $k = 9$, and 1.82575 for $k = 10$. For $k = 2$ the resulting lower bound is $\sqrt{2}$ and the construction (for the case $k = 2$) is indeed similar to that of [7, 4].

5 Better lower bounds for CCBP for some small values of k

In this section we prove the next theorem that improves the resulting bounds of Theorem 6 for these values of k.

▶ **Theorem 7.** *The following approximate values are lower bounds on the asymptotic competitive ratio:The value* 1.42857 *for* $k = 2$ *(the exact value of this lower bound is* $\frac{10}{7}$*),* 1.55642 *for* $k = 3$*,* 1.69776 *for* $k = 5$*,* 1.74093 *for* $k = 6$*,* 1.77223 *for* $k = 7$*,* 1.79634 *for* $k = 8$*,* 1.81563 *for* $k = 9$*, and* 1.83148 *for* $k = 10$*.*

6 Vector packing

As explained in the introduction, vector packing is a generalization of CCBP, and thus the results of the previous sections imply, in particular, a lower bound of 2 on the asymptotic competitive ratio for VP in two or more dimensions. In this section we show that VP is more general, by improving the result, and showing a lower bound above 2 for VP with constant dimensions. Our result is the first lower bound strictly above 2 for any fixed dimension VP (recall that currently, the best known upper bound for d-dimensional VP is $d + 0.7$ and for 2-dimensional VP 2.7 [14]). We prove the result for two dimensions (and the result for higher dimensions follows since the asymptotic competitive ratio is monotone in the dimension, as any d-dimensional vector can be augmented by $d' - d$ zeroes to become a d'-dimensional vector). Once again we consider a fixed deterministic online algorithm ALG, but this time it is an algorithm for VP. Let R be the asymptotic competitive ratio.

The main idea of the lower bound is at follows. First, there are items whose first component is $\frac{1}{k}$ for an appropriately chosen integer k, while the second components are very small. The items are such that the second components are sufficiently larger for items packed first into their bins by the algorithm compared to those that are not packed first. Afterwards, one option is that the following items have a very large second component and their first component is zero (this is equivalent to the items in the construction for CCBP). Every such item could be packed with k items that arrived earlier, but never with the first item of a bin of the algorithm, and thus the new items require new bins, while an optimal solution can pack almost everything densely. For this option it is most profitable for the algorithm to pack k items in each bin. In the other cases, it turns out that it is better to pack much less than k items per bin, as further items will have first components of $\frac{a}{k}$ for an integer value of a (which is selected based on the action of the algorithm). Those items will have second components above $\frac{1}{3}$, and there may be further items whose second components are above $\frac{1}{2}$.

Let $k \geq 10$ be a large integer. The set of inputs we define will consist of at most three phases (where a phase is a sub-input). The first phase of the input is based on the construction for CCBP as follows. For a large integer $N \geq 1000$, there are N items whose

first component is $\frac{1}{k}$. The second components of items are constructed using procedure SMALL with $\varepsilon = k^{-2^{N+4}}$, such that SMALL is applied for the construction of N elements (i.e., condition C_3 never happens). The condition C_2 is that the item is packed as the first item of some bin (i.e., it is packed into an empty bin), and the condition C_1 is that the item is packed into a non-empty bin. The N (two-dimensional) items presented so far will be called the first phase items. The second components of the first phase items are no larger than $k^{-2^{N+4}} \leq \frac{1}{k^4}$. Due to the value of the first component, in any packing every bin has at most k first phase items. A first phase item packed as the first item of a bin will be called large and any other first phase item will be called small.

The multiplicative gap between the smallest second component of any large item and the largest second component of any small item is greater than k. Let $\delta > 0$ denote the largest second component of any small first phase item. Let $\alpha = k \cdot \delta$. Any large first phase item has a second component strictly above α. Let $\Delta < \frac{1}{k^3}$ be the largest second component of any first phase item. Obviously, $1 - \alpha = 1 - k\delta > 1 - \frac{1}{k^3} > 0.999$.

Let X_i denote the number of bins packed with i first phase items and let $\Theta = \frac{(\sum_{i=1}^{k} X_i)}{N}$, where $\Theta \leq 1$ as every bin has at least one item out of the N items. Let the input of first phase items be denoted by I. At this time, any k items can be packed into a bin, and thus $OPT(I) \leq \lceil \frac{N}{k} \rceil$. If $ALG(I) = \Theta N \geq \frac{3N}{k}$, we get $R \geq \frac{ALG(I)}{OPT(I)-1} \geq 3$. Thus, we assume in what follows that $\Theta < \frac{3}{k}$. Since every bin of the algorithm contains exactly one large item and the remaining items are small, there are $\Theta N < \frac{3N}{k}$ large items and at least $N - \Theta N > \frac{(k-3)N}{k} \geq \frac{7N}{10}$ small items.

The first option for the second part of the input is similar to the construction for CCBP (the second part of the input will also be the last part of the input in this specific case). The next phase of items will consist of $\lceil \frac{N-\Theta \cdot N}{k} \rceil$ items called second phase items, whose first component is zero and the second component is $1 - \alpha = 1 - k\delta$. This input (consisting of the first phase items and the second phase items) is called I'. By the following lemma we have $1 + (k-1)\Theta \leq R$.

▶ **Lemma 8.** *We have* $ALG(I') = \Theta N + \lceil \frac{N-\Theta N}{k} \rceil \geq \Theta N + \frac{N-\Theta N}{k} = \frac{N+(k-1)\Theta N}{k}$ *and* $OPT(I') - 2 \leq \frac{N-\Theta N}{k} + \frac{\Theta N}{k} = \frac{N}{k}$.

Proof. It is possible to create a feasible solution for I' where each bin (except for possibly two bins) has k first phase items. This solution has $\lceil \frac{N-\Theta N}{k} \rceil$ bins, each with one second phase item and k small first phase items, and $\lceil \frac{\Theta N}{k} \rceil$ bins with k large first phase items (for each one of these two bin types, the last bin may have a smaller number of first phase items). Indeed the last solution is an optimal solution (since second phase items cannot be packed with large first phase items), though we will only use that it is a feasible solution. We conclude that $OPT(I') - 2 \leq \frac{N-\Theta N}{k} + \frac{\Theta N}{k} = \frac{N}{k}$. The algorithm uses a different new bin for each second phase item, since every such item has a second component larger than $\frac{1}{2}$, and every bin with first phase items has a total size above α in its second component. Thus, we get $ALG(I') = \Theta N + \lceil \frac{N-\Theta N}{k} \rceil \geq \Theta N + \frac{N-\Theta N}{k} = \frac{N+(k-1)\Theta N}{k}$. ◀

Let b be an integer such that $b < \frac{k-4}{2}$. For any integer a such that $1 \leq a \leq b$, there will be two possible inputs I_a^1 and I_a^2. All inputs start with the first phase items defined above. The second phase of items is identical for the two inputs I_a^1 and I_a^2 (but it is different for different values of a). Let $\Gamma_a = \lceil \frac{N-N\Theta}{k-2a} \rceil$. Intuitively, when considering an optimal packing of the small first phase items in I_a^1 and I_a^2, most of the bins will contain $k - 2a$ small first phase items, and thus Γ_a is approximately their number. The second phase items are constructed

using SMALL with the same value of k as for the first phase items as follows. The number of items is $N_a = \Gamma_a$ (and once again C_3 never happens and all items are presented). The sizes are built using $\varepsilon = 1$, and the conditions C_1 and C_2 are as follows. We let C_2 be the condition that the item is packed into a bin that does not have a second phase item, and C_1 is the condition that the item is packed into a bin that already has a second phase item. The first component of each item is $\frac{a}{k}$. Given the ith output of SMALL denoted by z, for the ith item, the second component is defined as $\frac{1}{3} + z$. If z is defined when C_2 holds, we say that the item whose vector is $(\frac{a}{k}, \frac{1}{3} + z)$ is large, and otherwise it is small. Since $0 < z \leq \frac{1}{k^4}$ for any item, the items satisfy that their second components are strictly larger than $\frac{1}{3}$, and they are not larger than $\frac{1}{3} + \frac{1}{k^4} < 0.3335$. Furthermore, we conclude that the difference between the smallest second component of a large second phase item and the largest second component of a small second phase item is at least $k^{-2^{N+3}}$.

Obviously, since second phase items have second components above $\frac{1}{3}$, no bin can have more than two such items. Let Y_1^a and Y_2^a denote the numbers of bins with one second phase item and two second phase items, respectively (note that there may be such bin that contain first phase items and bins that do not, and both kinds are included in these two values according to their numbers of second phase items, while bins with only first phase items are not included). There are Y_2^a small second phase items and $Y_1^a + Y_2^a$ large second phase items (and $Y_1^a + 2Y_2^a = \Gamma_a$). Note that since the first component of second phase items is $\frac{a}{k}$, they could not have been packed into bins with at least $k - a + 1$ first phase items.

Input I_a^1 continues with Γ_a items, each of the form $(\frac{a}{k}, 0.6)$. Let $\frac{1}{3} + \delta'$ be the largest second component of a small second phase item (such that for any large second phase item, its second component is larger than $\frac{1}{3} + 2\delta'$), and observe that since $\delta' \geq k^{-2^{N+3}}$, the total sum of second component of a set of at most k first phase items is at most δ'. Input I_a^2 continues with the third phase items as follows. $\lceil \frac{\Gamma_a + 2Y_2^a}{4} \rceil$ items, each of the form $(\frac{a}{k}, \frac{2}{3} - 2\delta')$, and $\lceil \frac{N}{4k} \rceil$ items, each of the form $(0, 1 - \alpha)$. Let $\Delta_c = \sum_{i=c}^{k} X_i$.

▶ **Lemma 9.** *The costs of the algorithm satisfy*

$$ALG(I_a^1) \geq \Delta_{k-a+1} + Y_2^a + \Gamma_a$$

and

$$ALG(I_a^2) \geq \Delta_{k-a+1} + Y_1^a + Y_2^a + \frac{\Gamma_a + 2Y_2^a}{4} + \frac{N}{4k}\,.$$

Proof. For I_a^1, the algorithm cannot use any bin with at least $k - a + 1$ first phase items to pack any other items (as second phase and third phase items afterwards have a first component of value $\frac{a}{k}$), and the algorithm cannot pack an item of the form $(\frac{a}{k}, 0.6)$ into a bin with two second phase items. Thus, using $\Delta_c = \sum_{i=c}^{k} X_i$, the total number of bins of the algorithm is at least $\Delta_{k-a+1} + Y_2^a + \Gamma_a$.

For I_a^2, the algorithm cannot use any bin with at least $k - a + 1$ first phase items to pack items whose first component is $\frac{a}{k}$, and it cannot use any bins with first phase items to pack items whose second component is $1 - \alpha$. Moreover, since every bin with second phase items has a large second phase item, the algorithm cannot pack any third phase item into a bin containing at least one second phase item (and each bin with a third phase item will contain exactly one third phase item). The only bins that can possibly be used for third phase items are those with at most $k - a$ first phase items and no other items. Thus, the number of bins is at least $\Delta_{k-a+1} + Y_1^a + Y_2^a + \frac{\Gamma_a + 2Y_2^a}{4} + \frac{N}{4k}$. ◀

We next analyze optimal solutions for I_a^1 and I_a^2.

▶ **Lemma 10.** *The cost of the optimal solutions for I_a^1 and I_a^2 satisfy*

$$OPT(I_1^a) \leq \Gamma_a + \lceil \frac{N\Theta}{k} \rceil$$

and

$$OPT(I_a^2) \leq \frac{N}{4k} + \frac{Y_1^a + Y_2^a}{2} + \frac{\Gamma_a + 2Y_2^a}{4} + \frac{9N}{4k^2} + 4 = \frac{N}{4k} + \frac{3\Gamma_a}{4} + \frac{9N}{4k^2} + 4 \ .$$

Proof. For I_a^1 consider the following feasible solution. There are Γ_a bins, each with a second phase item (whose first component is $\frac{a}{k}$ and its second component is in $(\frac{1}{3}, 0.3335)$), one item of the form $(\frac{a}{k}, 0.6)$, and $k - 2a$ first phase items where each such item has a first component of $\frac{1}{k}$ and its second component is no larger than $\frac{1}{k^4}$ (the last bin may contain a smaller number of first phase items). The first component of the sum of the vectors of these items is 1, and the second component is at most $0.3335 + 0.6 + \frac{1}{k^3} < 1$. The remaining first phase items (there are at most $N\Theta$ such items) are packed k in a bin. We find that $OPT(I_1^a) \leq \Gamma_a + \lceil \frac{N\Theta}{k} \rceil$.

For I_a^2, there are $\lceil \frac{N}{4k} \rceil$ bins, each with one item of the form $(0, 1-\alpha)$ and k small first phase items (recall that the number of small first phase items is larger than $\frac{N}{4} + k$), $\lceil \frac{Y_1^a + Y_2^a}{2} \rceil$ bins with at most two large second phase items and at most $k - 2a$ first phase items, $\lceil \frac{\Gamma_a + 2Y_2^a}{4} \rceil$ bins with one item of the form $(\frac{a}{k}, \frac{2}{3} - 2\delta')$, and at most one small second phase item, and at most $k - 2a$ first phase items. The remaining first phase items are packed into additional bins, such that every bin has k such items. All items are packed since the number of small second phase items, Y_2^a, is no larger than $\frac{\Gamma_a}{2}$, so $\frac{\Gamma_a + 2Y_2^a}{4} \geq Y_2^a$. The total space for first phase items in the first three kinds of bins is at least

$$\frac{N}{4} + (k-2a)\left(\frac{Y_1^a + Y_2^a}{2} + \frac{\Gamma_a + 2Y_2^a}{4}\right) = \frac{N}{4} + \frac{k-2a}{4} \cdot 3\Gamma_a$$

$$\geq \frac{N}{4} + \frac{3}{4}N(1-\Theta) = N - \frac{3}{4}N\Theta \ ,$$

so the number of bins of the last kind is at most $\lceil \frac{3N\Theta}{4k} \rceil \leq \frac{9N}{4k^2} + 1$ since $\Theta \leq \frac{3}{k}$. We find that

$$OPT(I_a^2) \leq \frac{N}{4k} + \frac{Y_1^a + Y_2^a}{2} + \frac{\Gamma_a + 2Y_2^a}{4} + \frac{9N}{4k^2} + 4 = \frac{N}{4k} + \frac{3\Gamma_a}{4} + \frac{9N}{4k^2} + 4 \ . \quad \blacktriangleleft$$

We get

$$R \geq \frac{ALG(I_1^a)}{OPT(I_1^a) - 2} \geq \frac{\Delta_{k-a+1} + Y_2^a + \frac{N - N\Theta}{k-2a}}{\frac{N-N\Theta}{k-2a} + \frac{N\Theta}{k}} \ ,$$

$$R \geq \frac{ALG(I_a^2)}{OPT(I_2^a) - 5} \geq \frac{\Delta_{k-a+1} + Y_1^a + Y_2^a + \frac{\frac{N-N\Theta}{k-2a} + 2Y_2^a}{4} + \frac{N}{4k}}{\frac{N}{4k} + \frac{3\frac{N-N\Theta}{k-2a}}{4} + \frac{9N}{4k^2}} \ .$$

We let $\beta = b/k$ and let k grows to infinity. Choosing $\beta \approx 0.192806$ and using the inequalities we showed, we find $R \geq 2.03731129$, and thus we conclude the following theorem.

▶ **Theorem 11.** *The asymptotic competitive ratio of any online algorithm for vector packing with $d \geq 2$ is at least* 2.03731129.

References

1 Y. Azar, I. R. Cohen, A. Fiat, and A. Roytman. Packing small vectors. In *Proc. of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA'16)*, pages 1511–1525, 2016.

2 Y. Azar, I. R. Cohen, S. Kamara, and F. B. Shepherd. Tight bounds for online vector bin packing. In *Proc. of the 45th ACM Symposium on Theory of Computing (STOC'13)*, pages 961–970, 2013.

3 Y. Azar, I. R. Cohen, and A. Roytman. Online lower bounds via duality. In *Proc. of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA'17)*, pages 1038–1050, 2017.

4 L. Babel, B. Chen, H. Kellerer, and V. Kotov. Algorithms for on-line bin-packing problems with cardinality constraints. *Discrete Applied Mathematics*, 143(1-3):238–251, 2004.

5 J. Balogh, J. Békési, and G. Galambos. New lower bounds for certain bin packing algorithms. *Theoretical Computer Science*, 1:1–13, 2012.

6 J. Békési, Gy. Dósa, and L. Epstein. Bounds for online bin packing with cardinality constraints. *Information and Computation*, 249:190–204, 2016.

7 David Blitz. Lower bounds on the asymptotic worst-case ratios of on-line bin packing algorithms. Technical Report 114682, University of Rotterdam, 1996. M.Sc. thesis.

8 David Blitz, Andre van Vliet, and Gerhard J. Woeginger. Lower bounds on the asymptotic worst-case ratio of online bin packing algorithms. Unpublished manuscript, 1996.

9 A. Caprara, H. Kellerer, and U. Pferschy. Approximation schemes for ordered vector packing problems. *Naval Research Logistics*, 92:58–69, 2003.

10 L. Epstein. Online bin packing with cardinality constraints. *SIAM Journal on Discrete Mathematics*, 20(4):1015–1030, 2006.

11 L. Epstein and A. Levin. AFPTAS results for common variants of bin packing: A new method for handling the small items. *SIAM Journal on Optimization*, 20(6):3121–3145, 2010.

12 H. Fujiwara and K. M. Kobayashi. Improved lower bounds for the online bin packing problem with cardinality constraints. *Journal of Combinatorial Optimization*, 29(1):67–87, 2015.

13 G. Galambos, H. Kellerer, and G. J. Woeginger. A lower bound for online vector packing algorithms. *Acta Cybernetica*, 10:23–34, 1994.

14 M. R. Garey, R. L. Graham, D. S. Johnson, and A. C. C. Yao. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory Series A*, 21(3):257–298, 1976.

15 K. Jansen, M. Maack, and M. Rau. Approximation schemes for machine scheduling with resource (in-)dependent processing times. In *Proc. of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA'16)*, pages 1526–1542, 2016.

16 D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3:256–278, 1974.

17 H. Kellerer and U. Pferschy. Cardinality constrained bin-packing problems. *Annals of Operations Research*, 92:335–348, 1999.

18 K. L. Krause, V. Y. Shen, and H. D. Schwetman. Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *Journal of the ACM*, 22(4):522–550, 1975.

19 K. L. Krause, V. Y. Shen, and H. D. Schwetman. Errata: "Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems". *Journal of the ACM*, 24(3):527–527, 1977.

20 C. C. Lee and D. T. Lee. A simple online bin packing algorithm. *Journal of the ACM*, 32(3):562–572, 1985.

21 P. Ramanan, D. J. Brown, C. C. Lee, and D. T. Lee. Online bin packing in linear time. *Journal of Algorithms*, 10:305–326, 1989.

22 S. S. Seiden. On the online bin packing problem. *Journal of the ACM*, 49(5):640–671, 2002.

23 A. van Vliet. An improved lower bound for online bin packing algorithms. *Information Processing Letters*, 43(5):277–284, 1992.

24 A. C. C. Yao. New algorithms for bin packing. *Journal of the ACM*, 27:207–227, 1980.

Modeling and Engineering Constrained Shortest Path Algorithms for Battery Electric Vehicles

Moritz Baum[1], Julian Dibbelt[2], Dorothea Wagner[3], and Tobias Zündorf[*4]

1 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
`moritz.baum@kit.edu@kit.edu`
2 **Mountain View, CA, USA**
`algo@dibbelt.de`
3 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
`dorothea.wagner@kit.edu`
4 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
`tobias.zuendorf@kit.edu`

Abstract

We study the problem of computing constrained shortest paths for battery electric vehicles. Since battery capacities are limited, fastest routes are often infeasible. Instead, users are interested in fast routes where the energy consumption does not exceed the battery capacity. For that, drivers can deliberately reduce speed to save energy. Hence, route planning should provide both path *and* speed recommendations. To tackle the resulting $\mathcal{NP}$-hard optimization problem, previous work trades correctness or accuracy of the underlying model for practical running times. In this work, we present a novel framework to compute *optimal* constrained shortest paths for electric vehicles that uses more realistic physical models, while taking speed adaptation into account. Careful algorithm engineering makes the approach practical even on large, realistic road networks: We compute optimal solutions in less than a second for typical battery capacities, matching performance of previous inexact methods. For even faster performance, the approach can easily be extended with heuristics that provide high quality solutions within milliseconds.

1998 ACM Subject Classification G.2.2 Graph Theory, G.2.3 Applications

Keywords and phrases electric vehicles, constrained shortest paths, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.11

1 Introduction

Battery electric vehicles (EVs) have matured, giving the prospect of high powertrain efficiency and independence of fossil fuels, but a major hindrance of their adoption remains the limited battery capacity of most vehicles combined with a lengthy recharge time. To overcome *range anxiety*, careful route planning that prevents battery depletion during a ride is paramount. Besides a limited cruising range, another substantial difference to vehicles run by combustion engines is the ability to recuperate energy when braking. Naturally, such aspects have to be reflected in any kind of route planning application for EVs.

Classic route planning approaches make use of a graph-based representation of the considered transportation network, where scalar edge weights correspond to, e. g., travel times. A shortest path is then found by Dijkstra's algorithm [15]. A wide range of *speedup techniques* [3]

* Supported by DFG Research Grant WA 654/23-1.

enable provably correct but faster queries in practice. For instance, A* Search [27] uses *vertex potentials* to guide the search towards the target. Contraction Hierarchies (CH) [23], on the other hand, employs a preprocessing step to obtain a directed acyclic search graph that allows to skip vast parts of the network at query time. For that, it iteratively *contracts* vertices according to a heuristic vertex ranking, while adding *shortcut* edges to maintain distances within the remaining graph. Extensions to multicriteria scenarios exist for both A* [17, 32, 33, 36] and CH [21, 22, 38]. Moreover, CH and A* can be combined to Core-ALT [6], where all but the highest-ranked vertices are contracted, which form the *core* graph. On that, a variant of A* uses precomputed distances to *landmark* vertices [24].

Route planning for EVs requires handling battery capacity constraints and negative edge weights (due to recuperation), which is tractable when optimizing energy consumption as a single criterion [9, 16, 35]. However, energy-optimal routes often exhibit disproportionate detours, as using minor, slow roads can save energy due to less air drag [9]. Variants of the $\mathcal{NP}$-hard *Constrained Shortest Path (CSP)* problem [26] overcome this by minimizing energy consumption without exceeding a given time limit [37] or finding the fastest route that does not exceed battery constraints [7, 41]. Yet, time–consumption tradeoffs are not only affected by choice of route but also by driving behavior. Assuming a single, fixed speed per road segment neglects attractive solutions that may still use major roads (e. g., motorways), saving energy by deliberately driving below posted speed limits, instead. Sampling such alternative speeds, tradeoffs can be modeled by parallel edges [8, 25], but this yields too many nondominated intermediate solutions, growing exponentially even for chains of vertices. Accordingly, only heuristics offer acceptable performance for common vehicle ranges [8, 25]. By discretizing a continuous range of possible speeds, the approach has further undesirable effects: The majority of its many intermediate solutions offers insignificant tradeoffs [8], while interesting solutions are lost to the discretization; adding degree-two vertices (commonly included for visualization) affects the solution space, even when distributing speeds and consumption evenly. Instead, Hartmann and Funke [28] model tradeoffs as continuous *functions* per edge, assuming the driver can go at *any* speed within limits. Yet, for that model they propose only a heuristic extension of CH that requires minutes to answer queries on large networks. Lv et al. [31] use dynamic programming to plan the speed of a solar-powered EV, but their approach aims at simulation and is too slow for interactive applications.

Contribution and Outline. We study a generalization of the CSP problem to capture the characteristics of EVs, considering *continuous, adaptive speeds*: We allow the EV to adjust its speed to reach its target quickly and with sufficient state of charge (SoC). Using realistic consumption models, we obtain for each road segment a function mapping travel time to energy consumption, yielding a challenging, more precise problem setting (Section 2). As a first solution, we propose an exponential-time extension of Dijkstra's algorithm: By propagating continuous consumption functions during network exploration, we greatly improve performance *and* solution quality over previous discretized approaches (Section 3). We also incorporate techniques based on A* and CH, for which a particular challenge is the computation of shortcuts that represent *bivariate functions* to capture the constraints of our model (Section 4). Our experimental evaluation (Section 5) reveals that we can compute *optimal* solutions in well below a second for typical battery capacities and less than a minute for large battery capacities, on par or faster than previous *heuristic* algorithms. Our own heuristic variant provides high-quality solutions and is fast enough for interactive applications.

2 Model and Problem Statement

We use directed graphs $G = (V, E)$ to model road networks, where edges $e \in E$ represent road segments. For each, we assume that a given *tradeoff function* $g_e \colon \mathbb{R}_{>0} \to \mathbb{R}$ maps desired driving time $x \in \mathbb{R}_{>0}$ along e to energy consumption $g_e(x)$. Consumption can be negative, due to recuperation. In reality, driving time cannot be chosen arbitrarily: Lower bounds are induced by speed limits and the vehicle's maximum speed. On the other hand, driving slower than a reasonable minimum speed would mean to become an obstacle for other drivers. This yields minimum and maximum driving times $\underline{\tau} \in \mathbb{R}_{>0}$ and $\bar{\tau} \in \mathbb{R}_{>0}$, respectively, for g_e. We incorporate them into a *consumption function* $c_e \colon \mathbb{R}_{\geq 0} \to \mathbb{R} \cup \{\infty\}$ with

$$c_e(x) := \begin{cases} \infty & \text{if } x < \underline{\tau}, \\ g_e(\bar{\tau}) & \text{if } x > \bar{\tau}, \\ g_e(x) & \text{otherwise.} \end{cases}$$

Thus, driving times below $\underline{\tau}$ are infeasible (modeled as infinite consumption) and driving times above $\bar{\tau}$ become unprofitable. In the special (degenerate) case of $\underline{\tau} = \bar{\tau}$, the function c_e represents a constant pair $(\underline{\tau}, c_e(\underline{\tau}))$ of driving time and consumption. We then call c_e *constant*, as the edge e allows no speed adaptation.

Further, the EV is equipped with a battery that has a *capacity* $M \in \mathbb{R}_{\geq 0}$. The SoC must not drop below 0 nor exceed M. Incorporating these constraints, we obtain a bivariate *SoC function* $f_e \colon \mathbb{R}_{\geq 0} \times [0, M] \cup \{-\infty\} \to [0, M] \cup \{-\infty\}$ for every $e = (u, v) \in E$, mapping SoC at u to SoC at v when traversing e with a specific driving time. It is given by

$$f_e(x, b) := \begin{cases} -\infty & \text{if } b - c_e(x) < 0, \\ M & \text{if } b - c_e(x) > M, \\ b - c_e(x) & \text{otherwise,} \end{cases}$$

where an SoC of $-\infty$ denotes an empty battery. Hence, $f_e(x, b) = -\infty$ means that the edge cannot be traversed at the corresponding speed (as the battery would run empty).

An *s–t path* in G is a sequence $P = [s = v_1, v_2 \ldots, v_k = t]$ of vertices with $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. If $s = t$, we call P a *cycle*. Given the SoC $b_s \in [0, M]$ at s, we obtain a corresponding SoC at t by iteratively picking driving times $x_i \in \mathbb{R}_{\geq 0}$ (starting at s) and evaluating the SoC function $f_{(v_i, v_{i+1})}$ for x_i and the SoC at v_i. Due to physical constraints, we presume that for cycles this procedure never increases the SoC at $s = t$. For paths $P = [v_1, \ldots v_i]$ and $Q = [v_i, \ldots, v_k]$, $P \circ Q := [v_1, \ldots, v_i, \ldots, v_k]$ is their concatenation.

Given a source $s \in V$, a target $t \in V$, and an initial SoC $b_s \in [0, M]$, the *Electric Vehicle Constrained Shortest Path (EVCSP)* Problem is to find an s–t path $P = [s = v_1, v_2 \ldots, v_k = t]$ together with driving times $x_i, i \in \{1, \ldots, k - 1\}$, for every edge in P that respect battery constraints and minimize overall travel time $x := \sum_{i=1}^{k-1} x_i$ in G. This yields an $\mathcal{NP}$-hard problem by reduction from CSP [26]. An instance of CSP corresponds to an instance of EVCSP where all functions are degenerate constant tuples with nonnegative consumption.

A Simplified Model. We illustrate SoC functions in an example using simplistic but vivid tradeoff functions. For now, let tradeoff functions be *decreasing* and *linear*, i. e., $g_e(x) = \alpha x + \beta$ for every $e \in E$, where $\alpha \in \mathbb{R}_{\leq 0}$ and $\beta \in \mathbb{R}$ are constant coefficients. The values α and β may differ between edges to reflect different road types or other relevant factors [12, 42]. Figures 1a and 1b show consumption functions (plugging in limits $\underline{\tau}$ and $\bar{\tau}$ on driving time) for two edges (u, v) and (v, w). We are interested in the consumption function of the path

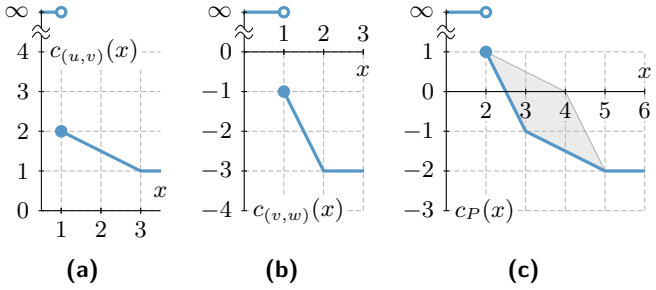


Figure 1 Consumption functions based on a simple model. (a) Function $c_{(u,v)}$ of an edge (u, v) with $\underline{\tau} = 1$ and $\bar{\tau} = 3$. (b) Function $c_{(v,w)}$ of an edge (v, w) with $\underline{\tau} = 1$ and $\bar{\tau} = 2$. (c) The function c_P of $P = [u, v, w]$. The shaded area indicates possible pairs of driving time and consumption.

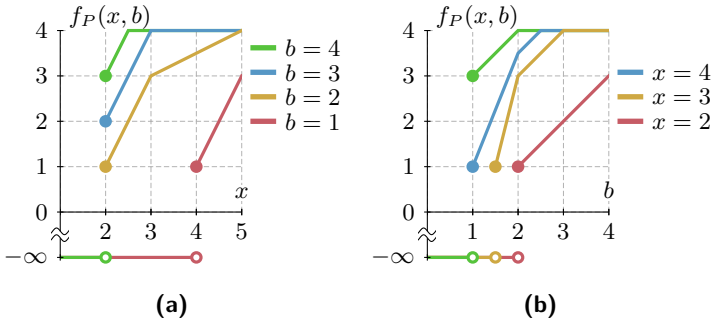


Figure 2 The bivariate SoC function of the path P from Figure 1, for $M = 4$. (a) The SoC f_P at v, subject to driving time x on P for different fixed values b of initial SoC. (b) The SoC f_P at v, subject to initial SoC b for different fixed values x of driving time.

$P = [u, v, w]$, i. e., a function c_P that maps driving time x spent on P to *minimum* energy consumption $c_P(x)$. Formally, to get $c_P(x)$ for a driving time $x \in \mathbb{R}_{\geq 0}$, we must pick values $x_1 \in \mathbb{R}_{\geq 0}$ and $x_2 \in \mathbb{R}_{\geq 0}$, such that $x = x_1 + x_2$ and $c_{(u,v)}(x_1) + c_{(v,w)}(x_2)$ is minimized. Figure 1c shows possible distributions of driving times among the two edges and the resulting energy consumption. Their lower envelope yields the desired function c_P. Intuitively, we want to spend as much of the available time as possible on the edge that provides the better tradeoff for saving the most energy, i. e., the function with steeper slope. As a result, the consumption function of a path is always *convex* on its finite imaginary part. Moreover, while tradeoff functions of edges are linear in the interval $[\underline{\tau}, \bar{\tau}]$ of admissible driving times, the tradeoff function of a path is *piecewise* linear within its corresponding interval.

When considering battery constraints, energy consumption depends not only on driving time but also on initial SoC. Note that consumption is positive on (u, v) and negative on (v, w). As before, the edge (v, w) provides the better tradeoff. However, for low initial SoC, we must ensure that (u, v) can be traversed first, spending additional time on this edge in order to obtain a feasible solution at all. In contrast, high initial SoC values may prevent recuperation along (v, w), limiting the payoff of driving slower. Figure 2 illustrates the resulting *bivariate* SoC function f_P for specific values of initial SoC and driving time.

A Realistic Model. In this work, we use a more realistic model, detailed below. Both driving time and energy consumption depend on the vehicle's speed. In accordance with realistic physical models [1, 2, 10, 18, 28, 30, 31], we assume that energy consumption on a

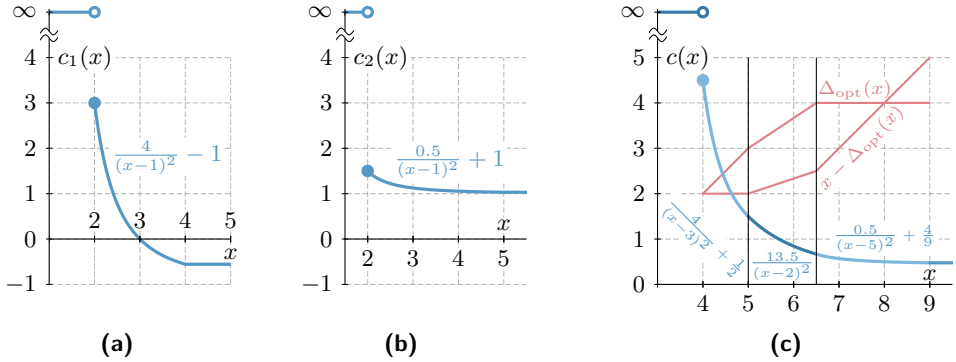


Figure 3 Linking consumption functions. (a) Function c_1 with $\alpha_1 = 4$, $\beta_1 = 1$, $\gamma_1 = -1$, $\underline{\tau} = 2$, and $\bar{\tau}_1 = 4$. (b) Function c_2 with $\alpha_1 = 0.5$, $\beta_1 = 1$, $\gamma_1 = 1$, $\underline{\tau}_1 = 2$, and $\bar{\tau}_1 = 5$. (c) Function $c = \text{link}(c_1 c_2)$, with $c(x) = c_1(\Delta_{\text{opt}}(x)) + c_2(x - \Delta_{\text{opt}}(x))$. It is defined by three subfunctions with subdomains $[4, 5]$, $[5, 6.5]$, $[6.5, 9]$. Values $\Delta_{\text{opt}}(x)$ and $x - \Delta_{\text{opt}}(x)$ indicate the share of c_1 and c_2.

road segment $e \in E$ is expressed by a function $h_e \colon \mathbb{R}_{>0} \to \mathbb{R}$ with $h_e(v) = \lambda_1 v^2 + \lambda_2 s_e + \lambda_3$, where $v \in \mathbb{R}_{>0}$ is the (constant) vehicle speed, $s_e \in \mathbb{R}$ is the (constant) slope of the road segment, and $\lambda_1 \in \mathbb{R}_{\geq 0}$, $\lambda_2 \in \mathbb{R}_{\geq 0}$, and $\lambda_3 \in \mathbb{R}_{\geq 0}$ are constant nonnegative coefficients of the consumption model (all values may vary for different edges). Note that assuming constant speed and slope per edge is not a restriction, as intermediate vertices can be added to model changing conditions. Further, one can show that varying the speed on a single road segment (with constant slope and speed limit) never pays off in our model [28, Corollary 1].

As we are interested in functions mapping *driving time* $x \in \mathbb{R}_{>0}$ to energy consumption $g_e(x)$, we substitute $v = \ell_e / x$, where ℓ_e is the length of the road segment. Slope and length of an edge are fixed, so we simplify this by setting $\alpha := \lambda_1 / \ell_e^2$ and $\gamma := \lambda_2 s_e + \lambda_3$. Observe that $\alpha \in \mathbb{R}_{\geq 0}$ is nonnegative, while $\gamma \in \mathbb{R}$ may be negative (for downhill edges). We introduce a third constant $\beta \in \mathbb{R}_{\geq 0}$, needed later to shift functions along the time axis. Altogether, we obtain the tradeoff function $g_e \colon \mathbb{R}_{>0} \to \mathbb{R}$ with

$$g_e(x) := \frac{\alpha}{(x - \beta)^2} + \gamma. \tag{1}$$

For single edges, we always obtain $\beta = 0$ and assume driving time x to be *strictly* positive. Thus, the denominator $x - \beta$ is strictly positive and $g_e(x)$ is finite. Further, g_e is *decreasing* and *convex* on $\mathbb{R}_{>0}$ in this case. In the simplistic model discussed above, we have seen that tradeoff functions of *paths* may be piecewise linear. Similarly, we allow tradeoff functions in the realistic model to be defined *piecewise*, so they may consist of multiple subfunctions of the form in Equation 1. Tradeoff functions of paths may also use values $0 < \beta < x$ to reflect additional time spent on previous edges. Plugging in the values $\underline{\tau} \in \mathbb{R}_{>0}$ and $\bar{\tau} \in \mathbb{R}_{>0}$, we obtain the *consumption function* $c_e \colon \mathbb{R}_{\geq 0} \to \mathbb{R} \cup \{\infty\}$.

3 Basic Approach

We generalize the (exponential-time) bicriteria variant [34] of Dijkstra's algorithm [15] to solve EVCSP. As a crucial ingredient, the algorithm requires a *link operation*: For two consumption functions c_1 and c_2 modeling consumption on two paths P_1 and P_2, the function $c := \text{link}(c_1, c_2)$ maps driving time spent on $P := P_1 \circ P_2$ to *minimum* possible energy consumption (bar battery constraints). Let $\underline{\tau}_1$, $\bar{\tau}_1$, $\underline{\tau}_2$, and $\bar{\tau}_2$ denote the respective minimum

and maximum driving times of c_1 and c_2. We obtain $c(x) = \infty$ for all $x < \underline{\tau}_1 + \underline{\tau}_2$ and $c(x) = c_1(\bar{\tau}_1) + c_2(\bar{\tau}_2)$ for $x > \bar{\tau}_1 + \bar{\tau}_2$. For all $x \in [\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2]$, we have to compute

$$c(x) = \min_{\substack{\Delta \in [\underline{\tau}_1, \bar{\tau}_1] \\ \Delta \in [x - \bar{\tau}_2, x - \underline{\tau}_2]}} c_1(\Delta) + c_2(x - \Delta).$$

In other words, we have to divide the amount of time that exceeds the minimum possible total driving time among the two paths such that consumption is minimized; see Figure 3 for an example. Although realistic functions require a more technical analysis, many observations made for our simplistic (linear) model from the previous section carry over to the more realistic (nonlinear) tradeoff functions. In fact, the function c can be computed in linear time in the number of subfunctions defining c_1 and c_2.

Algorithm Description. Given a source $s \in V$, a target $t \in V$, and initial SoC $b_s \in [0, M]$, the *tradeoff function propagating (TFP)* algorithm solves EVCSP. It propagates labels consisting of consumption functions (defined piecewise, by sequences of tradeoff functions) and applies battery constraints on-the-fly. Hence, it does not have to maintain bivariate SoC functions explicitly. The algorithm starts with the constant label $c_s \equiv M - b_s$ at s. The label is also added to a priority queue, which uses minimum driving time of a label as key. In each step of its main loop, the algorithm extracts and *settles* a label c_u (at some vertex $u \in V$) with minimum key from the queue. For every edge $(u, v) \in E$, the function $c := \text{link}(c_u, c_{(u,v)})$ is computed. Note that c may violate battery constraints, so we set $c(x) := \infty$ for all $x \in \mathbb{R}_{\geq 0}$ with $c(x) > M$ and $c(x) := 0$ for all $x \in \mathbb{R}_{\geq 0}$ with $c(x) < 0$. The resulting function is added to the priority queue, unless it is *dominated* by existing labels at v; we say that a label c_1 dominates another label c_2 if $c_1(x) \leq c_2(x)$ for all $x \in \mathbb{R}_{\geq 0}$.

To keep the number of label comparisons low, each vertex $v \in V$ maintains a set $L_{\text{set}}(v)$ and a heap $L_{\text{uns}}(v)$ containing its *settled* and *unsettled* labels, respectively. We maintain the invariant that for each $v \in V$, the unsettled label in $L_{\text{uns}}(v)$ with *minimum* key is not dominated by any settled label in $L_{\text{set}}(v)$. Labels (at v) added to the priority queue are also pushed into $L_{\text{uns}}(v)$. Every time the minimum element of $L_{\text{uns}}(v)$ changes (because an element is added or extracted), we check whether the new minimum element is dominated by any settled label in $L_{\text{set}}(v)$ and discard it in this case [7]. Dominance is tested as follows. For two *subfunctions* (with the form of Equation 1), we can test in constant time whether one dominates the other (by evaluating extreme points of their difference and subdomain borders). For piecewise-defined consumption functions, we exploit that we only need to compare subfunctions whose subdomains intersect. This allows us to test for dominance in a linear scan (comparing subfunctions in increasing order of driving time). Given a consumption function c in the set $L_{\text{uns}}(v)$ of some vertex $v \in V$, a naïve implementation then performs pairwise comparisons to functions in $L_{\text{set}}(v)$ to determine whether c is dominated by any of them. In doing so, the algorithm may miss cases where c is merely *partially* dominated, or dominated only by the lower envelope of *several* functions. Although including dominated labels in $L_{\text{set}}(v)$ does not affect correctness, it may lead to unnecessary vertex scans and increases the label size. Instead of pairwise dominance checks, we therefore identify dominated parts of c in a single coordinated scan over c and *all* functions in $L_{\text{set}}(v)$.

TFP is *label setting*, i. e., labels extracted from the queue are never dominated later on. An optimal (constrained) path is found once a label at t is extracted, which gives the optimal driving time. It is also possible to retrieve the optimal path and driving speeds.

A Polynomial-Time Heuristic. To improve running times, TFP can easily be extended to a heuristic search, at the cost of inexact results. We propose a polynomial-time approach based on ε-dominance [4]. When testing dominance of a label $c \in L_{\text{uns}}(v)$ at some vertex $v \in V$,

it is kept in $L_{\text{uns}}(v)$ only if it yields an improvement (over labels in $L_{\text{set}}(v)$) by at least a certain fraction εM, with $\varepsilon \in (0, 1]$, for some driving time. Hence, we test for every $x \in \mathbb{R}_{\geq 0}$ whether $c(x) + \varepsilon M \leq c_{\text{set}}(x)$ holds for *all* settled functions $c_{\text{set}} \in L_{\text{set}}(v)$. Then, the number of settled labels per set can become at most $\lceil 1/\varepsilon \rceil$, which yields polynomial running time.

4 Speedup Techniques

We propose speedup techniques based on A* and CH for TFP (and its heuristic variant). Combining both techniques, we obtain our fastest variant, *CHAsp (CH, A*, Adaptive Speeds)*. Our techniques do not alter the output of the algorithm, so correctness of TFP is maintained.

A* Search. This well-known technique [27, 33] uses a *potential function* $\pi\colon V \to \mathbb{R}_{\geq 0}$. The potential $\pi(v)$ of a vertex $v \in V$ is added to all keys of labels when running TFP, so labels are extracted in a different order. We compute the potential function at query time.

Our first variant uses a cost functions $\underline{d}\colon E \to \mathbb{R}_{\geq 0}$ with $\underline{d}(e) = c_e(\underline{\tau}_e)$, i. e., minimum driving time on an edge. Before running TFP, a *backward* search (i. e., Dijkstra's algorithm traversing edges in backward direction) from the target t computes, for each vertex $v \in V$, the minimum *unconstrained* driving time $\underline{d}(v, t)$ from v to the t. We obtain a consistent potential function $\pi_d\colon V \to \mathbb{R}_{\geq 0}$ by setting $\pi_d(v) := \underline{d}(v, t)$ [40]. Similarly, we compute lower bounds on energy consumption, which allow us to prune the TFP search [8].

The potential function $\pi_d(v)$ may be too conservative if consumption on the optimal path is very high. In such cases, it pays off to use a potential function $\pi_f\colon V \times [0, M] \to \mathbb{R}_{\geq 0}$ that incorporates current SoC at a vertex [7]. We represent $\pi_f(v, b)$ with a convex, piecewise linear function that maps SoC $b \in [0, M]$ at a vertex $v \in V$ to a lower bound on remaining driving time. The functions are determined in a label-correcting backward search from t.

Contraction Hierarchies. We propose an adaptation of CH to our scenario, which adds a preprocessing step for faster queries. As in plain CH [23], vertices are contracted iteratively (ordered by heuristic *rank*) during preprocessing and *shortcut edges* are added to maintain distances. However, we contract only a subset of the vertices, leaving an uncontracted *core* graph – a common approach in complex scenarios [7, 14, 28, 37]. Since the SoC at a vertex $u \in V$ is only known at query time in our setting, any shortcut (u, v) has to store a bivariate SoC function $f_{(u,v)}$. Figure 4 illustrates how the initial SoC influences energy consumption in our model. Their bivariate nature makes explicit construction and comparison of SoC functions rather challenging. We discuss simple representations of SoC functions in certain cases, exploiting that most consumption values are positive in realistic instances. We say that a path P is *discharging* if the SoC on P never exceeds the (arbitrary) initial SoC, i. e., there is no prefix of P that has negative minimum consumption for arbitrary driving times (subpaths with negative consumption are allowed, though). Hence, it is not necessary to explicitly check whether the SoC exceeds M on a discharging path. We show how the SoC function of a discharging path is represented by at most two consumption functions.

As a first example, assume we are given a path $P = P_1 \circ P_2$ consisting of two subpaths P_1 and P_2 with respective consumption functions c_1 and c_2, as in Figure 4. Let $\underline{\tau}_1, \bar{\tau}_1, \underline{\tau}_2, \bar{\tau}_2$ denote their corresponding minimum and maximum driving times. Assume that $c_1(x) > 0$ is *positive* for all $x \in \mathbb{R}_{\geq 0}$, while $c_2(x) \leq 0$ is *nonpositive* for all $x \in [\underline{\tau}_2, \infty)$. Finally, assume that $|c_1(\bar{\tau}_1)| \geq |c_2(\bar{\tau}_2)|$, i. e., the cost of P_1 is higher than the gain of P_2 for *any* driving time, so P is discharging. To derive the SoC function of P we introduce two auxiliary functions: a *positive part* c^+ with $c^+(x) := c_1(x - \underline{\tau}_2)$, and a *negative part* c^- with $c^-(x) := c_2(x + \underline{\tau}_2)$.

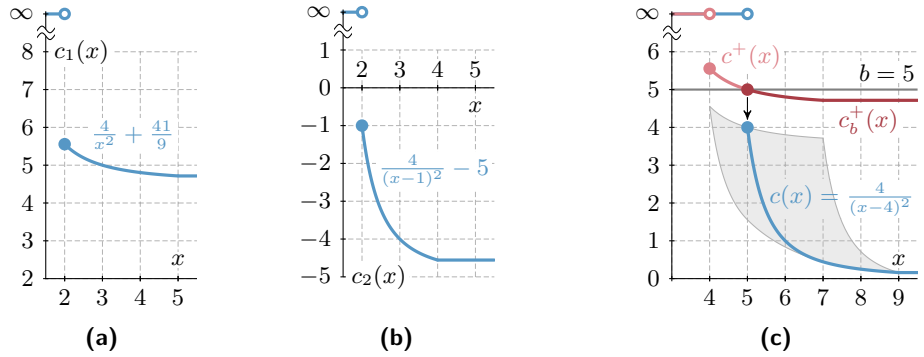


Figure 4 Constructing a consumption function depending on initial SoC. (a) Function c_1 of a path P_1. (b) Function c_2 of a path P_2. (c) Due to battery constraints, the minimum driving time on $P = P_1 \circ P_2$ is 5 for an initial SoC $b = 5$. This yields the consumption function $c = \text{link}(c_b^+, c^-)$. The shaded area indicates possible values of consumption functions for different values of initial SoC.

The original functions are shifted along the x-axis to simplify the analysis (note that the minimum feasible driving time of c^- is 0). Given some initial SoC $b \in [0, M]$, the positive part c^+, and the negative part c^-, we first define the *constrained positive part* c_b^+ as

$$c_b^+(x) := \begin{cases} \infty & \text{if } b < c^+(x) \\ c^+(x) & \text{otherwise,} \end{cases}$$

which applies battery constraints along P_1 for an initial SoC of b; see Figure 4. Then, the SoC function f_P of the path P evaluates to $f_P(x, b) = b - \text{link}(c_b^+, c^-)(x)$ for arbitrary $x \in \mathbb{R}_{\geq 0}$ and $b \in [0, M]$. The function first applies battery constraints on the positive part and links the resulting function with the negative part.

We now describe how SoC functions representing general discharging paths are constructed from two given SoC functions of discharging paths. Assume we are given a discharging path P_1 whose SoC function is defined by two consumption functions c_1^+ and c_1^-, as described above. Similarly, we are given a discharging path P_2 with respective consumption functions c_2^+ and c_2^-. Observe that the path $P := P_1 \circ P_2$ must be discharging as well. Apparently, if we know the initial SoC, we can compute energy consumption on P by computing $\text{link}(\text{link}(\text{link}(c_1^+, c_1^-)c_2^+)c_2^-)$ and applying battery constraints *before* each link operation, like in the TFP algorithm. However, we want to represent P with only two consumption functions c^+ and c^-. Recall that the only constraint we have to check for discharging paths is whether the SoC drops below 0. Thus, we identify a new positive part c^+ as follows. Since both c_1^- and c_2^- are nonpositive for all admissible driving times, the constraint needs only to be checked for c_1^+ and c_2^+ (i. e., before the first and third link operation). To integrate these checks into a single positive part c^+, we first compute the function $h := \text{link}(c_1^-, c_2^+)$. Clearly, the battery can only run empty on P_2 if this consumption function is positive for some admissible driving time. To distinguish this case, we split h into a positive part h^+ with $h^+(x) := \max\{h(x), 0\}$ and a negative part h^- with $h^-(x) := h(x)$ if $h(x) \leq 0$ and $h^-(x) := \infty$ otherwise. Since h is a decreasing consumption function, so are h^+ and h^-. We obtain the positive part c^+ of P by setting $c^+(x) := \text{link}(c_1^+, h^+)(x - \underline{\tau})$ and the negative part c^- by setting $c^-(x) := \text{link}(h^-, c_2^-)(x + \underline{\tau})$, where $\underline{\tau}$ is the minimum driving time of h^-. The SoC function of P is obtained from c^+ and c^- as described above.

During preprocessing, we only allow a vertex $v \in V$ to be contracted if all new shortcuts created as part of its contraction are discharging. We call v *active* in this case. Note that the number of active vertices grows as contraction proceeds, as contraction produces longer

Table 1 Benefits of our approach (Eur-PG, 2 kWh). For TFP and TFP-dom. (improved dominance tests), we report the number of settled labels (# Lbls.), number of label comparisons during the forward search (# Dom.), average and maximum running times, and relative driving time savings over the constrained path found by BSP on discretized speeds.

	Query				Path Savings	
Algo.	# Lbls.	# Dom.	avg. [ms]	max. [ms]	avg. [%]	max. [%]
BSP	30 990 276	21 300 657 522	47 755	779 756	–	–
TFP	103 119	4 399 002	444	14 347	2.7 %	9.4 %
TFP-dom.	46 228	700 546	103	3 851	2.7 %	9.4 %

shortcuts, which are more likely to consist of long positive parts. Since we deal with a bicriteria scenario, vertex contraction may produce multi-edges. In such cases, we only want to keep shortcuts whose SoC functions are not dominated by parallel shortcuts. Hence, after contraction of a vertex, we delete (parts of) SoC functions of shortcut candidates that are dominated by existing functions between the same pair of vertices (and vice versa). To this end, we derive efficient dominance checks for (simple) bivariate SoC functions that can be performed in linear time (in the number of subfunctions of all involved functions). Finally, before adding a (nondominated) shortcut candidate to the graph, we run a witness search [23] to test if the shortcut is necessary to maintain distances. As an exact approach would require propagation and comparison of bivariate SoC functions, our witness search computes univariate *upper bounds* on energy consumption instead. This does not violate correctness, but may result in unnecessary shortcuts.

Queries. Plain CH uses a bidirectional search, which scans only edges to vertices of higher rank in the input graph enriched with shortcuts obtained during preprocessing. In our case, however, the SoC at the target vertex $t \in V$ is not known at query time, which makes backward search difficult. Instead, we extract the search space in a (backward) BFS from t, scanning and marking only edges to vertices of higher rank. Afterwards, we execute TFP from the source vertex s, scanning upward edges (with respect to ranks of incident vertices), core edges, and marked downward edges. For faster queries, we can combine this search with A*.

5 Experiments

We implemented all approaches in C++, using g++ 4.8.3 (-O3) as compiler. Experiments were conducted on a single core of a 4-core Intel Xeon E5-1630v3 clocked at 3.7 GHz, with 128 GiB of DDR4-2133 RAM, 10 MiB of L3 cache, and 256 KiB of L2 cache.

We consider road networks of Europe with 22 198 628 vertices and 51 088 095 edges and Germany with 4 692 091 vertices and 10 805 429 edges, provided by PTV AG (`http://ptvgroup.com`). Combining reasonable minimum speeds for different road types (e. g., 80 km/h on motorways and 30 km/h in residential areas) with the posted speed limits (if higher), we get intervals of allowed speeds per road segment, resulting in 25 % and 38 % of nonconstant edges for Germany and Europe, respectively. Applying elevation data from the Shuttle Radar Topography Mission, v4.1 (`srtm.csi.cgiar.org`), we derived realistic energy consumption from two detailed micro-scale emission models [29]: one based on a Peugeot iOn and one artificial model [39] that additionally accounts for auxiliary consumers (e. g., air conditioning). These data sources are proprietary, but enable evaluation on

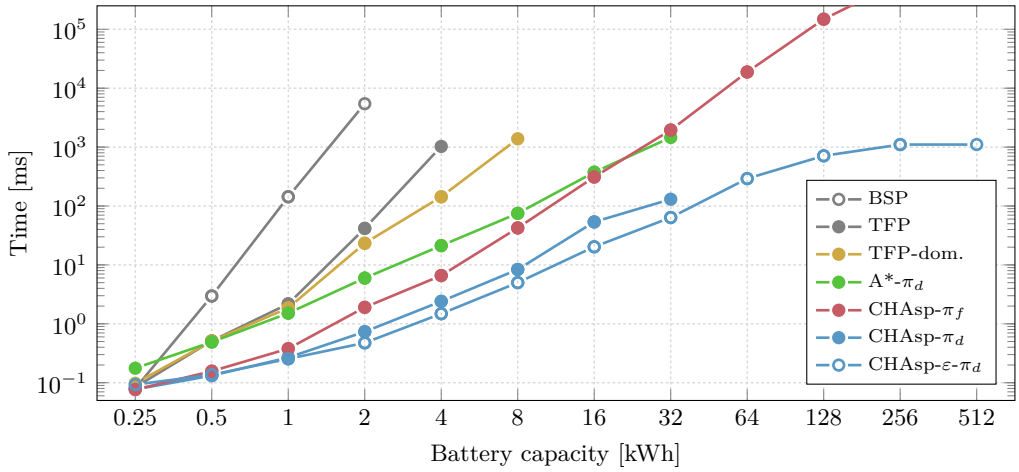


Figure 5 Scalability of BSP, our TFP algorithm, TFP with improved dominance tests (TFP-dom.), speedup techniques (A*-π_d, CHAsp-π_d, and CHAsp-π_f), and our heuristic approach CHAsp-ε-π_d with $\varepsilon := 0.1$. A capacity of 512 kWh corresponds to a range of roughly 3 000 km.

detailed and realistic input data. We denote our instances by Germany-Aux (Ger-AX), Germany-Peugeot (Ger-PG), Europe-Aux (Eur-AX), and Europe-Peugeot (Eur-PG). They have negative consumption (for at least some driving times) on 7.8 % (Ger-AX) to 12.9 % (Eur-PG) of their edges.

For comparison, we consider parallel edges and bicriteria shortest paths (BSP) [34] to model adaptive speeds, as was best practice in previous approaches [8]. We generate multi-edges by sampling consumption functions at discrete velocity steps of 10 km/h.

We evaluate random *in-range* queries, i. e., we pick a source vertex $s \in V$ uniformly at random. Among all vertices in range from s with an initial SoC $b_s = M$, we pick the target vertex $t \in V$ uniformly at random. Since unreachable targets are easily detected by backward search phases of A* (or any algorithm for computing energy-optimal routes [9, 16, 35]), this yields more challenging and interesting queries for us.

Model Validation and Scalability. We have argued that an approach based fully on consumption *functions* unlocks both better tractability and improved solution quality compared to discrete speeds and BSP. Indeed, we observe a significant speedup by simply switching to our more realistic model, as Table 1 shows. TFP is up to two orders of magnitudes faster than BSP and finds paths that are up to 9.4 % quicker (within SoC constraints), since it evaluates speed–consumption tradeoffs more fine-granularly while maintaining less query state (labels of continuous functions expressed by few parameters instead of large, discrete Pareto sets). This is interesting, as sampling was expressly considered to manage tractability [8, 25, 28]. In fact, even though atomic operations (linking and comparing labels) are more expensive for TFP, a drastic reduction in the number of vertex scans explains the speedup.

Figure 5 gives an overview of our approaches and their scalability across increasing battery capacities. For each capacity, we ran 100 random in-range queries, reporting median running time if all 100 queries terminated within one hour. Beyond the previously discussed BSP, TFP, and TFP-dom., A* enables reasonable running times for capacities of up to 32 kWh, without any preprocessing. Adding preprocessing, CHAsp-π_d provides further speedup by about an order of magnitude. In comparison, median running times of CHAsp-π_f are slower for all ranges up to 32 kWh. However, this algorithm is more robust against outliers and

Table 2 Impact of core size on performance (Ger-PG, 16 kWh). Vertex contraction stopped once the average degree of active vertices in the core reached a given threshold (Ø Deg). We report the resulting core size (# Vertices), preprocessing time, and average query times for 1 000 queries using CHAsp with potential functions π_d and π_f, respectively.

Core size		Prepr.	Query [ms]	
Ø Deg.	# Vertices	[h:m:s]	π_d	π_f
0	–	–	3 326.0	4 861.5
8	720 514 (15.36 %)	5:07	737.2	798.3
16	400 174 (8.53 %)	13:25	496.2	485.0
32	305 301 (6.51 %)	31:44	451.8	434.0
64	268 436 (5.72 %)	1:11:13	505.5	473.1
128	251 410 (5.36 %)	2:37:23	649.1	586.1

Table 3 Preprocessing and exact query performance for the potential functions π_d and π_f. For the ranges 16 kWh and 85 kWh, we show number of labels settled during the forward search (# Lbls.), number of label comparisons during the forward search (# Dom.) and total query times.

Prepro.			16 kWh			85 kWh		
Inst.	[h:m:s]	Algo.	# Lbls.	# Dom.	Query [ms]	# Lbls.	# Dom.	Query [ms]
Ger-AX	30:34	CHAsp-π_d	152	3 788	4.2	24 715	4 312 923	552.3
Ger-AX	30:34	CHAsp-π_f	61	448	17.0	406	11 813	1 236.7
Ger-PG	31:44	CHAsp-π_d	32 773	6 352 488	451.8	2 272 350	2 130 447 427	131 562.0
Ger-PG	31:44	CHAsp-π_f	6 008	491 173	434.0	32 182	6 836 380	14 873.5
Eur-AX	3:10:43	CHAsp-π_d	124	2 175	4.0	27 358	12 159 343	960.9
Eur-AX	3:10:43	CHAsp-π_f	73	1 006	15.8	871	46 529	1 174.7
Eur-PG	3:13:01	CHAsp-π_d	23 304	5 024 403	346.1	–	–	–
Eur-PG	3:13:01	CHAsp-π_f	6 629	800 430	341.7	105 792	44 986 403	34 617.4

is the only exact method that terminates within an hour for all queries at 64 kWh and up. Finally, our heuristic variant scales very well with vehicle range: Query times actually bottom out for large battery capacities, as the vehicle range gets close to the graph diameter.

Detailed Experiments. We evaluate different variants of our fastest approach, CHAsp. Table 2 shows CH preprocessing effort and query performance subject to core size on Ger-PG, for a common battery capacity of 16 kWh (corresponding to a range of 100 km). Contraction becomes much slower beyond a core degree of 32, which is explained by the small number of remaining active (i. e., contractable) vertices in the core. This also explains why speedup compared to the baseline (Ø deg = 0 is equivalent to plain TFP combined with A*) is much smaller than in simpler applications, where CH typically improves the baseline by several orders of magnitude [23]. Similar observations were made in other complex settings, including time-dependent [5, 11] and multicriteria [21, 22] scenarios. Nevertheless, CH still yields an improvement by up to an order of magnitude in our case. In our subsequent experiments, we pick an average core degree of 32 as stopping criterion of CH preprocessing.

Table 3 reports performance of CHAsp on all instances for capacities of 16 kWh and 85 kWh (as in Tesla models, with a range of 400–500 km). Figures are average values for 1 000 in-range queries. For 16 kWh, our techniques find the optimal solution in well below a second on average. For Ger-AX and Eur-AX, we even achieve query times in the order of milliseconds.

Table 4 Performance of the heuristic variant of CHAsp-π_d, for different choices of the parameter ε (see Section 3) on the hard instances Ger-PG and Eur-PG. We show figures on query performance for 1 000 random queries with a range of 16 kWh, as in Table 3. Additionally, we report the percentage of feasible and optimal results, as well as the average and maximum relative error.

			Query			Result Quality			
Inst.	Prepro.	ε	# Lbls.	# Dom.	T. [ms]	Feas.	Opt.	Avg.	Max.
Ger-PG	31:43	0.00	32 773	6 352 488	451.8	100.0 %	100.0 %	1.0000	1.0000
	30:41	0.01	19 922	1 949 458	225.6	100.0 %	89.4 %	1.0001	1.0047
	25:49	0.10	6 891	208 058	75.6	98.9 %	62.8 %	1.0013	1.0502
	17:48	1.00	1 742	11 149	30.7	95.1 %	47.6 %	1.0144	1.2294
Eur-PG	3:09:22	0.00	23 304	5 024 403	346.1	100.0 %	100.0 %	1.0000	1.0000
	3:04:48	0.01	12 803	1 132 685	151.6	100.0 %	82.8 %	1.0001	1.0145
	2:47:09	0.10	5 045	126 662	60.9	99.5 %	57.5 %	1.0020	1.0418
	2:14:03	1.00	1 428	7 641	28.2	92.7 %	45.8 %	1.0203	1.3960

This gap in running time is explained by the difference in the number of edges with negative cost, caused by the underlying consumption model. One could even argue that the instances Ger-PG and Eur-PG are actually rather excessive in this regard, by not accounting for any auxiliary consumers at all. As a result, these instances are significantly more difficult to solve. Regarding the potential functions π_d and π_f, the search space is consistently smaller when using π_f, but the backward search is more expensive. In fact, it becomes the major bottleneck for a battery capacity of 16 kWh on the easier instances. Consequently, query times are slower by about a factor of 4. For harder scenarios, however, the potential function π_f provides better results due to better scalability. Note that when using π_d, at least one query exceeded our threshold of one hour in computation time on Eur-PG. In summary, we can solve EVCSP *optimally* for typical ranges in less than a second, even on hard instances. For very long ranges, our algorithm computes the optimum in well below a minute on average (using π_f), despite its exponential running time.

In Table 4, we evaluate our heuristic approach for different choices of ε (in % of total SoC). During preprocessing, new shortcuts are included only if they *significantly* improve on the existing ones. Thus, preprocessing becomes faster and core sizes (not reported in the table) decrease by up to 30 %. Regarding queries, we achieve a speedup by an order of magnitude. However, the choice of ε clearly matters. For $\varepsilon = 0.01$, the decrease in quality is negligible, but speedup (about a factor of 2) is rather limited as well. For $\varepsilon = 0.1$, on the other hand, the optimal solution is still found very often. The average error is roughly 0.2 %, while the overall maximum error is 5 %, which is acceptable in practice. Finally, for $\varepsilon = 1.0$, both the average and maximum error increase significantly. Given that speedup is also limited compared to $\varepsilon = 0.1$, we conclude that the latter provides the best tradeoff in terms of quality and query performance: providing high-quality solutions, it enables query times of well below 100 ms, which is fast enough even for interactive applications. Moreover, note that in cases where no path is found (about 1 % of all queries for $\varepsilon = 0.1$), a simple fallback solution could return the energy-optimal path, which can be computed quickly [9, 16, 35].

6 Conclusion

We introduced a novel framework for computing constrained shortest paths for EVs in practice, using realistic consumption models. Our base algorithm TFP respects battery

constraints and accounts for adaptive speeds in a mathematically sounder way that unlocks *both* better query performance *and* improved solution quality when compared to previous approaches using discretized, sampled speeds. Nontrivial speedup techniques based on A* and CH make the algorithm practical. For typical EV ranges, it computes *optimal* solutions in less than a second, making it the first practical *exact* approach – with running times similar to previous inexact methods [8, 25, 28]. Our own heuristic enables even faster queries while retaining high-quality solutions.

The result of our computations is not only the suggested route from source to target but also optimal driving speeds along that route. In practice, these can be passed to the driver as recommendations or directly to a cruise control unit. With the advent of autonomous vehicles, the output of our algorithms can also be used for speed planning of self-driving EVs, either directly or after further refinement [19]. For future work, a next step would be the integration of planned charging stops [7, 37]. From a practical point of view, it might also be interesting to consider adaptive speeds only on the fastest roads (e. g., motorways), where going below the speed limit really pays off the most. Then, contracting vertices incident to *constant* edges in CH might be a promising approach. Finally, we are interested in the integration of variable speed limits imposed by, e. g., historic knowledge of traffic patterns [5, 13, 20].

References

1 Shubham Agrawal, Hong Zheng, Srinivas Peeta, and Amit Kumar. Routing Aspects of Electric Vehicle Drivers and their Effects on Network Performance. *Transportation Research Part D: Transport and Environment*, 46:246–266, 2016.

2 Johannes Asamer, Anita Graser, Bernhard Heilmann, and Mario Ruthmair. Sensitivity Analysis for Energy Demand Estimation of Electric Vehicles. *Transportation Research Part D: Transport and Environment*, 46:182–199, 2016.

3 Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In *Algorithm Engineering: Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.

4 Lucas S. Batista, Felipe Campelo, Frederico G. Guimarães, and Jaime A. Ramírez. A Comparison of Dominance Criteria in Many-Objective Optimization Problems. In *Proceedings of the 13th IEEE Congress on Evolutionary Computation (CEC'11)*, pages 2359–2366. IEEE, 2011.

5 Gernot V. Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 18:1.4:1–1.4:43, 2013.

6 Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15:2.3:1–2.3:31, 2010.

7 Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles. In *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'15)*, pages 44:1–44:10. ACM, 2015.

8 Moritz Baum, Julian Dibbelt, Lorenz Hübschle-Schneider, Thomas Pajor, and Dorothea Wagner. Speed-Consumption Tradeoff for Electric Vehicle Route Planning. In *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'14)*, volume 42 of *OpenAccess Series in Informatics (OASIcs)*, pages 138–151. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014. doi:10.4230/OASIcs.ATMOS.2014.138.

9 Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Energy-Optimal Routes for Electric Vehicles. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'13)*, pages 54–63. ACM, 2013.

10 Luca Bedogni, Luciano Bononi, Marco Di Felice, Alfredo D'Elia, Randolf Mock, Francesco Morandi, Simone Rondelli, Tullio Salmon Cinotti, and Fabio Vergari. An Integrated Simulation Framework to Model Electric Vehicles Operations and Services. *IEEE Transactions on Vehicular Technology*, 65(8):5900–5917, 2016.

11 Marco Blanco, Ralf Borndörfer, Nam-Dung Hoang, Anton Kaier, Adam Schienle, Thomas Schlechte, and Swen Schlobach. Solving Time Dependent Shortest Path Problems on Airway Networks Using Super-Optimal Wind. In *Proceedings of the 16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'16)*, volume 54 of *OpenAccess Series in Informatics (OASIcs)*, pages 12:1–12:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.ATMOS.2016.12.

12 Karin Brundell-Freij and Eva Ericsson. Influence of Street Characteristics, Driver Category and Car Performance on Urban Driving Patterns. *Transportation Research Part D: Transport and Environment*, 10(3):213–229, 2005.

13 Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

14 Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. User-Constrained Multi-Modal Route Planning. *ACM Journal of Experimental Algorithmics*, 19:3.2:1–3.2:19, 2015.

15 Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

16 Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal Route Planning for Electric Vehicles in Large Networks. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI'11)*, pages 1108–1113. AAAI Press, 2011.

17 Stephan Erb, Moritz Kobitzsch, and Peter Sanders. Parallel Bi-Objective Shortest Paths Using Weight-Balanced B-Trees with Bulk Updates. In *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*, volume 8504 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2014.

18 Chiara Fiori, Kyoungho Ahn, and Hesham A. Rakha. Power-Based Electric Vehicle Energy Consumption Model: Model Development and Validation. *Applied Energy*, 168:257–268, 2016.

19 Carlos Flores, Vicente Milanés, Joshué Pérez, David González, and Fawzi Nashashibi. Optimal Energy Consumption Algorithm Based on Speed Reference Generation for Urban Electric Vehicles. In *Proceedings of the 11th IEEE Intelligent Vehicles Symposium (IV'15)*, pages 730–735. IEEE, 2015.

20 Luca Foschini, John Hershberger, and Subhash Suri. On the Complexity of Time-Dependent Shortest Paths. *Algorithmica*, 68(4):1075–1097, 2014.

21 Stefan Funke and Sabine Storandt. Polynomial-Time Construction of Contraction Hierarchies for Multi-Criteria Objectives. In *Proceedings of the 15th Meeting on Algorithm Engineering & Experiments (ALENEX'13)*, pages 31–54. SIAM, 2013.

22 Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route Planning with Flexible Objective Functions. In *Proceedings of the 12th Workshop on Algorithm Engineering & Experiments (ALENEX'10)*, pages 124–137. SIAM, 2010.

23 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, 2012.

24 Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.

25 Michael T. Goodrich and Paweł Pszona. Two-Phase Bicriterion Search for Finding Fast and Efficient Electric Vehicle Routes. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'14)*, pages 193–202. ACM, 2014.

26 Gabriel Y. Handler and Israel Zang. A Dual Algorithm for the Constrained Shortest Path Problem. *Networks*, 10(4):293–309, 1980.

27 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

28 Frederik Hartmann and Stefan Funke. Energy-Efficient Routing: Taking Speed into Account. In *Proceedings of the 37th Annual German Conference on Advances in Artificial Intelligence (KI'14)*, volume 8736 of *Lecture Notes in Computer Science*, pages 86–97. Springer, 2014.

29 Stefan Hausberger, Martin Rexeis, Michael Zallinger, and Raphael Luz. Emission Factors from the Model PHEM for the HBEFA Version 3. Technical report I-20/2009, University of Technology, Graz, 2009.

30 James Larminie and John Lowry. *Electric Vehicle Technology Explained, 2nd Edition*. John Wiley & Sons, Ltd., 2012.

31 Mingsong Lv, Nan Guan, Ye Ma, Dong Ji, Erwin Knippel, Xue Liu, and Wang Yi. Speed Planning for Solar-Powered Electric Vehicles. In *Proceedings of the 7th International Conference on Future Energy Systems (e-Energy'16)*, pages 6:1–6:10. ACM, 2016.

32 Enrique Machuca and Lawrence Mandow. Multiobjective Heuristic Search in Road Maps. *Expert Systems with Applications*, 39(7):6435–6445, 2012.

33 Lawrence Mandow and José-Luis Pérez-de-la-Cruz. Multiobjective A* Search with Consistent Heuristics. *Journal of the ACM*, 57(5):27:1–27:24, 2010.

34 Ernesto Q. V. Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

35 Martin Sachenbacher, Martin Leucker, Andreas Artmeier, and Julian Haselmayr. Efficient Energy-Optimal Routing for Electric Vehicles. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI'11)*, pages 1402–1407. AAAI Press, 2011.

36 Peter Sanders and Lawrence Mandow. Parallel Label-Setting Multi-Objective Shortest Path Search. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS'13)*, pages 215–224. IEEE, 2013.

37 Sabine Storandt. Quick and Energy-Efficient Routes: Computing Constrained Shortest Paths for Electric Vehicles. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'12)*, pages 20–25. ACM, 2012.

38 Sabine Storandt. Route Planning for Bicycles – Exact Constrained Shortest Paths Made Practical via Contraction Hierarchy. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, pages 234–242. AAAI Press, 2012.

39 Tessa Tielert, David Rieger, Hannes Hartenstein, Raphael Luz, and Stefan Hausberger. Can V2X Communication Help Electric Vehicles Save Energy? In *Proceedings of the 12th International Conference on ITS Telecommunications (ITST'12)*, pages 232–237. IEEE, 2012.

40 Chi Tung Tung and Kim Lin Chew. A Multicriteria Pareto-Optimal Path Algorithm. *European Journal of Operational Research*, 62(2):203–209, 1992.

41 Yan Wang, Jianmin Jiang, and Tingting Mu. Context-Aware and Energy-Driven Route Optimization for Fully Electric Vehicles via Crowdsourcing. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1331–1345, 2013.

42 Enjian Yao, Zhiqiang Yang, Yuanyuan Song, and Ting Zuo. Comparison of Electric Vehicle's Energy Consumption Factors for Different Road Types. *Discrete Dynamics in Nature and Society*, 2013.

A Quasi-Polynomial-Time Approximation Scheme for Vehicle Routing on Planar and Bounded-Genus Graphs*

Amariah Becker[1], Philip N. Klein[2], and David Saulpic[3]

1 **Department of Computer Science, Brown University, Providence, RI, USA**
becker@cs.brown.edu
2 **Department of Computer Science, Brown University, Providence, RI, USA**
klein@cs.brown.edu
3 **Department of Computer Science, École Normale Supérieure, Paris, France**
david.saulpic@ens.fr

Abstract

The Capacitated Vehicle Routing problem is a generalization of the Traveling Salesman problem in which a set of clients must be visited by a collection of capacitated tours. Each tour can visit at most Q clients and must start and end at a specified depot. We present the first approximation scheme for Capacitated Vehicle Routing for non-Euclidean metrics. Specifically we give a quasi-polynomial-time approximation scheme for Capacitated Vehicle Routing with fixed capacities on planar graphs. We also show how this result can be extended to bounded-genus graphs and polylogarithmic capacities, as well as to variations of the problem that include multiple depots and charging penalties for unvisited clients.

1998 ACM Subject Classification G.2.2 Graph Algorithms

Keywords and phrases Capacitated Vehicle Routing, Approximation Algorithms, Planar Graphs

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.12

1 Introduction

Vehicle routing refers to a class of problems in which one selects routes for a vehicle that must make deliveries or pickups at specified locations. Irnich et al., in the introductory chapter [14, p. 3] of a book on vehicle routing, define the CAPACITATED VEHICLE ROUTING problem (CVRP) as follows: there is a (directed or undirected) graph G with edge costs, a distinguished vertex r called the *depot*, and for each vertex v, a demand $q(v)$. Finally, there is a number Q, which is the capacity of the vehicle(s). A solution is a set of *tours*, where each tour starts and ends at the depot and serves the demands of some of the vertices it visits. Each tour must serve a total demand of at most Q, and every demand must be served by one of the tours. The goal is to find a solution whose total cost is minimum.

To be consistent with the algorithms literature, we use a slightly different definition of CAPACITATED VEHICLE ROUTING: we assume that the demands are all 0 or 1, i.e. that there is a set Z of vertices, called the *clients*, where the demand is 1, and demands at other vertices are zero. (One can model multiple clients located at the same vertex v by introducing artificial vertices, adjacent to v via artificial edges of cost zero.)

* This research was supported by National Science Foundation grant CCF-1409520.

The problem is APX-hard for an arbitrary graph when $Q \geq 3$ [5], and to approximate it within a factor 1.5 is NP-complete even in a tree when Q is unbounded [10]. We are interested in finding solutions that are within a factor $1+\epsilon$ of optimal for any given ϵ. However, despite the fact that the problem is often described as a problem in **road networks**, theoretical work on algorithms achieving $1+\epsilon$ approximation has been restricted to the Euclidean case.

Since the family of planar graphs (or more generally graphs of bounded genus) are useful for modeling road networks,[1] it is desirable to find an algorithm that achieves a $1+\epsilon$ approximation on such graphs with arbitrary nonnegative edge costs. Before this work, no such approximation scheme was known for any graph class (except trees, where the problem is polynomial-time-solvable for fixed capacity).

▶ **Theorem 1.** *For any $\epsilon > 0$ and any $Q > 0$, there is a quasi-polynomial-time algorithm that, given an instance of* CAPACITATED VEHICLE ROUTING *in which the capacity is Q and the graph is planar, finds a solution whose cost is at most $1+\epsilon$ times optimum.*

A family of algorithms of this form, where for each $\epsilon > 0$ there is an algorithm that achieves a $1+\epsilon$ approximation, is an *approximation scheme*. By *quasi-polynomial* is meant a function $f(n)$ that is $O(n^{\log^c n})$ for some constant c.

As pointed out in [14], with a limited fleet, it may be impossible to service all requests, and there is an advantage in simultaneously optimizing both routing and request selection. We model this using a natural generalization of the capacitated-vehicle-routing problem: an instance specifies also a *penalty* for each client; the solution is allowed to miss some clients and the goal is to find a solution that minimizes the sum of cost plus penalties. We call this CAPACITATED VEHICLE ROUTING WITH PENALTIES.

This generalization can handle the *vehicle routing problem with private fleet and common carrier* (VRPPC), "where customers may either be served by using owned vehicles with traditional routes or be assigned to a common carrier, which serves them directly at a prefixed cost" [14, p. 13].

Our quasi-polynomial-time approximation scheme can also be extended to handle CAPACITATED MULTIPLE-DEPOT VEHICLE ROUTING. In this version, several vertices are designated as depots, and tours can start and end at different depots.

The algorithm can also handle a graph of bounded genus and a capacity Q that is polylogarithmic. (Q is considered constant in Theorem 1.)

▶ **Theorem 2.** *For any $\epsilon > 0$, any $g \geq 0$, any $R \geq 0$ and any $c \geq 0$, there is a quasi-polynomial-time algorithm that, given an instance of* CAPACITATED MULTIPLE-DEPOT VEHICLE ROUTING WITH PENALTIES *in which the capacity Q is $O(\log^c n)$ and the graph has genus g with R designated depots, finds a solution whose cost is at most $1+\epsilon$ times optimum.*

1.1 Related work

CAPACITATED VEHICLE ROUTING is a generalization of TRAVELING SALEMAN PROBLEM (for TSP $Q = n$).

Haimovich and Rinnoy Kan [12] showed the following:

▶ **Lemma 3.** *For* CAPACITATED VEHICLE ROUTING *with capacity Q and client set Z,*

$$OPT \geq \frac{2}{Q} \sum \{d(c,r) \ : \ c \in Z\} . \tag{1}$$

[1] Aside from highways, the nonplanarities in road networks tend to be localized, and informally approximation schemes for planar graphs can often "work around" these localized nonplanarities.

This lemma implies a constant-factor approximation in general metrics, where the constant depends on the approximation ratio for TSP. Capacitated Vehicle Routing in general graphs is APX-hard for every fixed $Q \geq 3$ [4, 5]. Haimovich and Rinnoy Kan [12] gave a polynomial-time approximation scheme (PTAS) for the Euclidean plane for the case when the capacity Q is constant. Asano et al. [5] gave an algorithm that was a PTAS when Q is $O(\log n / \log\log n)$. Mathieu and Das [7] gave a quasi-polynomial-time approximation scheme that handles arbitrary Q. Building on [7], Adamaszek, Czumaj, and Lingas [1] give a polynomial-time approximation scheme that for any $\epsilon > 0$ can handle Q up to $2^{\log^{\delta} n}$ where δ is a positive number that depends on ϵ. There has been some work on approximation schemes to $\mathbb{R}^3$ [16] and $\mathbb{R}^d$ [15] but these require that Q be $O(\log^{1/d} \log n)$. No polynomial-time approximation scheme is known for arbitrary Q, even for $\mathbb{R}^2$.

There is little known about approximation of vehicle routing in special metrics other than low-dimensional Euclidean metrics. Hamaguchi and Katoh [13] and Asano, Katoh, and Kawashima [3] gave better constant-factor approximation algorithms for the case where the graph is a tree.

1.2 Our Approach

Our algorithm uses a recursive decomposition of the graph via shortest-path separators. That is, there is a recursive clustering in which the vertices on the boundary of each cluster lie on a small number of shortest paths. This general idea has been used in several previous approximation schemes for planar and bounded-genus graphs [2, 6, 8, 11]. The closest previous use was in addressing the k-center problem [8], and we use a lemma from that paper stating that such a recursive decomposition exists that has logarithmic depth.[2]

This paper introduces several new ideas in order to apply the recursive decomposition to vehicle routing. Before finding the recursive decomposition, the algorithm must *prune* the graph to eliminate vertices too far from the depot to participate in an optimal solution. The shortest paths bounding each cluster are *subpaths of shortest paths from the depot*. This ensures that if one vertex of a bounding shortest path is farther along the bounding shortest path than another, it is also farther from the depot. This in turn is useful since, as we saw in Section 1.1, there is a lower bound (3) on OPT that is based on the distance of clients from the depot.

Some of the vertices of these bounding shortest paths are designated as *portals*, and (some) paths of the solution are restricted to entering and leaving clusters via portals. This in itself is not novel; portals have been used before. We introduce two new ideas. In previous use of portals in approximation schemes for planar graphs, portals are selected uniformly along a path. In this paper, it is essential that the portals be selected in a nonuniform fashion: the farther from the depot, the greater the spacing between portals. This introduces more error in areas of the graph farther from the depot but such error can be tolerated due to the lower bound (3).

Second, requiring the entire solution to pass in and out of clusters via portals would introduce too much error. Instead, we only require a tour to use portals in between picking up clients. That way, we can bound the error in terms of clients and their distance from the depot. (This error also depends on the depth of nesting of the recursion, since in the process

[2] Such a decomposition was earlier described by Thorup [17] in the context of approximate distance oracles. However, in addressing the generalization to multiple depots we use a generalization of the decomposition that follows from slight modification of the proof in [8].

of picking up a client the tour might pass through many clusters at different levels of nesting, but the depth of nesting is merely logarithmic.)

Having shown that an (approximately) optimal solution can be assumed to use clusters in this way, we reduce the problem to one we can address using dynamic programming. In most previous work on approximation schemes for planar graphs, this consists in simply finding an optimal solution in a graph of bounded branchwidth (or treewidth) but because the solution can pass through the boundary without using portals, that does not quite suffice in our case; the dynamic program must also make use of the metric on the entire graph as well.[3]

Paper Outline. Section 2 provides preliminary definitions. In Section 3 we describe the graph decomposition and portal selection. In Section 4 we prove a structure theorem that shows a near-optimal solution with the restricted structure exists. Section 5 provides the dynamic program that finds such a near optimal solution in quasi-polynomial time. Finally in Section 6 we describe several generalizations of our result.

2 Preliminaries

Let $G = (V, E)$ be a graph. We denote the vertex set of G by $V(G)$. G is *planar* if it can be embedded on the surface of a sphere without any edge crossings. We let $n = |V|$. A planar graph with n vertices and no parallel edges has $O(n)$ edges.

For any edge set $F \subseteq E$ the *boundary* of F, denoted $\partial(F)$, is the set of vertices that are incident both to edges in F and edges in $E - F$.

We use $d(u, v)$ to denote the (shortest path) distance from u to v. We can easily compute all-pairs shortest paths in polynomial time, so we assume throughout the paper that we have access to all (precomputed) distances. Additionally we assume that the cost of any edge (u, v) is $d(u, v)$; if not, it would not be used and can be removed from the graph. We use $d(P) = \sum_{(u,v)\in P} d(u, v)$ to denote the cost of a path P.

For a graph G and vertex r, an *r-rooted shortest path tree* T is an r-rooted tree in which for all v in V the r-to-v path in T is a shortest path. For any vertex u on a shortest r-to-v path, we call the u-to-v subpath a *from-r shortest subpath*. Such a subpath must also be a shortest u-to-v path.

A *triangulated* planar graph is one in which every face has exactly three incident edges. A planar graph can easily be triangulated in linear time by adding edges that recursively subdivide faces with more than three incident edges. Each new edge (u, v) is given cost $d(u, v)$. Triangulating a planar graph requires adding $O(n)$ edges.

A *recursive partition* of a set Y is a rooted binary tree in which each node is labeled with a *cluster* $\mathcal{C} \subseteq Y$ such that the root node is labeled with $\mathcal{C} = Y$, and for any node labeled with cluster $\mathcal{C}_0$ the children nodes are labeled with clusters $\mathcal{C}_1$ and $\mathcal{C}_2$ that form a partition of $\mathcal{C}_0$ [8].

A *recursive clustering* of a graph G is a rooted binary tree in which

- each node is labeled with a *cluster* $\mathcal{C} \subseteq V(G)$,
- If x is a child of y then the cluster associated with x is a subset of the cluster associated with y, and
- there is a mapping $\phi(\cdot)$ that maps each vertex v of G to a leaf cluster $\phi(v)$ that contains v.

Each vertex v is considered to be *assigned to* each cluster containing $\phi(v)$.

[3] A similar technique was used in [8].

A vertex v of a cluster $\mathcal{C}$ is a *boundary vertex* of the cluster if v also belongs to a cluster $\mathcal{C}'$ that neither contains nor is contained in C. An edge uv of G is a *boundary edge* of the cluster if u is in the cluster and v is not. The *depth* of a recursive clustering is the depth of the rooted binary tree.

For an instance of Capacitated Vehicle Routing, Z is the set of clients, r is the depot, and Q is the capacity. A solution is a collection of *tours*, each starting and ending at r and visiting at most Q clients.

3 Decomposing the Graph

3.1 Graph Pruning

As a preprocessing step, the algorithm prunes from the graph those vertices that have no clients and are very far from the depot. Specifically, the algorithm deletes each vertex that does not lie on some u-to-v shortest path with $u, v \in Z \cup \{r\}$. Since the optimal solution is composed of such paths, pruning does not increase OPT.

▶ **Lemma 4.** *For all vertices w that remain after the preprocessing step, $d(r, w) \leq OPT$*

Proof. Since w survived the pruning step, it must lie on some u-to-v shortest path with $u, v \in Z \cup \{r\}$. Without loss of generality, assume that the optimal solution visits u before visiting v. Therefore $OPT \geq d(r, u) + d(u, v) \geq d(r, u) + d(u, w) \geq d(r, w)$ where the final inequality comes from the triangle inequality. ◀

3.2 Cluster Decomposition

The following lemma is a slight generalization of a lemma in [8] (though it is probably folklore):

▶ **Lemma 5** (Generalization of Lemma 3.1 in [8])**.** *Let T be a tree of degree at most three. Let Y be a subset of the vertices. There is a depth-$O(\log |Y|)$ recursive clustering of T such that*

- *there are no boundary vertices,*
- *each leaf cluster is assigned only one vertex of Y, and*
- *each cluster C has at most four boundary edges.*

Let G be a planar embedded graph and let T be a spanning tree of G. Let G' be obtained from G by adding artificial edges to triangulate G. Let G^* be the planar dual of G'. Let T^* be the set of edges of G^* corresponding to edges of G' that are not in T. Then T^* is a spanning tree of G^* (the *interdigitating tree*). Each edge of T^* corresponds to a nontree edge in G, which in turn defines a cycle consisting of that nontree edge together with a path through T. Since G' is triangulated, G^* has degree at most three. Map each vertex of G' to one of the faces to which it is incident. Let Y be the faces to which elements of Z are mapped. By choosing T to be an r-rooted shortest-path tree of G and applying Lemma 5, we obtain the following generalization of a corollary of [8].

▶ **Lemma 6** (Generalization of Corollary 3.1 of [8])**.** *Let G be a planar embedded triangulated graph with edge costs, let Z be a subset of the vertices, and let r be a vertex of G. There is a depth-$O(\log |Z|)$ recursive clustering of G with the following properties:*

- *there are no boundary edges,*
- *for each cluster, there are at most eight from-r shortest paths such that the boundary vertices of the cluster are the vertices that lie on these paths, and*
- *at most three vertices of Z are assigned to each leaf cluster.*

The algorithm computes a recursive clustering as described in Lemma 6.

3.3 Choosing Portals

The algorithm designates *portals* along each cluster boundary in a two-step process. First it designates some of the vertices as *spacers*. Second, for each cluster it designates as portals a subset of the cluster's boundary vertices, including those boundary vertices that are spacers and also some additional vertices.

The choice of spacers depends on a parameter δ. We will choose

$$\delta = \frac{\epsilon}{(Q+4)c\log|Z|} \tag{2}$$

where c is an absolute constant to be determined in the proof of Theorem 11.

We first describe spacer selection. Let T be the shortest-path tree. Let $\hat{v}$ be the vertex farthest from r that remains after the pruning step and let $\delta > 0$. Consider the unpruned vertices in increasing order of distance from r. For each vertex v in turn, designate v as a spacer if no ancestor in T of v within distance $\delta \ \max(d(r,s_{i-1}), \frac{1}{|Z|}d(r,\hat{v}))$ has been designated a spacer.

▶ **Lemma 7.** *Each from-r shortest path has at most $2+\log_{1+\delta}\frac{|Z|}{\delta}$ spacers.*

Proof. Let P be a from-r shortest path, and let $r{=}s_0, s_1, \ldots, s_\ell$ be the spacers on P in increasing order of distance from r. We bound ℓ as follows. For each $i > 0$, $d(s_{i-1},s_i) > \delta\ d(r,s_{i-1})$, so

$$d(r,s_i) = d(r,s_{i-1}) + d(s_{i-1},s_i) > (1+\delta)d(r,s_{i-1})\,.$$

By induction,

$$d(r,s_\ell) > (1+\delta)^{\ell-1}d(r,s_1)\,.$$

Since $d(r,s_1) > \delta\frac{1}{|Z|}d(r,\hat{v})$, we infer $d(r,s_\ell) > (1+\delta)^{\ell-1}\frac{\delta}{|Z|}d(r,\hat{v})$, which implies

$$(1+\delta)^{\ell-1} < \frac{|Z|}{\delta}\frac{d(r,s_\ell)}{d(r,\hat{v})} \le \frac{|Z|}{\delta}$$

which shows

$$\ell - 1 < \log_{1+\delta}\frac{|Z|}{\delta}.$$ ◀

The algorithm then designates some of each cluster's boundary vertices as *portals*. For each cluster $\mathcal{C}$, the boundary vertices of $\mathcal{C}$ lie on $O(1)$ from-r shortest subpaths. For each such from-r subpath P, designate as portals the first vertex of P and all the vertices of P that are spacers.

By Lemma 7, each path P contributes $O(\delta^{-1}\log(\delta^{-1}|Z|))$ portals. Using the definition of δ in Equation 2, we obtain

▶ **Lemma 8.** *For each cluster boundary, the above algorithm designates $O(Q\epsilon^{-1}\log^2|Z|)$ portals.*

We also show a bound on the distance along the boundary to the nearest portal.

▶ **Lemma 9.** *For every cluster $\mathcal{C}$ and boundary vertex $v \in \partial(\mathcal{C})$, there is a portal p of $\mathcal{C}$ and a p-to-v path of cost at most $\delta\ (d(r,v) + OPT/|Z|)$.*

Proof. By Lemma 6, v must lie on some from-r shortest subpath of a from-r shortest path P. Let s be v's closest ancestor in T that is a spacer. By the algorithm for designating spacers,

$$d(s,v) \leq \delta \ \max(d(r,s), \frac{1}{|Z|}d(r,\hat{v})) \leq \delta \ \max(d(r,v), OPT/|Z|)\,.$$

If s belongs to P then s is a boundary vertex of $\mathcal{C}$ and hence a portal of $\mathcal{C}$. In this case, we take $p = s$. If not, then let p be the first vertex of P and note that $d(p,v) \leq d(s,v)$. ◀

4 Structure Theorem

Consider a solution to Capacitated Vehicle Routing. It consists of a set of tours. Associated with each tour P is a set of vertices in $V(P) \cap Z$ that the tour is considered to visit. For a cluster $\mathcal{C}$, a tour *fragment* with respect to $\mathcal{C}$ is a maximal subpath of a tour all of whose vertices are in C. Every tour starts and ends at the depot r. If r is in $\mathcal{C}$ then it is a boundary vertex of $\mathcal{C}$. Therefore, by maximality, each endpoint of a tour fragment with respect to $\mathcal{C}$ is a boundary vertex of $\mathcal{C}$.

A tour fragment is *visiting* if it visits clients and *passing* if it does not. The endpoints of a visiting fragment are called *gates*.

▶ **Lemma 10.** *Any solution to* Capacitated Vehicle Routing *crosses through* $O(\log|Z|)$ *gates between two consecutive visits to clients.*

Proof. Consider a solution to Capacitated Vehicle Routing. Let u and v be two consecutive clients visited by the solution, and let P be the subpath of the tour between its visit to u and its visit to v. Let C be a cluster and let S be a visiting tour segment with an endpoint x on P. Since S is a visiting segment, it visits some vertex. Since no visits occur on P between u and v, the other endpoint y of P must not be an internal vertex of P. Thus either u or v must be assigned to the cluster C. If u is assigned to C then all edges on the u-to-x subpath of P belong to C, and the edge of P after x does not. If v is assigned to C then all edges on the x-to-v subpath of P belong to C, and the edge of P before x does not.

Let $\mathcal{C}_{u,0} \subset \mathcal{C}_{u,1} \subset ... \subset \mathcal{C}_{u,\ell}$ be the clusters that are assigned u, and let $\mathcal{C}_{v,0} \subset \mathcal{C}_{v,1} \subset ... \subset \mathcal{C}_{v,k}$ be the clusters that are assigned v.

For $i = 0, 1, \ldots, \ell$, let $t_{u,i}$ be the last vertex x of P such that the u-to-x subpath of P consists of edges of cluster $C_{u,i}$. For $i = 0, 1, \ldots, k$, let $t_{v,i}$ be the first vertex x of P such that the x-to-u subpath of P consists of edges of cluster $C_{v,i}$.

Since the clusters are non-crossing, P visits $t_{u,0}, t_{u,1}, \ldots, t_{u,\ell}, t_{v,k}, t_{v,k-1}, ..., t_{v,0}$ in order (See Figure 1a). The argument above shows that every gate is one of $t_{u,0}, \ldots, t_{v,0}$.

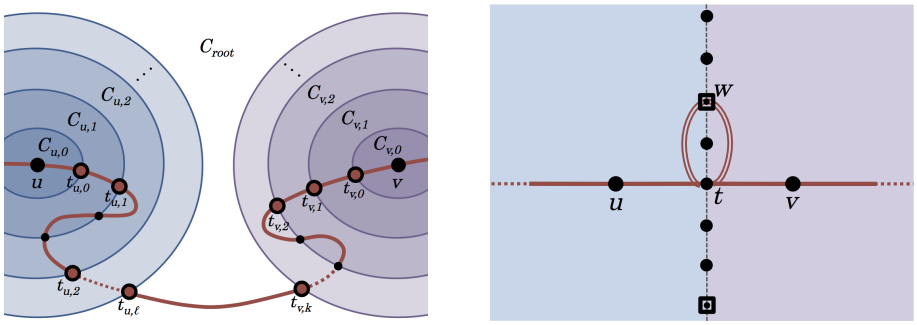
By Lemma 6, the depth of the decomposition is $O(\log|Z|)$, so $k + \ell$ is $O(\log|Z|)$. ◀

Now we break up tours in a different way. Again consider a solution to Capacitated Vehicle Routing. It consists of a set of tours. For a cluster C, a *tour segment* P with respect to $\mathcal{C}$ is *portal-respecting* if P has a portal-to-portal subpath P' such that every vertex not in P' is a boundary vertex of $\mathcal{C}$.

▶ **Theorem 11.** *There exists a portal-respecting solution with cost at most* $(1+\epsilon)OPT$.

Proof. Let $\mathcal{S}^*$ be an optimal solution to Capacitated Vehicle Routing. To prove the theorem, we show how to modify $\mathcal{S}^*$ to construct a portal-respecting solution $\hat{\mathcal{S}}$ by introducing *detours* at every gate, and bound the cost incurred by these detours.

Consider a tour fragment P with respect to some cluster $\mathcal{C}$, and let t be an endpoint of P. The corresponding detour is the subpath of a from-r shortest subpath from t to the



(a) Gates **(b)** Detour

Figure 1 (a) Gates are depicted by large, hollow circles at crossings. Non-gate crossings enclose passing segments (b) Boundary portals are denoted by squares. The double-line paths depict a detour.

nearest portal, and back. (See Figure 1b.) Splicing such a detour into the solution at each gate ensures that the solution is still feasible and is portal-respecting. It remains to show that the cost of these detours is small.

Let u and v be two consecutive clients visited by $\mathcal{S}^*$. By Lemma 10 there is some constant c such that there are at most $c \log |Z|$ gates between u and v where detours will be added. We use this constant c in the definition of δ given in Equation 2. By Lemma 9 the distance from any crossing t to the nearest portal is at most $\delta\ (d(r,t) + OPT/|Z|)$, so the cost of each detour is at most $2\delta\ (d(r,t) + OPT/|Z|)$.

Since $\mathcal{S}^*$ is optimal, the path that $\mathcal{S}^*$ takes from u to v must be a shortest path. By the triangle inequality, $d(r,t) \leq d(r,v) + d(v,t) \leq d(r,v) + d(v,u)$. Therefore, the cost of each detour is at most $2\delta\ (d(u,v) + d(r,v) + OPT/|Z|)$. Summing over all gates gives $2\delta c \log |Z| \big(d(u,v) + d(r,v) + OPT/|Z|\big)$, and summing over all pairs of consecutive clients and using Lemma 3,

$$\sum_{(u,v)\in\mathcal{S}^*} 2\delta c \log |Z| \big(d(u,v) + d(r,v) + OPT/|Z|\big) \leq 2\delta c \log n (OPT + \frac{Q}{2} OPT + OPT)$$

$$= \delta c \log |Z| (Q+4) OPT\,.$$

The definition of δ given in Equation 2 ensures that the total detour cost is at most ϵOPT. ◀

The notion of *portal-respecting* can be applied not just to a solution but to a partial solution as well. This is used in the next section.

5 Dynamic Program

We present two dynamic programs: the first is slow but is the basis of the second, which gives a QPTAS. Both have the same configurations but enumerate the transitions in different ways.

5.1 Configurations

For each cluster, the dynamic program computes the minimal cost of a *portal-respecting* solution that visits all the clients assigned to the cluster. A *configuration* for a cluster $\mathcal{C}$

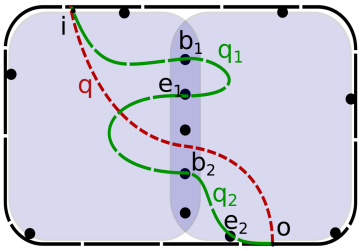


Figure 2 The segment (i, o, q) of the parent cluster is shown in red (short-dashed line). The green path (long-dashed line) shows one way to map this segment onto the child clusters: segment (b_1, e_1) visits q_1 clients and (b_2, e_2) visits q_2 clients with $q_1 + q_2 = q$. Segments (i, b_1), (e_1, b_2), and (e_2, o) are passing segments and visit no clients

describes the tour segments formed by the intersection of $\mathcal{C}$ with a solution. Recall that if the depot r is contained in a cluster, it is a portal of its boundary. Additionally, since each client is assigned to exactly one leaf cluster, we avoid overcounting any client's demand. For a client z, let $D_{\mathcal{C},z}$ be the demand of z inside the cluster $\mathcal{C}$. $D_{\mathcal{C},z} = 1$ if z is assigned to $\mathcal{C}$, otherwise $D_{\mathcal{C},z} = 0$.

A *configuration* $\mathcal{X}$ for cluster $\mathcal{C}$ specifies, for each pair of portals (i, o) of the cluster and for each $q \in \{1, ..., Q\}$, a number $\mathcal{X}_{i,o,q}$ of segments that enter $\mathcal{C}$ at portal i, visit exactly q clients in $\mathcal{C}$, and leave $\mathcal{C}$ at portal o. A configuration for cluster $\mathcal{C}$ is *admissible* if $\Sigma_{i,o,q} q\mathcal{X}_{i,o,q} = \Sigma_{z\in\mathcal{C}} D_{\mathcal{C},z}$.

A *partial solution* S for cluster $\mathcal{C}$ is a set of tour segments that stays inside the cluster. We say that a partial solution S for cluster $\mathcal{C}$ *induces the configuration* $\mathcal{X}$ if every *visiting* segment of S corresponds to a segment described in $\mathcal{X}$ (recall that a visiting segment is one that visits a client).

Our dynamic program computes, for each cluster $\mathcal{C}$ and admissible configuration $\mathcal{X}$, the weight $DP(\mathcal{C}, \mathcal{X})$ of the minimum-weight, portal-respecting partial solution that induces the configuration $\mathcal{X}$.

5.2 Compatibility

To compute the minimal cost of a partial solution for a cluster $\mathcal{C}_0$ that induces the configuration $\mathcal{X}^0$, the algorithm determines possible configurations for the two child clusters of $\mathcal{C}_0$, namely $\mathcal{C}_1$ and $\mathcal{C}_2$. Let $\mathcal{C}_0$ be a cluster, with two child clusters $\mathcal{C}_1$ and $\mathcal{C}_2$, and let $\mathcal{X}^0$, $\mathcal{X}^1$, and $\mathcal{X}^2$ be three configurations such that $\mathcal{X}^i$ is admissible for $\mathcal{C}_i$. We say that $\mathcal{X}^1$ and $\mathcal{X}^2$ are *compatible* with $\mathcal{X}^0$ if each segment (i, o, q) described in $\mathcal{X}^0$ can be mapped to a tuple of segments of $\mathcal{X}^1$ and $\mathcal{X}^2$, $\big((b_1, e_1, j_1, q_1), \cdots, (b_K, e_K, j_K, q_K)\big)$, where $j_i \in \{1, 2\}$, b_i and e_i are portals of $\mathcal{C}_{j_i}$, and such that $0 < q_i < Q$ and $\Sigma_{i=1}^{K} q_i = q$. We use $(\mathcal{X}^1, \mathcal{X}^2) \sim \mathcal{X}^0$ to denote that $\mathcal{X}^1$ and $\mathcal{X}^2$ are compatible with $\mathcal{X}^0$. To ensure that partial solutions are portal-respecting, configurations only need to describe the segments that visit clients. The other segments need not cross boundaries at portals, so we assume them to be shortest paths in the original graph.

Conversely, every segment of $\mathcal{X}^1$ and $\mathcal{X}^2$ must correspond to exactly one segment of $\mathcal{X}^0$. Intuitively, this means that every segment in $\mathcal{X}^0$ can be broken into subsegments that respect the portals of $\mathcal{C}_1$ and $\mathcal{C}_2$. The path from i to o can be divided into a shortest path from i to b_1, segments from b_i to e_i visiting q_i clients in the cluster $\mathcal{C}_{j_i}$, and shortest paths from e_i to b_{i+1} and from e_K to o visiting 0 clients.

We define the *price*, P, of the compatible configurations to be the cost of connecting the segments: $P(\mathcal{C}_0, \mathcal{X}^0, \mathcal{X}^1, \mathcal{X}^2) = d(i, b_1) + \Sigma_{i=1}^{K-1} d(e_i, b_{i+1}) + d(e_K, o)$. Therefore, the minimal cost of a partial solution for a cluster $\mathcal{C}_0$ that induces the configuration $\mathcal{X}^0$ is $DP(\mathcal{C}_0, \mathcal{X}^0) = \min_{(\mathcal{X}^1, \mathcal{X}^2) \sim \mathcal{X}^0} DP(\mathcal{C}_1, \mathcal{X}^1) + DP(\mathcal{C}_2, \mathcal{X}^2) + P(\mathcal{C}_0, \mathcal{X}^0, \mathcal{X}^1, \mathcal{X}^2)$.

The algorithm enumerates all possible configurations $\mathcal{X}^1$ and $\mathcal{X}^2$ that are compatible with $\mathcal{X}^0$. As in the above definitions, the algorithm breaks every segment of $\mathcal{X}^0$ into subsegments, each visiting some clients in $\mathcal{C}_1$ or in $\mathcal{C}_2$. It then adds each subsegment to the corresponding subconfiguration: the subsegment is added to $\mathcal{X}^{j_i}$. As $q_i > 0$ and $\Sigma_{i=1}^{K} q_i = q \leq Q$, $K \leq Q$. The algorithm enumerates all possibilities and calculates the value of $DP(\mathcal{C}_0, \mathcal{X}^0)$.

5.3 Base Cases

Each base case is a cluster in which there are at most three clients. It is therefore straightforward to find the minimal cost of a configuration.

5.4 Final Output and correctness

Since the topmost cluster has r as its only portal, all configurations will consist of r-to-r segments visiting at most Q clients, and collectively visiting all clients of Z. These are exactly the feasible Capacitated Vehicle Routing solutions. The algorithm returns the minimum over all admissible configurations of the top-level cluster, which is the cost of the optimal portal-respecting solution.

▶ **Theorem 12.** *The dynamic programming algorithm described above outputs the minimal weight of a portal-respecting solution to* Capacitated Vehicle Routing.

The proof is omitted here due to space limitations.

5.5 Complexity Analysis

The complexity is determined by the number of compatible configurations. For each cluster and each admissible configuration for this cluster, the algorithm computes

$$|\{\text{ways of breaking a segment}\}|^{|\{\text{segments}\}|}$$

compatible subconfigurations. We first count the number of admissible configurations for a cluster.

▶ **Lemma 13.** *The number of admissible configurations $\mathcal{X}$ for a given cluster $\mathcal{C}$ is*

$$|Z|^{O(Q^3 \epsilon^{-2} \log^2 |Z|)}.$$

Proof. An admissible configuration is a vector $\mathcal{X}$ indexed by two portals (i, o) and a number of clients q. $\mathcal{X}_{i,o,q}$ is the number of segments going from i to o visiting q clients. As the configuration is admissible, $\Sigma_{i,o,q} q \mathcal{X}_{i,o,q} = \Sigma_{z \in \mathcal{C}} D_{\mathcal{C},z} \leq |Z|$ (because $D_{\mathcal{C},z} \in \{0, 1\}$). Therefore $\mathcal{X}_{i,o,q} \leq |Z|$. Moreover, $q \leq Q$ and because by Lemma 8 there are $O(Q\epsilon^{-1} \log |Z|)$ portals for $\mathcal{C}$ the number of choices of (i, o) is less than $O(Q^2 \epsilon^{-2} \log^2 |Z|)$. The number of admissible configurations $\mathcal{X}$ for a given cluster is thus $|Z|^{O(Q^3 \epsilon^{-2} \log^2 |Z|)}$ ◀

We now count the number of compatible subconfigurations for a given configuration $\mathcal{X}$.

▶ **Lemma 14.** *There are $O\big((2Q^3 \epsilon^{-2} \log^2 |Z|)^{Q|Z|}\big)$ compatible subconfiguration pairs for a given configuration.*

Proof. Using the same argument as in Lemma 13, $\Sigma_{i,o,q} q\mathcal{X}_{i,o,q} \leq |Z|$, and as $q > 0$ we can bound the number of segments of a configuration: $\Sigma_{i,o,q}\mathcal{X}_{i,o,q} \leq |Z|$. To break a segment, the algorithm chooses at most Q subsegments, each one consisting of a boolean, a pair of portals and a number of clients visited by the segment (see Section 5.2). As there are $O(Q\epsilon^{-1}\log|Z|)$ child-cluster portals by Lemma 8 and the capacity is bounded by Q, there are $O((2\cdot Q\cdot Q^2\epsilon^{-2}\log^2|Z|)^Q)$ ways of breaking a single segment. As there are fewer than $|Z|$ segments, we have $O((2Q^3\epsilon^{-2}\log^2|Z|)^{Q|Z|})$ compatible subconfiguration pairs per configuration. ◀

▶ **Lemma 15.** *The overall complexity of the dynamic program is*

$$|Z|^{O(\frac{Q^3\log^4|Z|}{\epsilon^2})}\cdot O(\frac{Q^3\log^4|Z|}{\epsilon^2})^{Q|Z|}.$$

Proof. As stated above, the dynamic program computes for each cluster and each admissible configuration, all compatible subconfigurations. As the decomposition is a binary tree with at most one leaf per client, the number of clusters is $O(Z)$. Combining this with Lemma 13 and Lemma 14, the total complexity is therefore $O\big(|Z|\cdot|Z|^{O(Q^3\epsilon^{-2}\log^2|Z|)}\big(2Q^3\epsilon^{-2}\log^2|Z|\big)^{Q|Z|}\big)$. ◀

5.6 QPTAS

The slowest operation in the DP is generating all compatible subconfigurations for a given cluster. But this is very redundant: two different segments can be broken into the same subsegments. We present a preprocessing step that computes, for every cluster and every triplet of configurations for parent and children clusters, the minimal price $P(\mathcal{C}_0,\mathcal{X}^0,\mathcal{X}^1,\mathcal{X}^2)$ of *passing* segments needed to make the configurations compatible. Recall that a passing segment is one that visits no clients and that they are necessary to connect the *visiting* segments of the subconfigurations.

5.6.1 Preprocessing Algorithm

We describe a recursive algorithm. The entries are a cluster $\mathcal{C}_0$, a configuration $\mathcal{X}^0$ of $\mathcal{C}_0$ and two configurations $\mathcal{X}^1$ and $\mathcal{X}^2$ of the children of $\mathcal{C}_0$. To determine the price for connecting these three configurations, the algorithm considers the first segment appearing in $\mathcal{X}^0$ and breaks it into subsegments belonging to the children. It then deletes this segment from $\mathcal{X}^0$ (giving a new configuration $\hat{\mathcal{X}}^0$) and deletes the subsegments from the children configurations, to obtain the subconfigurations $\hat{\mathcal{X}}^1$ and $\hat{\mathcal{X}}^2$. It tries all possibilities for breaking this segment and returns the cheapest one :

$$P[\mathcal{C}_0,\mathcal{X}^0,\mathcal{X}^1,\mathcal{X}^2] = \min\big(P(\mathcal{C}_0,\hat{\mathcal{X}}^0,\hat{\mathcal{X}}^1,\hat{\mathcal{X}}^2) + d(i,b_1) + \Sigma d(e_\iota,b_{\iota+1}) + d(b_K,o)\big).$$

The base case is when there are no segments in $\mathcal{X}^0$ (i.e. $\mathcal{X}^0_{i,o,q} = 0,\ \forall i,o,q$). Here the algorithm just checks that there are no remaining segments in $\mathcal{X}^1$ and $\mathcal{X}^2$. It returns 0 if there are none and ∞ otherwise. This algorithm can be memoized because the number of segments in the first configuration is strictly decreasing.

5.6.2 Correctness

We prove the correctness of this algorithm by induction. The base case is an empty configuration for the parent cluster. The two subconfigurations are therefore compatible

with it only if they are also empty. If $\mathcal{X}^0$ is compatible with $\mathcal{X}^1$ and $\mathcal{X}^2$, then the chosen segment of $\mathcal{X}^0$ corresponds by definition to at most Q subsegments in $\mathcal{X}^1$ and $\mathcal{X}^2$. The algorithm enumerates all possible ways to break the segment, so at least one will result in a compatible configuration. Reciprocally, if $\mathcal{X}^0$ is not compatible with $\mathcal{X}^1$ and $\mathcal{X}^2$, the first segment cannot be broken in such a way that results in three compatible configurations, so the algorithm avoids false positives.

5.6.3 Complexity

The complexity of this preprocessing step is

$$O\big(|\{\text{clusters}\}| \cdot |\{\text{configurations}\}|^3 \cdot |\{\text{ways of breaking a segment}\}|\big).$$

We showed in Lemma 13 that there is $|Z|^{O(Q^3\epsilon^{-2}\log^4|Z|)}$ configurations and we showed in the proof of Lemma 14 that there are $O\big((2Q^3\varepsilon^2\log^4|Z|)^Q\big)$ ways of breaking a segment, so the preprocessing can be achieved in $O(|Z| \cdot |Z|^{O\left(Q^3\epsilon^{-2}\log^4|Z|\right)}(2Q^3\varepsilon^{-2}\log^4|Z|)^Q) = O(2^{P(\log|Z|,Q,\varepsilon^{-1})})$ where P is some polynomial.

We can use this preprocessing step to improve the complexity of the main DP. Instead of breaking segments we try all compatible configurations for the children clusters, of which there are at most $O(|\{\text{configuration}\}|^2)$. The complexity of this improved DP is therefore $O(|\{\text{clusters}\}| \times |\{\text{configuration}\}|^3)$ and is dominated by the preprocessing step. Combining this analysis with Theorem 11 gives a QPTAS for planar graphs, proving Theorem 1.

6 Generalizations

6.1 Multiple depots

The techniques presented in this paper can be extended to address the multiple-depot version of Vehicle Routing, assuming a constant number of depots. In this variation, each tour can start and end at different depots. Let R denote the set of depots, and for $v \in Z$ let r_v denote the closest depot to v (note that the tour that visits v does not necessarily visit r_v). The generalization relies on two key observations.

First, the recursive clustering can be slightly modified in the following way. Let a *from-R shortest path* be a from-r shortest path for some $r \in R$.

▶ **Lemma 16.** *Let G be a planar embedded graph with edge costs, and let R and Z be subsets of the vertices. There is a depth-$O(\log|Z|)$ recursive clustering of G with the following properties:*

- *there are no boundary edges,*
- *for each cluster, there are $O(|R|)$ from-R shortest subpaths such that the boundary vertices of the cluster are the vertices that lie on these paths, and*
- *at most three vertices of Z are assigned to each leaf cluster.*

Proof. We sketch the proof. It follows the proof of Lemma 6, which in turn follows that of [8]. Consider the R-rooted shortest-path forest F. Each tree is rooted at some $r \in R$, and consists of those vertices v for which $r_v = r$. Construct a tree T by arbitrarily linking the trees of F, and then use the construction of Lemma 6. This gives a decomposition such that each cluster is bounded by four fundamental cycles of the tree T. Since a fundamental cycle in T consists of at most $2|R|$ from-R shortest paths, this concludes the proof. ◀

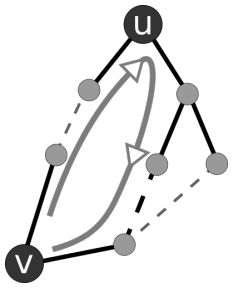


Figure 3 Here, u and v are depots. The forest is in black, the plain lines are the shortest-path trees from u and v and the dashed one is the connecting edge. The dashed, grey lines are edges not in T. The arrow is the boundary of a cluster: there is one fundamental cycle in T, and thus two from-R shortest paths.

Portals in this decomposition are designated the same way as in Section 3. The cost of a detour becomes $\delta(d(v, r_v) + \mathrm{OPT}/Z)$ (using the notation of Section 3), and therefore Lemma 3 has to be adapted in order to obtain an approximate solution.

Each tour P in the optimal solution contains a trip between one depot and the farthest client in P, so the cost of P is at least $\max\{d(c, r_c) : c \text{ is a client of P}\}$, which in turn is at least

$$\frac{1}{Q}\sum\{d(c, r_c) : c \text{ is a client of P}\}$$

by averaging over the at-most Q clients in P.

Using these modified bounds, we can prove a multiple-depot version of the structure theorem, analogous to Theorem 11. Assuming that $|R|$ is constant, we can adapt the dynamic program for this decomposition to get a QPTAS.

6.2 Bounded genus

To extend our algorithm to handle the case when G is embedded on a surface of genus $g > 0$, we adapt a technique previously used by Eistenstat et al. [8]. Let T be any spanning tree of G, in our case the shortest-path tree rooted at the depot. The algorithm selects [9] $2g$ edges not in T such that cutting the surface along the corresponding cycles (each edge forms a cycle with the corresponding simple path in T) yields a surface (with boundary) of genus 0, and a graph embedded on this surface. The vertices of these cycles lie on shortest from-r paths where r is the depot. The algorithm then cuts along these cycles, duplicating the vertices (an edge belonging to such a cycle ends up on one side or the other). The resulting graph is planar. Next the algorithm forms the cluster decomposition for that graph. Finally, the algorithm merges duplicate vertices together. Merging the duplicates can result in those merged vertices being boundary vertices of the clusters, but fortunately all of those merged vertices lie on at most $2g$ from-r shortest paths, so designation of portals can continue as in Section 3.3 and in total the number of portals per cluster will be $O(Qg\epsilon^{-1}\log^2|Z|)$.

6.3 Handling penalties

The dynamic program of Section 5 computes, for each cluster and each admissible configuration of that cluster, the minimum cost partial solution that induces that configuration. To handle penalties, we change the definition of *solution*, of *admissible* and of *cost*. A solution is now allowed to not visit all the clients, an admissible configuration is allowed to not

visit all the clients, and the cost includes the penalties of unvisited clients. The base cases change slightly to accommodate these changes, but otherwise the dynamic program is mostly unchanged.

One other change to the algorithm is needed. As described in Section 3.1, the algorithm needs to prune the graph, removing vertices that are too far to be included. To handle penalties, we need a more complicated pruning step. The algorithm computes an upper bound b on the value of the optimum that is at most Q times the value of the optimum, and then prunes away every vertex whose distance from the depot is greater than $b/2$. This ensures that, for any cluster found in the pruned graph, the value $d(r, \hat{v})$ is at most $(Q/2)\mathrm{OPT}$, and our analysis can be adapted to show that the number of portals is not too large.

▶ **Lemma 17.** *There exist a polynomial-time algorithm that computes a Q-approximation of the penalty variant of vehicle routing.*

Proof. Consider an instance of the penalty version with capacity Q, and a modified version in which the capacity is 1. Solving the modified instance is easy: for each client, include a depot-to-client round-trip if the cost of this trip is no more than the client's penalty. It remains to show that the optimum value for the original instance is at most Q times the optimum value for the modified instance.

Consider an optimal solution for the original instance. For each tour T in that solution, the cost of T is at least the cost of a round-trip from the depot to the farthest client visited by T. Replace T by a collection of tours, one visiting each of the clients visited by T. Each of these tours is a round trip, and there are Q of them, so their total cost is at most Q times the cost of T. The set of unvisited clients has not changed so the sum of their penalties remains unchanged. Thus the total value of the solution thus obtained is at most Q times the optimum value for the original instance. ◀

References

1 Anna Adamaszek, Artur Czumaj, and Andrzej Lingas. PTAS for k-tour cover problem on the plane for moderately large values of k. *Algorithms and Computation*, pages 994–1003, 2009.

2 Sanjeev Arora, M. Grigni, D. R. Karger, Philip N. Klein, and A. Woloszyn. A polynomial-time approximation scheme for weighted planar graph TSP. In *Proceedings of the 9th Symposium on Discrete Algorithms*, pages 33–41, 1998.

3 Tetsuo Asano, Naoki Katoh, and Kazuhiro Kawashima. A new approximation algorithm for the capacitated vehicle routing problem on a tree. *Journal of Combinatorial Optimization*, 5(2):213–231, 2001.

4 Tetsuo Asano, Naoki Katoh, Hisao Tamaki, and Takeshi Tokuyama. Covering points in the plane by k-tours: a polynomial approximation scheme for fixed k. *IBM Tokyo Research Laboratory Research Report RT0162*, 1996.

5 Tetsuo Asano, Naoki Katoh, Hisao Tamaki, and Takeshi Tokuyama. Covering points in the plane by k-tours: towards a polynomial time approximation scheme for general k. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 275–283. ACM, 1997.

6 Vincent Cohen-Addad, Éric Colin de Verdière, Philip N. Klein, Claire Mathieu, and David Meierfrankenfeld. Approximating connectivity domination in weighted bounded-genus graphs. In *Proceedings of the 48th Annual ACMSymposium on Theory of Computing (STOC*, pages 584–597, 2016. doi:10.1145/2897518.2897635.

7 Aparna Das and Claire Mathieu. A quasipolynomial time approximation scheme for euclidean capacitated vehicle routing. *Algorithmica*, 73(1):115–142, 2015.

8 David Eisenstat, Philip N. Klein, and Claire Mathieu. Approximating k-center in planar graphs. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 617–627. Society for Industrial and Applied Mathematics, 2014.

9 David Eppstein. Dynamic generators of topologically embedded graphs. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 599–608. Society for Industrial and Applied Mathematics, 2003.

10 Bruce L. Golden and Richard T. Wong. Capacitated arc routing problems. *Networks*, 11(3):305–315, 1981.

11 M. Grigni, E. Koutsoupias, and C. Papadimitriou. An approximation scheme for planar graph TSP. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 640–645, 1995.

12 Mark Haimovich and A. H. G. Rinnooy Kan. Bounds and heuristics for capacitated routing problems. *Mathematics of operations Research*, 10(4):527–542, 1985.

13 Shin-ya Hamaguchi and Naoki Katoh. A capacitated vehicle routing problem on a tree. In *International Symposium on Algorithms and Computation*, pages 399–407. Springer, 1998.

14 Stefan Irnich, Paolo Toth, and Daniele Vigo. Chapter 1: The family of vehicle routing problems. In Paolo Toh and Daniele Vigo, editors, *Vehicle Routing: Problems, Methods, and Applications*. SIAM, 2014.

15 Michael Khachay and Roman Dubinin. PTAS for the Euclidean Capacitated Vehicle Routing Problem in R^d. In *Proceedings of the 9th International Conference on Discrete Optimization and Operations Research (DOOR 2016)*, pages 193–205. Springer, 2016.

16 Michael Khachay and Helen Zaytseva. Polynomial time approximation scheme for single-depot euclidean capacitated vehicle routing problem. In *Combinatorial Optimization and Applications*, pages 178–190. Springer, 2015.

17 Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM*, 51(6):993–1024, 2004.

The Directed Disjoint Shortest Paths Problem*

Kristóf Bérczi[1] and Yusuke Kobayashi[2]

1 **Eötvös University, Budapest, Hungary**
berkri@cs.elte.hu
2 **University of Tsukuba, Tsukuba, Japan**
kobayashi@sk.tsukuba.ac.jp

Abstract

In the k disjoint shortest paths problem (k-DSPP), we are given a graph and its vertex pairs (s_1, t_1), ..., (s_k, t_k), and the objective is to find k pairwise disjoint paths $P_1, \ldots, P_k$ such that each path P_i is a shortest path from s_i to t_i, if they exist. If the length of each edge is equal to zero, then this problem amounts to the disjoint paths problem, which is one of the well-studied problems in algorithmic graph theory and combinatorial optimization. Eilam-Tzoreff [5] focused on the case when the length of each edge is positive, and showed that the undirected version of 2-DSPP can be solved in polynomial time. Polynomial solvability of the directed version was posed as an open problem in [5]. In this paper, we solve this problem affirmatively, that is, we give a first polynomial time algorithm for the directed version of 2-DSPP when the length of each edge is positive. Note that the 2 disjoint paths problem in digraphs is NP-hard, which implies that the directed 2-DSPP is NP-hard if the length of each edge can be zero. We extend our result to the case when the instance has two terminal pairs and the number of paths is a fixed constant greater than two. We also show that the undirected k-DSPP and the vertex-disjoint version of the directed k-DSPP can be solved in polynomial time if the input graph is planar and k is a fixed constant.

1998 ACM Subject Classification G.2.2 [Graph Theory] Graph Algorithms

Keywords and phrases Disjoint paths, shortest path, polynomial time algorithm

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.13

1 Introduction

1.1 Disjoint paths problem and disjoint shortest paths problem

The *vertex-disjoint paths problem* is one of the classic and well-studied problems in algorithmic graph theory and combinatorial optimization. In the problem, the input is a graph (or a digraph) $G = (V, E)$ and k pairs of vertices $(s_1, t_1), \ldots, (s_k, t_k)$, and the objective is to find k pairwise vertex-disjoint paths from s_i to t_i, if they exist. If k is part of the input, the vertex-disjoint paths problem is NP-hard [9], and it remains NP-hard even if the input graph is constrained to be planar [12]. The vertex-disjoint paths problem in undirected graphs can be solved in polynomial time when $k = 2$ [17, 19, 22], and Robertson and Seymour's graph minor theory gives an $O(|V|^3)$-time algorithm for the problem when k is a fixed constant [15]. The running time of this algorithm is improved to $O(|V|^2)$ in [10]. The vertex-disjoint paths problem in digraphs is much harder than the undirected version. Indeed, the directed version is NP-hard even when $k = 2$ [6]. The vertex-disjoint paths problem in planar digraphs can

* This work is partially supported by JST ERATO Grant Number JPMJER1305 and by JSPS KAKENHI Grant Numbers JP16K16010 and JP16H03118.

be solved in polynomial time for fixed k [16], and it is fixed parameter tractable with respect to parameter k [3].

The vertex-disjoint paths problem has many applications, for example in transportation networks, VLSI-design [7, 14], or routing in networks [13, 20]. When we deal with such practical applications, it is natural to generalize the problem to finding *short* or *cheap* vertex-disjoint paths. There are many results on the problem to find disjoint paths minimizing a given objective function such as the total length of the paths or the length of the longest path (see Section 1.2). In this paper, we consider the disjoint shortest paths problem introduced in [5], in which each path has to be a shortest path from s_i to t_i. Note that, in contrast to the other problems, the length of each path appears in the constraint of the problem. For an integer k, our problem is formally described as follows.

k Disjoint Shortest Paths Problem (k-DSPP)

Input. A digraph (or a graph) $G = (V, E)$ with a length function $\ell : E \to \mathbb{R}_+$ and k pairs of vertices $(s_1, t_1), \ldots, (s_k, t_k)$ in G.

Find. Pairwise disjoint (vertex-disjoint or edge-disjoint) paths $P_1, \ldots, P_k$ such that P_i is a shortest path from s_i to t_i for $i = 1, 2, \ldots, k$, if they exist.

Note that $\mathbb{R}_+$ denotes the set of non-negative real numbers. We can consider both directed and undirected variants of this problem, which we call the *directed k-DSPP* and the *undirected k-DSPP*, respectively. For each problem, we can consider vertex-disjoint and edge-disjoint versions. If the length of each edge is equal to zero, then these problems amount to the directed or the undirected version of the k disjoint paths problem. With this observation, most hardness results on the k disjoint paths problem can be extended to the directed (or undirected) k-DSPP. In particular, since the k disjoint paths problem in digraphs is NP-hard even when $k = 2$ [6], almost all variants of the directed k-DSPP are hard.

Only few positive results are known for k-DSPP. An important positive result is a polynomial time algorithm of Eilam-Tzoreff [5] for the undirected 2-DSPP, in which the length of each edge is positive. It is interesting to note that the algorithm in [5] is completely different from the algorithms for the 2 disjoint paths problem in [17, 19, 22]. This means that properties or tractability of k-DSPP will be different from those of the k disjoint paths problem by assuming that the length of each edge is positive. This fact motivates us to study polynomial solvability of the directed k-DSPP under this assumption. Indeed, for the case when k is a fixed constant and the length of each edge is positive, polynomial solvability of the directed k-DSPP was posed as an open problem in [5].

1.2 Related work

There are many results on the problem in which we find k disjoint paths minimizing a given objective function. Such a problem is sometimes called the *shortest disjoint paths problem*. A natural objective function is the total length of the paths. That is, the aim of the problem is to find disjoint paths $P_1, \ldots, P_k$ that minimize $\sum_i \ell(P_i)$ when we are given a length function $\ell : E \to \mathbb{R}_+$, which we call the *min-sum k disjoint paths problem*. Here, $\ell(P_i)$ denotes the length of P_i. We note that a solution of the k disjoint shortest paths problem must be an optimal solution of the corresponding min-sum k disjoint paths problem, which shows that if we can solve the min-sum k disjoint paths problem, then we can also solve the k disjoint shortest paths problem. Another objective function is the length of the longest path. That is, the aim of the problem is to find disjoint paths $P_1, \ldots, P_k$ that minimize $\max_i \ell(P_i)$, which we call the *min-max k disjoint paths problem*.

Table 1 Results on the k disjoint paths problem and the k-DSPP. In the results with $(*)$, we assume that the length of each edge is positive.

	Conditions	Disjoint Paths	Disjoint Shortest Paths
$k = 2$	undirected	P [17, 19, 22]	P [5] $(*)$
	directed	NP-hard [6]	NP-hard (Proposition 1)
			P (Theorem 2) $(*)$
k: fixed	undirected	P [14]	OPEN
	planar, vertex-disjoint		P (Corollary 11)
	planar, edge-disjoint		P (Theorem 5)
	directed	NP-hard [6]	OPEN $(*)$ / NP-hard
	planar, vertex-disjoint	P [16]	P (Theorem 4)
	planar, edge-disjoint	OPEN	OPEN
	acyclic	P [6]	P (Proposition 10)
k: general	undirected/directed	NP-hard [9]	NP-hard

Since the min-sum or min-max k disjoint paths problem is a generalization of the k disjoint paths problem, hardness results on the k disjoint paths problem can be extended to the optimization problem. See [11] for classical results on the min-sum and min-max k disjoint paths problems. We now describe several positive results on the min-sum k disjoint paths problem. Colin de Verdière and Schrijver [4] presented a polynomial time algorithm for the case when the input digraph (or graph) is planar, $s_1, \ldots, s_k$ are on the boundary of a common face, and $t_1, \ldots, t_k$ are on the boundary of another face. Kobayashi and Sommer [11] gave a polynomial time algorithm for the case when the graph is planar, $k = 2$, and the terminals are on at most two faces. Borradaile et al. [2] gave a polynomial time algorithm for the case when the graph is planar, the terminals are ordered nicely on a common face. Björklund and Husfeldt [1] gave a randomized polynomial time algorithm for the case when $k = 2$ and each edge has a unit length, which is based on interesting algebraic techniques. This result was recently generalized to the case with two terminal pairs by Hirai and Namba [8].

1.3 Our results

In this subsection, we describe our results, which are summarized in Table 1.

As mentioned in Section 1.1, it is not difficult see that the directed k-DSPP is NP-hard even when $k = 2$ if the length of each edge can be zero.

▶ **Proposition 1.** *Both vertex-disjoint and edge-disjoint versions of the directed k-DSPP are NP-hard even when $k = 2$.*

Proof. Suppose that the length of each edge is equal to zero. In this case, since any path is a shortest path, the directed k-DSPP is equivalent to finding two vertex-disjoint (or edge-disjoint) paths P_1 and P_2 such that P_i is from s_i to t_i. This problem is known to be NP-hard [6], and hence the directed k-DSPP is NP-hard even when $k = 2$. ◀

The main result of this paper is to show that the directed k-DSPP can be solved in polynomial time when the length of each dicycle (directed cycle) is positive and $k = 2$.

▶ **Theorem 2.** *If the length of each dicycle is positive, both vertex-disjoint and edge-disjoint versions of the directed 2-DSPP can be solved in polynomial time. In particular, the directed 2-DSPP can be solved in polynomial time if each edge has a positive length.*

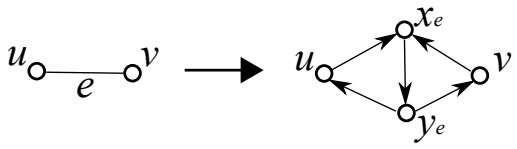


Figure 1 Reduction to the directed case.

The proof of this theorem is given in Section 3. It is posed as an open problem by Eilam-Tzoreff [5] to determine whether or not the directed k-DSPP can be solved in polynomial time when each edge has a positive length and k is a fixed constant. Theorem 2 answers this problem affirmatively for the case of $k = 2$. It is interesting to note that the assumption on the edge length affects the polynomial solvability of the problem as we can see in Proposition 1 and Theorem 2. We also note that a polynomial time algorithm for the undirected version can be derived from Theorem 2, that is, we obtain an alternative elementary proof for the following result.

▶ **Corollary 3** (Eilam-Tzoreff [5]). *If each edge has a positive length, both vertex-disjoint and edge-disjoint versions of the undirected 2-DSPP can be solved in polynomial time.*

Proof. Suppose we are given an instance of the undirected 2-DSPP in which $\ell(e) > 0$ for every $e \in E$. Replace each edge $e = uv$ with two new vertices x_e, y_e and five new directed edges $ux_e, vx_e,\ x_ey_e, y_eu, y_ev$ (see Fig. 1). Define a new length function ℓ' by $\ell'(ux_e) = \ell'(vx_e) = \ell'(x_ey_e) = \ell'(y_eu) = \ell'(y_ev) = \frac{\ell(uv)}{3}$. Then, each edge has a positive length in the obtained digraph. In this way, we can reduce the undirected 2-DSPP to the directed 2-DSPP, which shows the corollary by Theorem 2. ◀

Theorem 2 can be extended to the case when the input digraph contains two terminal pairs and k is a fixed constant, which is discussed in Section 4.

We also discuss the case when the input (di)graph is restricted to be planar in Section 5. We first show that the vertex-disjoint version of the directed k-DSPP can be solved in polynomial time in planar digraphs.

▶ **Theorem 4.** *If k is a fixed constant and the input digraph is planar, the vertex-disjoint version of the directed k-DSPP can be solved in polynomial time.*

The proof is given in Section 5. Our proof is based on the reduction technique used in the proof of Theorem 2 and the algorithm for the disjoint paths problem in planar digraphs proposed in [16]. Note that this result implies that we can also solve the undirected version in polynomial time. Since Schrijver's algorithm for the disjoint paths problem [16] works only for the vertex-disjoint case, the proof of Theorem 4 cannot be extended to the edge-disjoint case directly. However, when the graph is undirected, we can show the following theorem, whose proof is given in Section 5.

▶ **Theorem 5.** *If k is a fixed constant and the input graph is planar, the edge-disjoint version of the undirected k-DSPP can be solved in polynomial time.*

2 Preliminary

For a digraph $G = (V, E)$, a directed edge from u to v is denoted by uv. For a directed edge e in G, the head and the tail of e are denoted by $\mathrm{head}_G(e)$ and $\mathrm{tail}_G(e)$, respectively, that is,

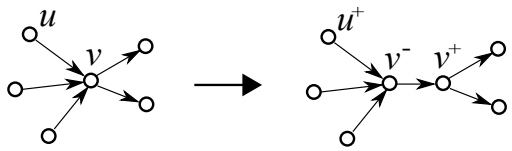


Figure 2 Reduction to the edge-disjoint version.

e is a directed edge from $\mathrm{tail}_G(e)$ to $\mathrm{head}_G(e)$. A *dipath* (or a *directed path*) is a sequence $(v_0, e_1, v_1, e_2, \ldots, e_p, v_p)$ such that $v_0, v_1, \ldots, v_p \in V$ are distinct vertices and $e_i = v_{i-1}v_i \in E$ for each i. If $v_0 = v_p$ in the definition of a dipath, the sequence is called a *dicycle* (or a *directed cycle*). If no confusion may arise, a dicycle, a dipath, and a directed edge are simply called a cycle, a path, and an edge, respectively. For a dipath, a dicycle, or a subgraph Q, its vertex set and edge set are denoted by $V(Q)$ and $E(Q)$, respectively. For a length function $\ell : E \to \mathbb{R}_+$ and for an edge set $F \subseteq E$, we denote $\ell(F) = \sum_{e \in F} \ell(e)$. For a dipath or a dicycle Q, we identify Q with its edge set, and $\ell(E(Q))$ is simply denoted by $\ell(Q)$.

3 Proof of Theorem 2

In this section, we give a proof of Theorem 2, that is, we show that the directed 2-DSPP can be solved in polynomial time if the length of each dicycle is positive. To solve this problem, we will efficiently reduce it to a set of 2 disjoint paths problem in acyclic digraphs. Although the original digraph is not necessarily acyclic, we decompose the digraph into smaller subgraphs and modify each subgraph to an acyclic digraph.

We first note that the vertex-disjoint version of the directed 2-DSPP can be reduced to the edge-disjoint version of the directed 2-DSPP by the following procedure: replace each vertex v with two vertices v^+ and v^-, replace each edge uv with an edge u^+v^- of the same length, and add an edge v^-v^+ of length zero for each v (see Fig. 2). Therefore, it suffices to give a polynomial time algorithm for the edge-disjoint version of the problem.

Suppose we have an instance of the edge-disjoint version of the directed 2-DSPP in which each dicycle is of positive length. For $i = 1, 2$, let $E_i \subseteq E$ be the set of edges that are contained in some shortest path from s_i to t_i. By the definition, an s_i-t_i path is a shortest s_i-t_i path if and only if it consists of edges in E_i. Note that we can compute E_i in polynomial time as follows. We first apply a shortest path algorithm (e.g., Dijkstra's algorithm) and obtain the distance $d_i(v)$ from s_i to v for every $v \in V$. Let $E_i' \subseteq E$ be the set of all the edges uv with $d_i(v) - d_i(u) = \ell(uv)$. Then, $\{uv \in E_i' \mid E_i' \text{ contains a } v\text{-}t_i \text{ path}\}$ is the desired set E_i. With this observation, the edge-disjoint version of the directed 2-DSPP can be reduced to the following problem: given a digraph $G = (V, E)$, subsets $E_1, E_2 \subseteq E$, and two pairs of vertices (s_1, t_1) and (s_2, t_2) in G, find edge-disjoint paths P_1 and P_2 such that $E(P_i) \subseteq E_i$ and P_i is a path from s_i to t_i for $i = 1, 2$. We now show some properties of E_i.

▶ **Lemma 6.** *The edge set E_i forms no dicycle for $i = 1, 2$.*

Proof. Assume that E_i forms a dicycle C. By the definition of d_i and E_i, $d_i(v) - d_i(u) = \ell(uv)$ for each $uv \in E(C)$. This shows that $\ell(C) = \sum_{uv \in E(C)} \ell(uv) = \sum_{uv \in E(C)} (d_i(v) - d_i(u)) = 0$, which contradicts that the length of each dicycle is positive. ◀

For a set F of directed edges, let $\overline{F}$ be the set of directed edges obtained from F by reversing all the edges, that is, $\overline{F} = \{vu \mid uv \in F\}$. Then, we have the following lemma.

▶ **Lemma 7.** *Suppose that C is a dicycle in $E_1 \cup \overline{E}_2$. Then, $E_1 \cap E(C) \subseteq E_2$ and $E_2 \cap \overline{E(C)} \subseteq E_1$.*

Proof. Since C is a dicycle in $E_1 \cup \overline{E}_2$, it can be decomposed into subpaths $P_1, \overline{Q_1}, P_2, \overline{Q_2}, \dots, P_r, \overline{Q_r}$ such that P_i is a dipath from u_i to v_i with $E(P_i) \subseteq E_1$ and Q_i is a dipath from u_{i+1} to v_i with $E(Q_i) \subseteq E_2$ for $i = 1, \dots, r$, where we denote $u_{r+1} = u_1$. By the definition of d_1 and E_1, $d_1(v_i) - d_1(u_i) = \ell(P_i)$ and $d_1(v_i) - d_1(u_{i+1}) \leq \ell(Q_i)$ for $i = 1, \dots, r$. By combining them, we obtain $\sum_{i=1}^{r} \ell(P_i) \leq \sum_{i=1}^{r} \ell(Q_i)$. Similarly, by the definition of d_2 and E_2, $d_2(v_i) - d_2(u_i) \leq \ell(P_i)$ and $d_2(v_i) - d_2(u_{i+1}) = \ell(Q_i)$ for $i = 1, \dots, r$, which shows that $\sum_{i=1}^{r} \ell(P_i) \geq \sum_{i=1}^{r} \ell(Q_i)$. Therefore, $\sum_{i=1}^{r} \ell(P_i) = \sum_{i=1}^{r} \ell(Q_i)$ and all the above inequalities are tight. That is, $d_1(v_i) - d_1(u_{i+1}) = \ell(Q_i)$ and $d_2(v_i) - d_2(u_i) = \ell(P_i)$ for $i = 1, \dots, r$, which shows that $E(Q_i) \subseteq E_1'$ and $E(P_i) \subseteq E_2'$. Since $E(P_i) \subseteq E_1$ for $i = 1, \dots, r$, there is a v_i-t_1 path in E_1'. This implies that E_1' contains a v-t_1 path for any $v \in V(Q_i)$, and hence $E(Q_i) \subseteq E_1$. Similarly, since $E(Q_i) \subseteq E_2$ for $i = 1, \dots, r$, there is a v_i-t_2 path in E_2', which shows that $E(P_i) \subseteq E_2$. ◀

We add four vertices s_1', s_2', t_1', and t_2', and four edges $s_1's_1$, $s_2's_2$, t_1t_1', and t_2t_2'. We update $E_i \leftarrow E_i \cup \{s_i's_i, t_it_i'\}$ for $i = 1, 2$. Then, a path from s_i to t_i is corresponding to a path whose first and last edges are $s_i's_i$ and t_it_i', respectively. By using this correspondence, we can rephrase the problem to the following: find edge-disjoint paths P_1 and P_2 such that $E(P_i) \subseteq E_i$ and P_i is a path whose first and last edges are $s_i's_i$ and t_it_i', respectively.

Let $E_0 := E_1 \cap E_2$, $E_1^* = E_1 \setminus E_0$, and $E_2^* = E_2 \setminus E_0$. We remove all the edges in $E \setminus (E_1 \cup E_2)$ from G, contract all the edges in E_0, and reverse all the edges in E_2^*. Then, we obtain a digraph $G^* = (V^*, E^*)$. Let $V_0 \subseteq V^*$ be the set of all the vertices in V^* that are newly created by contracting E_0. In other words, $V^* \setminus V_0 \subseteq V$ is the set of all original vertices. For $v \in V_0$, let G_v be the subgraph of $G - (E \setminus (E_1 \cup E_2))$ induced by the vertex set corresponding to v. For any edge e in G_v, by the definition of G_v, either $e \in E_0$ or there exist edges $f_1, f_2, \dots, f_r \in E_0$ such that $e, f_1, f_2, \dots, f_r$ form a cycle when we ignore the direction of the edges. In the latter case, these edges induce a dicycle C in $E_1 \cup \overline{E}_2$, which shows that $e \in E_0$ by Lemma 7. Thus, every edge in G_v is in E_0, which implies that we can identify E^* with $E_1^* \cup \overline{E_2^*}$. Furthermore, since every edge in G_v is in E_0, G_v is an acyclic digraph by Lemma 6.

We can also see that, by Lemma 7, G^* is an acyclic digraph. In what follows, roughly, we find two disjoint paths in G^* such that one is from s_1' to t_1' and the other is from t_2' to s_2'. Our algorithm is based on the algorithm for finding disjoint paths in digraphs proposed in [6].

We define a new digraph $\mathcal{G}$ whose vertex set is $W = E_1^* \times \overline{E_2^*}$ as follows. For $(e_1, e_2), (e_1', e_2') \in W$, $\mathcal{G}$ has a directed edge from (e_1, e_2) to (e_1', e_2') if one of the following holds.

- $e_1' = e_1$, $\mathrm{head}_{G^*}(e_2) = \mathrm{tail}_{G^*}(e_2') =: v$, and there is no path in G^* from $\mathrm{head}_{G^*}(e_1)$ to v. Furthermore, if $v \in V_0$, then G_v contains a path from $\mathrm{tail}_G(e_2')$ to $\mathrm{head}_G(e_2)$.
- $e_2' = e_2$, $\mathrm{head}_{G^*}(e_1) = \mathrm{tail}_{G^*}(e_1') =: v$, and there is no path in G^* from $\mathrm{head}_{G^*}(e_2)$ to v. Furthermore, if $v \in V_0$, then G_v contains a path from $\mathrm{head}_G(e_1)$ to $\mathrm{tail}_G(e_1')$.
- $\mathrm{head}_{G^*}(e_1) = \mathrm{head}_{G^*}(e_2) = \mathrm{tail}_{G^*}(e_1') = \mathrm{tail}_{G^*}(e_2') =: v$. Furthermore, if $v \in V_0$, then G_v contains two edge-disjoint paths such that one is from $\mathrm{head}_G(e_1)$ to $\mathrm{tail}_G(e_1')$ and the other is from $\mathrm{tail}_G(e_2')$ to $\mathrm{head}_G(e_2)$.

To construct $\mathcal{G}$, it suffices to solve the two disjoint paths problem in each acyclic digraph G_v, which can be done in polynomial time by [6]. We now show that we can solve the edge-disjoint version of the directed 2-DSPP by finding a path in $\mathcal{G}$ from $(s_1's_1, t_2't_2)$ to (t_1t_1', s_2s_2').

▶ **Lemma 8.** *There is a path in $\mathcal{G}$ from (s'_1s_1, t'_2t_2) to $(t_1t'_1, s_2s'_2)$ if and only if G has two edge-disjoint paths P_1 and P_2 such that P_i is from s_i to t_i and $E(P_i) \subseteq E_i$ for $i = 1, 2$.*

Proof. *Sufficiency* ("if" part). Suppose that G has two edge-disjoint paths P_1 and P_2 such that P_i is from s_i to t_i and $E(P_i) \subseteq E_i$ for $i = 1, 2$. $E(P_1) \setminus E_0$ forms a path P_1^* from s_1 to t_1 in G^*, and $\overline{E(P_2) \setminus E_0}$ forms a path P_2^* from t_2 to s_2 in G^*. Suppose that P_1^* traverses edges $e_1^1, e_1^2, \ldots, e_1^p$ in this order, and let $e_1^0 := s'_1s_1$ and $e_1^{p+1} := t_1t'_1$. Similarly, suppose that P_2^* traverses edges $e_2^1, e_2^2, \ldots, e_2^q$ in this order, and let $e_2^0 := t'_2t_2$ and $e_2^{q+1} := s_2s'_2$. It is obvious that $e_1^i \in E_1^*$ for $i = 0, 1, \ldots, p+1$ and $e_2^j \in \overline{E_2^*}$ for $j = 0, 1, \ldots, q+1$. Since G^* is acyclic, for any $i = 0, 1, \ldots, p+1$ and for any $j = 0, 1, \ldots, q+1$, at least one of the following holds.

(1) There is no dipath in G^* from $\mathrm{head}_{G^*}(e_1^i)$ to $\mathrm{head}_{G^*}(e_2^j)$.

(2) There is no dipath in G^* from $\mathrm{head}_{G^*}(e_2^j)$ to $\mathrm{head}_{G^*}(e_1^i)$.

(3) $\mathrm{head}_{G^*}(e_1^i) = \mathrm{head}_{G^*}(e_2^j)$.

For each case, we obtain the following by the definition of the edge set of $\mathcal{G}$.

- If (1) holds and $j \neq q+1$, then $\mathcal{G}$ has an edge from (e_1^i, e_2^j) to (e_1^i, e_2^{j+1}). Note that if $v := \mathrm{head}_{G^*}(e_2^j) \in V_0$, then $E(P_2) \cap E(G_v)$ forms a path in G_v from $\mathrm{tail}_G(e_2^{j+1})$ to $\mathrm{head}_G(e_2^j)$.
- If (2) holds and $i \neq p+1$, then $\mathcal{G}$ has an edge from (e_1^i, e_2^j) to (e_1^{i+1}, e_2^j). Note that if $v := \mathrm{head}_{G^*}(e_1^i) \in V_0$, then $E(P_1) \cap E(G_v)$ forms a path in G_v from $\mathrm{head}_G(e_1^i)$ to $\mathrm{tail}_G(e_1^{i+1})$.
- If (3) holds, then $\mathcal{G}$ has an edge from (e_1^i, e_2^j) to (e_1^{i+1}, e_2^{j+1}). Note that if $v := \mathrm{head}_{G^*}(e_1^i) = \mathrm{head}_{G^*}(e_2^j) \in V_0$, then $E(P_1) \cap E(G_v)$ and $E(P_2) \cap E(G_v)$ form two edge-disjoint paths in G_v such that one is from $\mathrm{head}_G(e_1^i)$ to $\mathrm{tail}_G(e_1^{i+1})$ and the other is from $\mathrm{tail}_G(e_2^{j+1})$ to $\mathrm{head}_G(e_2^j)$.

By observing that (1) holds if $i = p+1$ and (2) holds if $j = q+1$, we can see that $\mathcal{G}$ has an edge from (e_1^i, e_2^j) to (e_1^i, e_2^{j+1}), (e_1^{i+1}, e_2^j), or (e_1^{i+1}, e_2^{j+1}) unless $(i, j) = (p+1, q+1)$. We begin with $(i, j) = (0, 0)$ and find an edge leaving (e_1^i, e_2^j) in $\mathcal{G}$ as above, repeatedly. Then, we obtain a path in $\mathcal{G}$ from $(e_1^0, e_2^0) = (s'_1s_1, t'_2t_2)$ to $(e_1^{p+1}, e_2^{q+1}) = (t_1t'_1, s_2s'_2)$, which shows the sufficiency.

Necessity ("only if" part). Suppose that there is a path in $\mathcal{G}$ from $(f_1^0, f_2^0) := (s'_1s_1, t'_2t_2)$ to $(f_1^r, f_2^r) := (t_1t'_1, s_2s'_2)$ that traverses vertices (f_1^0, f_2^0), (f_1^1, f_2^1), $\ldots$, (f_1^r, f_2^r) of $\mathcal{G}$ in this order. In this proof, we regard a path in G as a sequence of edges, and the concatenation of two paths P and Q is denoted by $P \cdot Q$. We define two paths P_1 and P_2 as follows.

1. Set $P_1 = P_2 = \emptyset$.
2. For $i = 0, 1, 2, \ldots, r$, we update P_i as follows.
   - Suppose that $f_1^{i+1} = f_1^i$, $\mathrm{head}_{G^*}(f_2^i) = \mathrm{tail}_{G^*}(f_2^{i+1}) =: v$, and there is no dipath in G^* from $\mathrm{head}_{G^*}(f_1^i)$ to v. In this case, let Q be the path in G_v from $\mathrm{tail}_G(f_2^{i+1})$ to $\mathrm{head}_G(f_2^i)$ if $v \in V_0$ and let $Q = \emptyset$ if $v \notin V_0$. Then, update P_2 as $P_2 \leftarrow f_2^{i+1} \cdot Q \cdot P_2$.
   - Suppose that $f_2^{i+1} = f_2^i$, $\mathrm{head}_{G^*}(f_1^i) = \mathrm{tail}_{G^*}(f_1^{i+1}) =: v$, and there is no dipath in G^* from $\mathrm{head}_{G^*}(f_2^i)$ to v. In this case, let Q be the path in G_v from $\mathrm{head}_G(f_1^i)$ to $\mathrm{tail}_G(f_1^{i+1})$ if $v \in V_0$ and let $Q = \emptyset$ if $v \notin V_0$. Then, update P_1 as $P_1 \leftarrow P_1 \cdot Q \cdot f_1^{i+1}$.
   - Suppose that $\mathrm{head}_{G^*}(f_1^i) = \mathrm{head}_{G^*}(f_2^i) = \mathrm{tail}_{G^*}(f_1^{i+1}) = \mathrm{tail}_{G^*}(f_2^{i+1}) =: v$. In this case, if $v \in V_0$, then G_v contains two edge-disjoint paths Q_1 and Q_2 such that Q_1 is from $\mathrm{head}_G(f_1^i)$ to $\mathrm{tail}_G(f_1^{i+1})$ and Q_2 is from $\mathrm{tail}_G(f_2^{i+1})$ to $\mathrm{head}_G(f_2^i)$. Let $Q_1 = Q_2 = \emptyset$ if $v \notin V_0$. Then, update P_1 and P_2 as $P_1 \leftarrow P_1 \cdot Q_1 \cdot f_1^{i+1}$ and $P_2 \leftarrow f_2^{i+1} \cdot Q_2 \cdot P_2$.

Then, P_1 and P_2 are edge-disjoint paths in G such that P_i is from s_i to t_i and $E(P_i) \subseteq E_i$ for $i = 1, 2$, which shows the necessity. ◀

Since $\mathcal{G}$ contains at most $|E|^2$ vertices, we can detect a path in $\mathcal{G}$ in polynomial time. Thus, Lemma 8 shows that the directed 2-DSPP can be solved in polynomial time.

We note that the most time consuming part of our algorithm is to construct $\mathcal{G}$. We have already seen that, for each pair of vertices in $\mathcal{G}$, the existence of an edge between them can be checked by solving the two disjoint paths problem in an acyclic digraph. Thus, in a naive implementation of our algorithm, we solve the two disjoint paths problem in an acyclic digraph $O(|E|^4)$ times. If we adopt the algorithm of [18] for the two disjoint paths problem, which runs in $O(|V||E|)$ time, the total running time of our algorithm is $O(|V||E|^5)$. Note that a faster algorithm for the two disjoint paths problem is proposed in [21]. Although the above estimation of the running time is very rough, we do not discuss its improvement in this paper, since we focus on the polynomial solvability of the problem.

4 Disjoint Shortest Paths with Two Terminal Pairs

In this section, we extend Theorem 2 to the case when the digraph has two terminal pairs. More precisely, for fixed integers k_1 and k_2, we consider the following problem and give a polynomial time algorithm for it.

Directed Disjoint Shortest Paths Problem with Two Terminal Pairs.

Input. A digraph $G = (V, E)$ with a length function $l : E \to \mathbb{R}_+$, two pairs of vertices (s_1, t_1) and (s_2, t_2) in G.

Find. Internally-vertex-disjoint (or edge-disjoint) paths $P_1^1, \ldots, P_{k_1}^1, P_1^2, \ldots, P_{k_2}^2$ such that P_j^i is a shortest path from s_i to t_i for $i = 1, 2$ and $j = 1, 2, \ldots, k_i$.

Our result is formally stated as follows.

▶ **Theorem 9.** *Let k_1 and k_2 be fixed integers. If the length of each dicycle is positive, both internally-vertex-disjoint and edge-disjoint versions of the directed disjoint shortest paths problem with two terminal pairs can be solved in polynomial time.*

Proof. In the same way as the proof of Theorem 2, it suffices to give an algorithm for the edge-disjoint version. For $i = 1, 2$, let $E_i \subseteq E$ be the set of all the edges that are contained in some shortest path from s_i to t_i, which satisfy Lemmas 6 and 7. Then, an s_i-t_i path is a shortest s_i-t_i path if and only if it consists of edges in E_i.

We add $2(k_1 + k_2)$ vertices $s'_{1,1}, \ldots, s'_{1,k_1}, s'_{2,1}, \ldots, s'_{2,k_2}, t'_{1,1}, \ldots, t'_{1,k_1}, t'_{2,1}, \ldots, t'_{2,k_2}$, and $2(k_1 + k_2)$ edges $s'_{1,j}s_1$ and $t_1, t'_{1,j}$ for $j = 1, \ldots, k_1$, and $s'_{2,j}s_2$ and $t_2, t'_{2,j}$ for $j = 1, \ldots, k_2$. We update $E_i \leftarrow E_i \cup \{s'_{i,j}s_i, t_it'_{i,j} \mid j = 1, \ldots, k_i\}$ for $i = 1, 2$. Then, we can rephrase the problem to the following: find edge-disjoint paths $P_1^1, \ldots, P_{k_1}^1, P_1^2, \ldots, P_{k_2}^2$ such that $E(P_j^i) \subseteq E_i$ and P_j^i is a path whose first and last edges are $s'_{i,j}s_i$ and $t_it'_{i,j}$ for each i and j.

Define E_0, E_1^*, E_2^*, G^*, V_0, and G_v for $v \in V_0$ in the same way as the proof of Theorem 2. Let $S_0 := \{(i, j) \mid i = 1, 2,\ j = 1, \ldots, k_i\}$. We define a digraph $\mathcal{G}$ whose vertex set is $W = (E_1^*)^{k_1} \times (\overline{E_2^*})^{k_2}$ as follows. For $(e_1^1, \ldots, e_{k_1}^1, e_1^2, \ldots, e_{k_2}^2) \in W$ and $(f_1^1, \ldots, f_{k_1}^1, f_1^2, \ldots, f_{k_2}^2) \in W$, $\mathcal{G}$ has an edge from $(e_1^1, \ldots, e_{k_1}^1, e_1^2, \ldots, e_{k_2}^2)$ to $(f_1^1, \ldots, f_{k_1}^1, f_1^2, \ldots, f_{k_2}^2)$ if there exists a non-empty set $S \subseteq S_0$ and a vertex $v \in V^*$ such that

> $\mathrm{head}_{G^*}(e_j^i) = \mathrm{tail}_{G^*}(f_j^i) = v$ for $(i, j) \in S$, and $e_j^i = f_j^i$ and there is no path in G^* from $\mathrm{head}_{G^*}(e_j^i)$ to v for $(i, j) \in S_0 \setminus S$. Furthermore, if $v \in V_0$, then G_v contains $|S|$ edges disjoint paths such that each path is from $\mathrm{head}_G(e_j^1)$ to $\mathrm{tail}_G(f_j^1)$ with $(1, j) \in S$ or from $\mathrm{tail}_G(f_j^2)$ to $\mathrm{head}_G(e_j^2)$ with $(2, j) \in S$.

Note that this is a generalization of the construction in the proof of Theorem 2. To construct $\mathcal{G}$, it suffices to solve the disjoint paths problem with at most k terminal pairs in each acyclic digraph G_v, which can be done in polynomial time by [6].

In the same way as Lemma 8, there is a path in $\mathcal{G}$ from $(s'_{1,1}s_1, \ldots, s'_{1,k_1}s_1, t'_{2,1}t_2, \ldots, t'_{2,k_2}t_2)$ to $(t_1t'_{1,1}, \ldots, t_1t'_{1,k_1}, s_2s'_{2,1}, \ldots, s_2s'_{2,k_2})$ if and only if G has $k_1 + k_2$ edge-disjoint paths $P^1_1, \ldots, P^1_{k_1}$, $P^2_1, \ldots, P^2_{k_2}$ such that $E(P^i_j) \subseteq E_i$ and P^i_j is a path whose first and last edges are $s'_{i,j}s_i$ and $t_it'_{i,j}$ for each i and j. Since $\mathcal{G}$ has a polynomial size in $|V|$, we can detect a path in $\mathcal{G}$ from $(s'_{1,1}s_1, \ldots, s'_{1,k_1}s_1, t'_{2,1}t_2, \ldots, t'_{2,k_2}t_2)$ to $(t_1t'_{1,1}, \ldots, t_1t'_{1,k_1}, s_2s'_{2,1}, \ldots, s_2s'_{2,k_2})$ in polynomial time. Hence, we can solve the directed disjoint shortest paths problem with two terminal pairs in polynomial time. ◀

In order to construct $\mathcal{G}$, we solve the k disjoint paths problem in an acyclic digraph $|E|^{O(k)}$ times. Since the k disjoint paths problem in an acyclic digraph can be solved in $|E|^{O(k)}$ time [6], the total running time of our algorithm is $|E|^{O(k)}$, which is also denoted by $|V|^{O(k)}$.

We note that, by using the same argument as the proofs of Theorems 2 and 9, we can show that the directed k-DSPP in acyclic digraphs can be solved in polynomial time if k is a fixed constant.

▶ **Proposition 10.** *If k is a fixed constant and the input graph is acyclic, both vertex-disjoint and edge-disjoint versions of the directed k-DSPP can be solved in polynomial time.*

Proof. It suffices to consider the edge-disjoint version. For $i = 1, \ldots, k$, let $E_i \subseteq E$ be the set of all the edges that are contained in some shortest path from s_i to t_i. Then, an s_i-t_i path is a shortest s_i-t_i path if and only if it consists of edges in E_i. We add $2k$ vertices $s'_1, \ldots, s'_k, t'_1, \ldots, t'_k$, and $2k$ edges $s'_1s_1, \ldots, s'_ks_k, t_1t'_1, \ldots, t_kt'_k$, and update $E_i \leftarrow E_i \cup \{s'_is_i, t_it'_i\}$ for $i = 1, \ldots, k$. The obtained acyclic digraph is also denoted by G. Then, the directed k-DSPP is equivalent to finding k edge-disjoint paths $P_1, \ldots, P_k$ such that $E(P_i) \subseteq E_i$ and P_i is a path whose first and last edges are s'_is_i and $t_it'_i$, respectively.

We define a digraph $\mathcal{G}$ whose vertex set is $W = E_1 \times \cdots \times E_k$ as follows. For $(e_1, \ldots, e_k) \in W$ and $(f_1, \ldots, f_k) \in W$, $\mathcal{G}$ has an edge from $(e_1, \ldots, e_k)$ to $(f_1, \ldots, f_k)$ if there exists an index i such that $e_j = f_j$ for $j \in \{1, \ldots, k\} \setminus \{i\}$, $\mathrm{head}_G(e_i) = \mathrm{tail}_G(f_i) =: v$, and there is no path in G from $\mathrm{head}_G(e_j)$ to v for $j \in \{1, \ldots, k\} \setminus \{i\}$.

In the same way as Lemma 8, there is a path in $\mathcal{G}$ from $(s'_1s_1, \ldots, s'_ks_k)$ to $(t'_1t_1, \ldots, t'_kt_k)$ if and only if G has k edge-disjoint paths $P_1, \ldots, P_k$ such that $E(P_i) \subseteq E_i$ and P_i is a path whose first and last edges are s'_is_i and $t_it'_i$ for each i. Since $\mathcal{G}$ has $|V|^{O(k)}$ vertices, a path in $\mathcal{G}$ from $(s'_1s_1, \ldots, s'_ks_k)$ to $(t'_1t_1, \ldots, t'_kt_k)$ can be detected in $|V|^{O(k)}$ time. ◀

5 Planar Cases

In this section, we discuss the case when the input (di)graph is planar. We first give a proof of Theorem 4, that is, we show that the vertex-disjoint version of the directed k-DSPP can be solved in polynomial time if k is a fixed constant and the input digraph is planar.

Proof of Theorem 4. For $i = 1, \ldots, k$, let $E_i \subseteq E$ be the set of all the edges that are contained in some shortest path from s_i to t_i. Since an s_i-t_i path is a shortest s_i-t_i path if and only if it consists of edges in E_i, the directed k-DSPP in a planar digraph can be reduced to the following problem: given a planar digraph $G = (V, E)$, subsets $E_1, \ldots, E_k \subseteq E$, and k pairs of vertices $(s_1, t_1), \ldots, (s_k, t_k)$ in G, find vertex-disjoint paths $P_1, \ldots, P_k$ such that $E(P_i) \subseteq E_i$ and P_i is a path from s_i to t_i for $i = 1, \ldots, k$. It is shown in [16] that this

problem can be solved in $|V|^{O(k)}$ time if G is planar. Therefore, for fixed k, the directed k-DSPP can be solved in polynomial time if the input digraph is planar. ◀

By replacing each edge with two parallel edges in opposite directions, we can reduce the undirected version to the directed version. Hence, Theorem 4 implies the following as a corollary.

▶ **Corollary 11.** *If the input graph is planar, the vertex-disjoint version of the undirected k-DSPP can be solved in $|V|^{O(k)}$ time.*

We note that Schrijver's algorithm for finding disjoint paths $P_1, \ldots, P_k$ with $E(P_i) \subseteq E_i$ [16] works only for the vertex-disjoint case, and no polynomial time algorithm is known for the edge-disjoint version of this problem. However, when the graph is undirected, the edge-disjoint version of k-DSPP can be solved in polynomial time (Theorem 5). To prove Theorem 5, we first give a polynomial time algorithm for the case when the obtained paths do not cross each other. Here, we say that two edge-disjoint paths P and Q in a planar graph *cross* at a vertex v if P contains two edges e_1 and e_2 and Q contains two edges f_1 and f_2 such that e_1, f_1, e_2, and f_2 are incident to v clockwise in this order. The problem is formally described as follows.

Undirected k Edge-disjoint Non-crossing Shortest Paths Problem

Input. A planar graph $G = (V, E)$ with a length function $\ell : E \to \mathbb{R}_+$ and k pairs of vertices $(s_1, t_1), \ldots, (s_k, t_k)$ in G.

Find. Pairwise edge-disjoint paths $P_1, \ldots, P_k$ such that P_i is a shortest path from s_i to t_i for $i = 1, 2, \ldots, k$ and they do not cross each other, if they exist.

▶ **Proposition 12.** *The undirected k edge-disjoint non-crossing shortest paths problem can be solved in $|V|^{O(k)}$ time.*

Proof. We first reduce the problem to the case when each terminal is of degree one. Suppose we are given an instance of the the undirected k edge-disjoint non-crossing shortest paths problem. For $i = 1, \ldots, k$, we guess the first and last edges of P_i, say s_iu_i and v_it_i. Then, replace edge s_iu_i with a new vertex u'_i and a new edge u'_iu_i of length $\ell(s_iu_i)$, and define a new terminal $s'_i = u'_i$. Similarly, replace edge v_it_i with a new vertex v'_i and a new edge $v_iv'_i$ of length $\ell(v_it_i)$, and define a new terminal $t'_i = v'_i$. In the obtained graph, we consider the undirected k edge-disjoint non-crossing shortest paths problem with terminal pairs $(s'_1, t'_1), \ldots, (s'_k, t'_k)$. Note that each terminal is of degree one in the obtained instance. Since the number of choices of s_iu_i and v_it_i is at most $|V|^{O(k)}$, in order to solve the original instance, it suffices to solve $|V|^{O(k)}$ instances in which each terminal is of degree one.

In what follows, we give an algorithm for the case when each terminal is of degree one by using a reduction to the vertex-disjoint version of the undirected k-DSPP. Suppose that we are given an instance $G = (V, E)$, $\ell : E \to \mathbb{R}_+$, and $(s_1, t_1), \ldots, (s_k, t_k)$ of the undirected k edge-disjoint non-crossing shortest paths problem in which each terminal is of degree one. For a vertex $v \in V$ of degree at least four, let $e_1, \ldots, e_r$ be the edges that are incident to v clockwise in this order. We replace v with r vertices $w_1, \ldots, w_r$ so that each edge e_i is incident to w_i, and add r edges $w_1w_2, w_2w_3, \ldots, w_{r-1}w_r, w_rw_1$ of length zero (see Fig. 3). Note that this transformation keeps the planarity of the graph.

By applying this transformation to every vertex $v \in V$ of degree at least four, we obtain a new planar graph $G' = (V', E')$ whose maximum degree is at most three. We can easily see that the undirected k edge-disjoint non-crossing shortest paths problem in G is equivalent to that in G'. Since the maximum degree of G' is at most three and the degree of each terminal

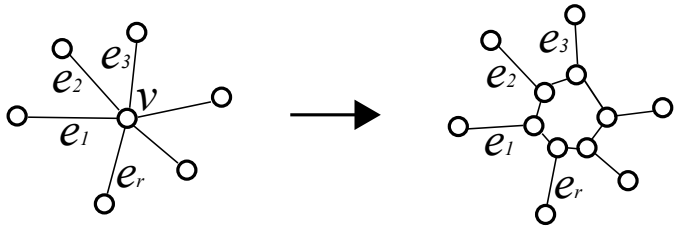


Figure 3 Reduction to the vertex-disjoint version.

is one, edge-disjoint paths in G' have to be vertex-disjoint, and hence it suffices to solve the vertex-disjoint version of the undirected k-DSPP in G'. This can be done in $|V|^{O(k)}$ time by Corollary 11, which completes the proof. ◀

We are now ready to prove Theorem 5.

Proof of Theorem 5. Suppose we are given an instance of the edge-disjoint version of the undirected k-DSPP in a planar graph. We begin with the following claim.

▶ **Claim 13.** *If there exists a solution of the edge-disjoint version of the undirected k-DSPP in a planar graph, then there exists a solution $P_1, \dots, P_k$ such that P_i and P_j cross at most once for every pair $i, j \in \{1, \dots, k\}$.*

Proof. Let $P_1, \dots, P_k$ be a solution of the edge-disjoint version of the undirected k-DSPP that minimizes the total number of crossings of the paths. We show that this solution satisfies the condition in the claim. Assume to the contrary that P_i and P_j cross at two distinct vertices u and v. Then, there exists a subpath Q_i of P_i and a subpath Q_j of P_j such that both Q_i and Q_j are paths from u to v. Since P_i is a shortest path from s_i to t_i and P_j is a shortest path from s_j to t_j, we have $\ell(Q_i) = \ell(Q_j)$. This shows that, we can obtain another solution of the edge-disjoint version of the undirected k-DSPP by replacing P_i and P_j with two paths P_i' and P_j' such that $E(P_i') = (E(P_i) \setminus E(Q_i)) \cup E(Q_j)$ and $E(P_j') = (E(P_j) \setminus E(Q_j)) \cup E(Q_i)$. We can see that the number of crossings of P_i' and P_j' is strictly smaller than that of P_i and P_j. We can also see that, for any $h \in \{1, \dots, k\} \setminus \{i, j\}$, the number of crossings of P_h and $\{P_i', P_j'\}$ is at most that of P_h and $\{P_i, P_j\}$. Therefore, the total number of crossings of the obtained solution is smaller than the original solution, which is a contradiction. ◀

Let $P_1, \dots, P_k$ be a solution of the edge-disjoint version of the undirected k-DSPP satisfying the condition in the above claim. For $i = 1, \dots, k$, by the above claim, there exist at most $k-1$ vertices $u_1^i, u_2^i, \dots, u_{r_i}^i$ such that P_i crosses another path at some u_j^i and $s_i =: u_0^i, u_1^i, u_2^i, \dots, u_{r_i}^i, u_{r_i+1}^i := t_i$ appear in this order along P_i. Then, P_i can be divided into $r_i + 1 \leq k$ subpaths $Q_1^i, \dots Q_{r_i+1}^i$, where Q_j^i is a shortest path from u_{j-1}^i to u_j^i. By the definition of Q_j^i, we can see that Q_j^i $(i = 1, \dots, k,\ j = 1, \dots, r_i + 1)$ are edge-disjoint paths and they do not cross each other.

With this observation, we can solve the edge-disjoint version of the undirected k-DSPP as follows.

Step 1. For $i = 1, \dots, k$, guess an integer $r_i \leq k - 1$ and vertices $u_1^i, u_2^i, \dots, u_{r_i}^i$.

Step 2. Find pairwise edge-disjoint paths Q_j^i $(i = 1, \dots, k,\ j = 1, \dots, r_i + 1)$ such that they do not cross each other and Q_j^i is a shortest path from u_{j-1}^i to u_j^i, where $u_0^i = s_i$ and $u_{r_i+1}^i = t_i$.

Step 3. For each i, define P_i as the concatenation of $Q_1^i, \dots, Q_{r_i+1}^i$. Check whether or not $P_1, \dots, P_k$ form a solution of the original instance.

In Step 1, the number of choices of r_i and $u_1^i, u_2^i, \ldots, u_{r_i}^i$ is at most $|V|^{O(k^2)}$. In Step 2, we can find desired edge-disjoint paths Q_j^i $(i = 1, \ldots, k,\ j = 1, \ldots, r_i + 1)$ if they exist in $|V|^{O(k^2)}$ time by Proposition 12. Note that the number of terminals is at most $O(k^2)$. In Step 3, we can easily check whether or not $P_1, \ldots, P_k$ are a solution of the original problem in polynomial time. Therefore, the edge-disjoint version of the undirected k-DSPP can be solved in $|V|^{O(k^2)}$ time if the input graph is planar. ◀

References

1 Andreas Björklund and Thore Husfeldt. Shortest two disjoint paths in polynomial time. In *ICALP*, pages 211–222, 2014.

2 Glencora Borradaile, Amir Nayyeri, and Farzad Zafarani. Towards single face shortest vertex-disjoint paths in undirected planar graphs. In *ESA*, pages 227–238, 2015.

3 Marek Cygan, Dániel Marx, Marcin Pilipczuk, and Michał Pilipczuk. The planar directed k-vertex-disjoint paths problem is fixed-parameter tractable. In *FOCS*, pages 197–206, 2013.

4 Éric Colin de Verdière and Alexander Schrijver. Shortest vertex-disjoint two-face paths in planar graphs. In *STACS*, pages 181–192, 2008.

5 Tali Eilam-Tzoreff. The disjoint shortest paths problem. *Discrete Applied Mathematics*, 85(2):113–138, 1998.

6 Steven Fortune, John E. Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.

7 András Frank. *Paths, Flows, and VLSI-Layout*, chapter Packing paths, cuts and circuits – a survey, pages 49–100. Springer-Verlag, 1990.

8 Hiroshi Hirai and Hiroyuki Namba. Shortest $(A + B)$-path packing via hafnian. arXiv:1603.08073, 2016.

9 Richard M. Karp. On the computational complexity of combinatorial problems. *Networks*, 5:45–68, 1975.

10 Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce Reed. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B*, 102(2):424–435, 2012.

11 Yusuke Kobayashi and Christian Sommer. On shortest disjoint paths in planar graphs. *Discrete Optimization*, 7(4):234–245, 2010.

12 James F. Lynch. The equivalence of theorem proving and the interconnection problem. *SIGDA Newsletter*, 5(3):31–36, 1975.

13 Richard G. Ogier, Vladislav Rutenburg, and Nachum Shacham. Distributed algorithms for computing shortest pairs of disjoint paths. *IEEE Transactions on Information Theory*, 39(2):443–455, 1993.

14 Neil Robertson and Paul D. Seymour. *Paths, Flows, and VLSI-Layout*, chapter An outline of a disjoint paths algorithm, pages 267–292. Springer-Verlag, 1990.

15 Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995. doi:10.1006/jctb.1995.1006.

16 Alexander Schrijver. Finding k disjoint paths in a directed planar graph. *SIAM Journal on Computing*, 23(4):780–788, 1994.

17 Paul D. Seymour. Disjoint paths in graphs. *Discrete Mathematics*, 29:293–309, 1980.

18 Y. Shiloach and Y. Perl. Finding two disjoint paths between two pairs of vertices in a graph. *J. ACM*, 25(1):1–9, 1978.

19 Yossi Shiloach. A polynomial solution to the undirected two paths problem. *Journal of the ACM*, 27(3):445–456, 1980. doi:10.1145/322203.322207.

20 Anand Srinivas and Eytan Modiano. Finding minimum energy disjoint paths in wireless ad-hoc networks. *Wireless Networks*, 11(4):401–417, 2005. doi:10.1007/s11276-005-1765-0.

21 Torsten Tholey. Finding disjoint paths on directed acyclic graphs. In *WG*, pages 319–330, 2005.

22 Carsten Thomassen. 2-linked graphs. *European Journal of Combinatorics*, 1:371–378, 1980.

Triangle Packing in (Sparse) Tournaments: Approximation and Kernelization*

Stéphane Bessy[1], Marin Bougeret[2], and Jocelyn Thiebaut[3]

1 **Université de Montpellier – CNRS, LIRMM, Montpellier, France**
bessy@lirmm.fr
2 **Université de Montpellier – CNRS, LIRMM, Montpellier, France**
bougeret@lirmm.fr
3 **Université de Montpellier – CNRS, LIRMM, Montpellier, France**
thiebaut@lirmm.fr

Abstract

Given a tournament $\mathcal{T}$ and a positive integer k, the C_3-Packing-T problem asks if there exists a least k (vertex-)disjoint directed 3-cycles in $\mathcal{T}$. This is the dual problem in tournaments of the classical minimal feedback vertex set problem. Surprisingly C_3-Packing-T did not receive a lot of attention in the literature. We show that it does not admit a PTAS unless P=NP, even if we restrict the considered instances to sparse tournaments, that is tournaments with a feedback arc set (FAS) being a matching. Focusing on sparse tournaments we provide a $(1+\frac{6}{c-1})$ approximation algorithm for sparse tournaments having a linear representation where all the backward arcs have "length" at least c. Concerning kernelization, we show that C_3-Packing-T admits a kernel with $\mathcal{O}(m)$ vertices, where m is the size of a given feedback arc set. In particular, we derive a $\mathcal{O}(k)$ vertices kernel for C_3-Packing-T when restricted to sparse instances. On the negative size, we show that C_3-Packing-T does not admit a kernel of (total bit) size $\mathcal{O}(k^{2-\epsilon})$ unless NP $\subseteq$ coNP / Poly. The existence of a kernel in $\mathcal{O}(k)$ vertices for C_3-Packing-T remains an open question.

1998 ACM Subject Classification G.2.2 [Graph Theory] Graph Algorithms

Keywords and phrases Tournament, triangle packing, feedback arc set, approximation and parameterized algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.14

1 Introduction and related work

Tournament

A tournament $\mathcal{T}$ on n vertices is an orientation of the edges of the complete undirected graph K_n. Thus, given a tournament $\mathcal{T} = (V, A)$, where $V = \{v_i, i \in [n]\}$, for each $i, j \in [n]$, either $v_iv_j \in A$ or $v_jv_i \in A$. A tournament $\mathcal{T}$ can alternatively be defined by an ordering $\sigma(\mathcal{T}) = (v_1, \ldots, v_n)$ of its vertices and a set of *backward arcs* $\overleftarrow{A}_\sigma(\mathcal{T})$ (which will be denoted $\overleftarrow{A}(\mathcal{T})$ as the considered ordering is not ambiguous), where each arc $a \in \overleftarrow{A}(\mathcal{T})$ is of the form $v_{i_1}v_{i_2}$ with $i_2 < i_1$. Indeed, given $\sigma(\mathcal{T})$ and $\overleftarrow{A}(\mathcal{T})$, we can define $V = \{v_i, i \in [n]\}$ and $A = \overleftarrow{A}(\mathcal{T}) \cup \overrightarrow{A}(\mathcal{T})$ where $\overrightarrow{A}(\mathcal{T}) = \{v_{i_1}v_{i_2} : (i_1 < i_2)$ and $v_{i_2}v_{i_1} \notin \overleftarrow{A}(\mathcal{T})\}$ is the set of forward arcs of $\mathcal{T}$ in the given ordering $\sigma(\mathcal{T})$. In the following, $(\sigma(\mathcal{T}), \overleftarrow{A}(\mathcal{T}))$ is called a *linear*

* An extended version of this paper is available at [4], https://hal-lirmm.ccsd.cnrs.fr/lirmm-01550313.

representation of the tournament $\mathcal{T}$. For a backward arc $e = v_j v_i$ of $\sigma(\mathcal{T})$ the *span value* of e is $j - i - 1$. Then $\texttt{minspan}(\sigma(\mathcal{T}))$ (resp. $\texttt{maxspan}(\sigma(\mathcal{T}))$) is simply the minimum (resp. maximum) of the span values of the backward arcs of $\sigma(\mathcal{T})$.
A set $A' \subseteq A$ of arcs of $\mathcal{T}$ is a *feedback arc set* (or *FAS*) of $\mathcal{T}$ if every directed cycle of $\mathcal{T}$ contains at least one arc of A'. It is clear that for any linear representation $(\sigma(\mathcal{T}), \overleftarrow{A}(\mathcal{T}))$ of $\mathcal{T}$ the set $\overleftarrow{A}(\mathcal{T})$ is a FAS of $\mathcal{T}$. A tournament is *sparse* if it admits a FAS which is a matching.
We denote by C_3-Packing-T the problem of packing the maximum number of vertex disjoint triangles in a given tournament, where a triangle is a directed 3-cycle. More formally, an input of C_3-Packing-T is a tournament $\mathcal{T}$, an output is a set (called a *triangle packing*) $S = \{t_i, i \in [|S|]\}$ where each t_i is a triangle and for any $i \neq j$ we have $V(t_i) \cap V(t_j) = \emptyset$, and the objective is to maximize $|S|$. We denote by $opt(\mathcal{T})$ the optimal value of $\mathcal{T}$. We denote by C_3-Perfect-Packing-T the decision problem associated to C_3-Packing-T where an input $\mathcal{T}$ is positive iff there is a triangle packing S such that $V(S) = V(\mathcal{T})$ (which is called a *perfect triangle packing*).

Related work

We refer the reader to the extended version of the paper [4] where we recall the definitions of the problems mentioned bellow as well as the standard definitions about parameterized complexity and approximation. A first natural related problem is 3-Set-Packing as we can reduce C_3-Packing-T to 3-Set-Packing by creating an hyperedge for each triangle.

Classical complexity / approximation. It is known that C_3-Packing-T is polynomial if the tournament does not contain the three forbidden sub-tournaments described in [5]. From the point of view of approximability, the best approximation algorithm is the $\frac{4}{3} + \epsilon$ approximation of [7] for 3-Set-Packing, implying the same result for K_3-packing and C_3-Packing-T. Concerning negative results, it is known [9] that even K_3-packing is MAX SNP-hard on graphs with maximum degree four. The related "dual" problems FAST and FVST received a lot of attention with for example the NP-hardness and PTAS for FAS in [6] and [12] respectively, and the $\frac{7}{3}$ approximation and inapproximability results for FVST in [13].

Kernelization. We precise that the implicitly considered parameter here is the size of the solution. There is a $\mathcal{O}(k^2)$ vertex kernel for K_3-packing in [14], and even a $\mathcal{O}(k^2)$ vertex kernel for 3-Set-Packing in [1], which is obtained by only removing vertices of the ground set. This remark is important as it directly implies a $\mathcal{O}(k^2)$ vertex kernel for C_3-Packing-T (see Section 4). Let us now turn to negative results. There is a whole line of research dedicated to finding lower bounds on the size of polynomial kernels. The main tool involved in these bounds is the weak composition introduced in [10] (whose definition is recalled in [4]). Weak composition allowed several almost tight lower bounds, with for examples the $\mathcal{O}(k^{d-\epsilon})$ for d-Set-Packing and $\mathcal{O}(k^{d-4-\epsilon})$ for K_d-packing of [10]. These results where improved in [8] to $\mathcal{O}(k^{d-\epsilon})$ for perfect d-Set-Packing, $\mathcal{O}(k^{d-1-\epsilon})$ for K_d-packing, and leading to $\mathcal{O}(k^{2-\epsilon})$ for perfect K_3-packing. Notice that negative results for the "perfect" version of problems (where parameter $k = \frac{n}{d}$, where d is the number of vertices of the packed structure) are stronger than for the classical version where k is arbitrary. Kernel lower bound for these "perfect" versions is sometimes referred as *sparsification lower bounds*.

Our contributions

Our objective is to study the approximability and kernelization of C_3-Packing-T. On the approximation side, a natural question is a possible improvement of the $\frac{4}{3}+\epsilon$ approximation implied by 3-Set-Packing. We show that, unlike FAST, C_3-Packing-T does not admit a PTAS unless P=NP, even if the tournament is sparse. We point out that, surprisingly, we were not able to find any reference establishing a negative result for C_3-Packing-T, even for the NP-hardness. As these results show that there is not much room for improving the approximation ratio, we focus on sparse tournaments and followed a different approach by looking for a condition that would allow ratio arbitrarily close to 1. In that spirit, we provide a $(1+\frac{6}{c-1})$ approximation algorithm for sparse tournaments having a linear representation with `minspan` at least c. Concerning kernelization, we complete the panorama of sparsification lower bounds of [11] by proving that C_3-Perfect-Packing-T does not admit a kernel of (total bit) size $\mathcal{O}(n^{2-\epsilon})$ unless NP $\subseteq$ coNP / Poly. This implies that C_3-Packing-T does not admit a kernel of (total bit) size $\mathcal{O}(k^{2-\epsilon})$ unless NP $\subseteq$ coNP / Poly. We also prove that C_3-Packing-T admits a kernel of $\mathcal{O}(m)$ vertices, where m is the size of a given FAS of the instance, and that C_3-Packing-T restricted to sparse instances has a kernel in $\mathcal{O}(k)$ vertices (and so of total size bit $\mathcal{O}(k\log(k))$). The existence of a kernel in $\mathcal{O}(k)$ vertices for the general C_3-Packing-T remains our main open question.

2 Specific notations and observations

Given a linear representation $(\sigma(\mathcal{T}),\overleftarrow{A}(\mathcal{T}))$ of a tournament $\mathcal{T}$, a triangle t in $\mathcal{T}$ is a triple $t=(v_{i_1},v_{i_2},v_{i_3})$ with $i_l<i_{l+1}$ such that either $v_{i_3}v_{i_1}\in\overleftarrow{A}(\mathcal{T})$, $v_{i_3}v_{i_2}\notin\overleftarrow{A}(\mathcal{T})$ and $v_{i_2}v_{i_1}\notin\overleftarrow{A}(\mathcal{T})$ (in this case we call t a *triangle with backward arc* $v_{i_3}v_{i_1}$), or $v_{i_3}v_{i_1}\notin\overleftarrow{A}(\mathcal{T})$, $v_{i_3}v_{i_2}\in\overleftarrow{A}(\mathcal{T})$ and $v_{i_2}v_{i_1}\in\overleftarrow{A}(\mathcal{T})$ (in this case we call t a *triangle with two backward arcs* $v_{i_3}v_{i_2}$ and $v_{i_2}v_{i_1}$). Given two tournaments $\mathcal{T}_1,\mathcal{T}_2$ defined by $\sigma(\mathcal{T}_l)$ and $\overleftarrow{A}(\mathcal{T}_l)$ we denote by $\mathcal{T}=\mathcal{T}_1\mathcal{T}_2$ the tournament called the concatenation of $\mathcal{T}_1$ and $\mathcal{T}_2$, where $\sigma(\mathcal{T})=\sigma(\mathcal{T}_1)\sigma(\mathcal{T}_2)$ is the concatenation of the two sequences, and $\overleftarrow{A}(\mathcal{T})=\overleftarrow{A}(\mathcal{T}_1)\cup\overleftarrow{A}(\mathcal{T}_2)$. Given a tournament $\mathcal{T}$ and a subset of vertices X, we denote by $\mathcal{T}\setminus X$ the tournament $\mathcal{T}[V(\mathcal{T})\setminus X]$ induced by vertices $V(\mathcal{T})\setminus X$, and we call this operation *removing* X *from* $\mathcal{T}$. Given an arc $a=uv$ we define $h(a)=v$ as the head of a and $t(a)=u$ as the tail of a. Given a linear representation $(V(\mathcal{T}),\overleftarrow{A}(\mathcal{T}))$ and an arc $a\in\overleftarrow{A}(\mathcal{T})$, we define $s(a)=\{v : h(a)<v<t(a)\}$ as the *span* of a. Notice that the span value of a is then exactly $|s(a)|$. Given a linear representation $(V(\mathcal{T}),\overleftarrow{A}(\mathcal{T}))$ and a vertex $v\in V(\mathcal{T})$, we define the degree of v by $d(v)=(a,b)$, where $a=|\{vu\in\overleftarrow{A}(\mathcal{T}) : u<v\}|$ is called the *left degree* of v and $b=|\{uv\in\overleftarrow{A}(\mathcal{T}) : u>v\}|$ is called the *right degree* of v. We also define $V_{(a,b)}=\{v\in V(\mathcal{T})|d(v)=(a,b)\}$. Given a set of pairwise distinct pairs D, we denote by C_3-Packing-T^D the problem C_3-Packing-T restricted to tournaments such that there exists a linear representation where $d(v)\in D$ for all v. Notice that when $D_M=\{(0,1),(1,0),(0,0)\}$, instances of C_3-Packing-T^{D_M} are the sparse tournaments. Finally let us point out that it is easy to decide in polynomial time if a tournament is sparse or not, and if so, to give a linear representation whose FAS is a matching. The corresponding algorithm is detailed in [4]. Thus, in the following, when considering a sparse tournament we will assume that a linear ordering of it where backward arcs form a matching is also given. Finally, due to space limitations, the proofs of the results marked with '($\star$)' have been removed and are available in [4].

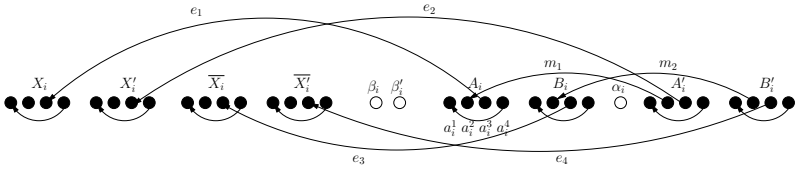


Figure 1 Example of a variable gadget L_i.

3 Approximation for sparse tournaments

3.1 APX-hardness for sparse tournaments

In this subsection we prove that C_3-Packing-T^{D_M} is APX-hard by providing a L-reduction (see Definition in [4]) from Max 2-SAT(3), which is known to be APX-hard [2, 3]. Recall that in the Max 2-SAT(3) problem each clause contains exactly 2 variables and each variable appears in at most 3 clauses (and at most twice positively and once negatively).

Definition of the reduction. Let $\mathcal{F}$ be an instance of Max 2-SAT(3). In the following, we will denote by n the number of variables in $\mathcal{F}$ and m the number of clauses. Let $\{x_i, 1 \in [n]\}$ be the set of variables of $\mathcal{F}$ and $\{C_j, j \in [m]\}$ its set of clauses.

We now define a reduction f which maps an instance $\mathcal{F}$ of Max 2-SAT(3) to an instance $\mathcal{T}$ of C_3-Packing-T^{D_M}. For each variable x_i with $i \in [n]$, we create a tournament L_i as follows and we call it *variable gadget*. We refer the reader to Figure 1 where an example of variable gadget is depicted. Let $\sigma(L_i) = (X_i, X'_i, \overline{X_i}, \overline{X_i}', \{\beta_i\}, \{\beta'_i\}\ , A_i, B_i, \{\alpha_i\}, A'_i, B'_i)$. We define $C = \{X_i, X'_i, \overline{X_i}, \overline{X_i}'\ , A_i, B_i, A'_i, B'_i\}$. All sets of C have size 4. We denote $X_i = (x_i^1, x_i^2, x_i^3, x_i^4)$, and we extend the notation in a straightforward manner to the other others sets of C. Let us now define $\overleftarrow{A}(L_i)$. For each set of C, we add a backward arc whose head is the first element and the tail is the last element (for example for X_i we add the arc $x_i^4 x_i^1$). Then, we add to $\overleftarrow{A}(L_i)$ the set $\{e_1, e_2, e_3, e_4\}$ where $e_1 = x_i^3 a_i^3$, $e_2 = x_i'^3 a_i'^3$, $e_3 = \overline{x_i^3} b_i^3$, $e_4 = \overline{x_i'^3} b_i'^3$ and the set $\{m_1, m_2\}$ where $m_1 = a_i'^2 a_i^2$, $m_2 = b_i'^2 b_i^2$ called the two *medium arcs* of the variable gadget. This completes the description of tournament L_i. Let $L = L_1 \dots L_n$ be the concatenation of the L_i.

For each clause C_j with $j \in [1, m]$, we create a tournament K_j with ordering $\sigma(K_i) = (\theta_j, d_j^1, c_j^1, c_j^2, d_j^2)$ and $\overleftarrow{A}(K_i) = \{d_j^2 d_j^1\}$. We also define $K = K_1 \dots K_m$. Let us now define $\mathcal{T} = LK$. We add to $\overleftarrow{A}(\mathcal{T})$ the following backward arcs from $V(K)$ to $V(L)$. If $C_j = l_{i_1} \vee l_{i_2}$ is a clause in $\mathcal{F}$ then we add the arcs $c_j^1 v_{i_1}, c_j^2 v_{i_2}$ where v_{i_c} is the vertex in $\{x_{i_c}^2, x_{i_c}'^2, \overline{x_{i_c}^2}\}$ corresponding to l_{i_c}: if l_{i_c} is a positive occurrence of variable i_c we chose $v_{i_c} \in \{x_{i_c}^2, x_{i_c}'^2\}$, otherwise we chose $v_{i_c} = \overline{x_{i_c}^2}$. Moreover, we chose vertices v_{i_c} in such a way that for any $i \in [n]$, for each $v \in \{x_i^2, x_i'^2, \overline{x_i^2}\}$ there exists a unique arc $a \in \overleftarrow{A}(\mathcal{T})$ such that $h(a) = v$. This is always possible as each variable has at most two positive occurrences and one negative occurrence. Thus, x_i^2 represent the first positive occurrence of variable i, and $x_i'^2$ the second one. We refer the reader to Figure 2 where an example of the connection between variable and clause gadget is depicted.

Notice that vertices of $\overline{X'_i}$ are never linked to the clauses gadget. However, we need this set to keep the variable gadget symmetric so that setting x_i to true or false leads to the same number of triangles inside L_i. This completes the description of $\mathcal{T}$. Notice that the degree of any vertex is in $\{(0,1),(1,0),(0,0)\}$, and thus $\mathcal{T}$ is an instance of C_3-Packing-T^{D_M}.

Let us now distinguish three different types of triangles in $\mathcal{T}$. A triangle $t = (v_1, v_2, v_3)$ of $\mathcal{T}$ is called an *outer* triangle iff $\exists j \in [m]$ such that $v_2 = \theta_j$ and $v_3 = c_j^l$ (implying that $v_1 \in V(L)$),

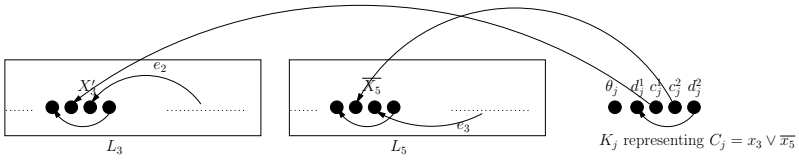


Figure 2 Example showing how a clause gadget is attached to variable gadgets.

variable inner iff $\exists i \in [n]$ such that $V(t) \subseteq V(L_i)$, and *clause inner* iff $\exists j \in [m]$ such that $V(t) \subseteq V(K_j)$. Notice that a triangle $t = (v_1, v_2, v_3)$ of $\mathcal{T}$ which is neither outer, variable or clause inner has necessarily $v_3 = c_j^l$ for some j, and $v_2 \neq \theta_j$ (v_2 could be in $V(L)$ or $V(K)$). In the following definition, for any $Y \in C$ (where $C = \{X_i, X'_i, \overline{X_i}, \overline{X_i}', A_i, B_i, A'_i, B'_i\}$) with $Y = (y^1, y^2, y^3, y^4)$, we define $t_Y^2 = (y^1, y^2, y^4)$ and $t_Y^3 = (y^1, y^3, y^4)$. For example, $t^2_{X'_i} = (x_i'^1, x_i'^2, x_i'^4)$. For any $i \in [n]$, we define P_i and $\overline{P_i}$, two sets of vertex disjoint variable inner triangles of $V(L_i)$, by:

- $P_i = \{t^3_{X_i}, t^3_{X'_i}, t^2_{\overline{X_i}}, t^2_{\overline{X'_i}}, t^3_{A_i}, t^2_{B_i}, t^3_{A'_i}, t^2_{B'_i}, (h(e_3), \beta_i, t(e_3)), (h(e_4), \beta'_i, t(e_4)), (h(m_1), \alpha_i, t(m_1))\}$
- $\overline{P_i} = \{t^2_{X_i}, t^2_{X'_i}, t^3_{\overline{X_i}}, t^3_{\overline{X'_i}}, t^2_{A_i}, t^3_{B_i}, t^2_{A'_i}, t^3_{B'_i}, (h(e_1), \beta_i, t(e_1)), (h(e_2), \beta'_i, t(e_2)), (h(m_2), \alpha_i, t(m_2))\}$

Notice that P_i (resp. $\overline{P_i}$) uses all vertices of L_i except $\{x_i^2, x_i'^2\}$ (resp. $\{\overline{x_i^2}, \overline{x_i'^2}\}$). For any $j \in [m]$, and $x \in [2]$ we define the set of clause inner triangle of K_j, that is $Q_j^x = \{(d_j^1, c_j^x, d_j^2)\}$.

Informally, setting variable x_i to true corresponds to create the 11 triangles of P_i in L_i (as leaving vertices $\{x_i^2, x_i^{2'}\}$ available allows to create outer triangles corresponding to satisfied clauses), and setting it to false corresponds to create the 11 triangles of $\overline{P_i}$. Satisfying a clause j using its x^{th} literal (represented by a vertex $v \in V(L)$) corresponds to create triangle in Q_j^{3-x} as it leaves c_j^x available to create the triangle (v, θ_j, c_j^x). Our final objective (in Lemma 4) is to prove that satisfying k clauses is equivalent to find $11n+m+k$ vertex disjoint triangles.

Restructuration lemmas. Given a solution S, let $I_i^L = \{t \in S : V(t) \subseteq V(L_i)\}$, $I_j^K = \{t \in S : V(t) \subseteq V(K_j)\}$, $I^L = \cup_{i\in[n]} I_i^L$ be the set of variable inner triangles of S, $I^K = \cup_{j\in[m]} I_j^K$ be the set of clause inner triangles of S, and $O = \{t \in S \ t \text{ is an outer triangle }\}$ be the set of outer triangles of S. Notice that *a priori* I^L, I^K, O does not necessarily form a partition of S. However, we will show in the next lemmas how to restructure S such that I^L, I^K, O becomes a partition.

▶ **Lemma 1** (⋆). *For any S we can compute in polynomial time a solution $S' = \{t'_l, l \in [k]\}$ such that $|S'| \geq |S|$ and for all $j \in [m]$ there exists $x \in [2]$ such that $I_j'^K = Q_j^x$ and*

- *either S' does not use any other vertex of K_j ($V(S') \cap V(K_j) = V(Q_j^x)$)*
- *either S' contains an outer triangle $t'_l = (v, \theta_j, c_j^{3-x})$ with $v \in V(L)$ (implying $V(S') \cap V(K_j) = V(K_j)$)*

▶ **Corollary 2.** *For any S we can compute in polynomial time a solution S' such that $|S'| \geq |S|$, and S' only contains outer, variable inner, and clause inner triangles. Indeed, in the solution S' of Lemma 1, given any $t \in S'$, either $V(t)$ intersects $V(K_j)$ for some j and then t is an outer or a clause inner triangle, or $V(t) \subseteq V(L_i)$ for $i \in [n]$ as there is no backward arc uv with $u \in V(L_{i_1})$ and $v \in V(L_{i_2})$ with $i_1 \neq i_2$.*

▶ **Lemma 3** (⋆). *For any S we can compute in polynomial time a solution S' such that $|S'| \geq |S|$, S' satisfies Lemma 1, and for every $i \in [n]$, $I_i'^L = P_i$ or $I_i'^L = \overline{P_i}$.*

Proof of the L-reduction. We are now ready to prove the main lemma (recall that f is the reduction from Max 2-SAT(3) to C_3-Packing-T^{D_M} described in Section 3.1), and also the main theorem of the section.

▶ **Lemma 4.** *Let $\mathcal{F}$ be an instance of* Max 2-SAT(3)*. For any k, there exists an assignment a of $\mathcal{F}$ satisfying at least k clauses if and only if there exists a solution S of $f(\mathcal{F})$ with $|S| \geq 11n + m + k$, where n and m are respectively is the number of variables and clauses in $\mathcal{F}$. Moreover, in the $\Leftarrow$ direction, assignment a can be computed from S in polynomial time.*

Proof. For any $i \in [n]$, let $A_i = P_i$ if x_i is set to true in a, and $A_i = \overline{P_i}$ otherwise. We first add to S the set $\cup_{i\in[n]} A_i$. Then, let $\{C_{j_l}, l \in [k]\}$ be k clauses satisfied by a. For any $l \in [k]$, let i_l be the index of a literal satisfying C_{j_l}, let $x \in [2]$ such that $c^x_{j_l}$ corresponds to this literal, and let $Z_l = \{x^2_{i_l}, x^{'2}_{i_l}\}$ if literal i_l is positive, and $Z_l = \{\overline{x^2_{i_l}}\}$ otherwise. For any $j \in [m]$, if $j = i_l$ for some l (meaning that j corresponds to a satisfied clause), we add to S the triangle in Q^{3-x}_j, and otherwise we arbitrarily add the triangle Q^1_j. Finally, for any $l \in [k]$ we add to S triangle $t_l = (y_l, \theta_{j_l}, c^x_{j_l})$ where $y_l \in Z_l$ is such that y_l is not already used in another triangle. Notice that such an y_l always exists as triangles of $A_i, i \in [n]$ do not intersect Z_l (by definition of the A_i), and as there are at most two clauses that are true due to positive literal, and one clause that is true due to a negative literal. Thus, S has $11n + m + k$ vertex disjoint triangles.

Conversely, let S a solution of $f(\mathcal{F})$ with $|S| \geq 11n + m + k$. By Lemma 3 we can construct in polynomial time a solution S' from S such that $|S'| \geq |S|$, S' only contains outer, variable or clause inner triangles, for each $j \in [m]$ there exists $x \in [2]$ such that $I^{'K}_j = Q^x_j$, and for each $i \in [n], I^{'L}_i = P_i$ or $I^{'L}_i = \overline{P_i}$. This implies that the $k' \geq k$ remaining triangles must be outer triangles. Let $\{t'_l, l \in [k']\}$ be these k' outer triangles with $t'_l = (y_l, \theta_{j_l}, c^{x_l}_{j_l})$ Let us define the following assignation a: for each $i \in [n]$, we set x_i to true if $I^{'L}_i = P_i$, and false otherwise. This implies that a satisfies at least clauses $\{C_{j_l}, l \in [k']\}$. ◀

▶ **Theorem 5.** *C_3-Packing-T^{D_M} is APX-hard, and thus does not admit a PTAS unless $P = NP$.*

Proof. Let us check that Lemma 4 implies a L-reduction (whose definition is recalled in [4]). Let OPT_1 (resp. OPT_2) be the optimal value of $\mathcal{F}$ (resp. $f(\mathcal{F})$). Notice that Lemma 4 implies that $OPT_2 = OPT_1 + 11n + m$. It is known that $OPT_1 \geq \frac{3}{4}m$ (where m is the number of clauses of $\mathcal{F}$). As $n \leq m$ (each variable has at least one positive and one negative occurrence), we get $OPT_2 = OPT_1 + 11n + m \leq \alpha OPT_1$ for an appropriate constant α, and thus point (a) of the definition is verified. Then, given a solution S' of $f(\mathcal{F})$, according to Lemma 4 we can construct in polynomial time an assignment a satisfying $c(a)$ clauses with $c(a) \geq S' - 11n - m$. Thus, the inequality (b) of the Definition of a L-reduction with $\beta = 1$ becomes $OPT_1 - c(a) \leq OPT_2 - S' = OPT_1 + 11n + m - S'$, which is true. ◀

Reduction of Theorem 5 does not imply the NP-hardness of C_3-Perfect-Packing-T as there remain some unused vertices. However, it is straightforward to adapt the reduction by adding backward arcs whose head (resp. tail) are before (resp. after) $\mathcal{T}$ to consume the remaining vertices. This leads to the following result.

▶ **Theorem 6** ($\star$)**.** *C_3-Perfect-Packing-T^{D_M} is NP-hard.*

To establish the kernel lower bound of Section 4, we also need the NP-hardness of C_3-Perfect-Packing-T where instances have a slightly simpler structure (to the price of losing the property that there exists a FAS which is a matching).

▶ **Theorem 7** (⋆). *C_3-Perfect-Packing-T remains NP-hard even restricted to tournaments $\mathcal{T}$ admitting the following linear ordering.*

- *$\mathcal{T} = LK$ where L and K are two tournaments*
- *tournaments L and K are "fixed":*
  - *$K = K_1 \dots K_m$ for some m, where for each $j \in [m]$ we have $V(K_j) = (\theta_j, c_j)$*
  - *$L = R_1 L_1 \dots L_n R_2$, where each L_i has is a copy of the variable gadget of Section 3.1, $R_i = \{r_i^l, l \in [n']\}$ where $n' = 2n - m$, and in addition $\overleftarrow{L}$ also contains $R = \{(r_2^l r_1^l), l \in [n']\}$ which are called the dummy arcs.*

3.2 $(1 + \frac{6}{c-1})$-approximation when backward arcs have large minspan

Given a set of pairwise distinct pairs D and an integer c, we denote by C_3-Packing-T$^D_{\geq c}$ the problem C_3-Packing-T^D restricted to tournaments such that there exists a linear representation of minspan at least c and where $d(v) \in D$ for all v. In all this section we consider an instance $\mathcal{T}$ of C_3-Packing-T$^{D_M}_{\geq c}$ with a given linear ordering $(V(\mathcal{T}), \overleftarrow{A}(\mathcal{T}))$ of minspan at least c and whose degrees belong to D_M. The motivation for studying the approximability of this special case comes from the situation of MAX-SAT(c) where the approximability becomes easier as c grows, as the derandomized uniform assignment provides a $\frac{2^c}{2^c-1}$ approximation algorithm. Somehow, one could claim that MAX-SAT(c) becomes easy to approximate for large c as there are many ways to satisfy a given clause. As the same intuition applies for tournaments admitting an ordering with large minspan (as there are $c-1$ different ways to use a given backward in a triangle), our objective was to find a polynomial approximation algorithm whose ratio tends to 1 when c increases.

Let us now define algorithm Φ. We define a bipartite graph $G = (V_1, V_2, E)$ with $V_1 = \{v_a^1 : a \in \overleftarrow{A}(\mathcal{T})\}$ and $V_2 = \{v_l^2 : v_l \in V_{(0,0)}\}$. Thus to each backward arc we associate a vertex in V_1 and to each vertex v_l with $d(v_l) = (0,0)$ we associate a vertex in V_2. Then $\{v_a^1, v_l^2\} \in E$ iff $(h(a), v_l, t(a))$ is a triangle in $\mathcal{T}$.

In phase 1, Φ computes a maximum matching $M = \{e_l, l \in [|M|]\}$ in G. For every $e_l = \{v_{ij}^1, v_l^2\} \in M$ create a triangle $t_l^1 = (v_j, v_l, v_i)$. Let $S^1 = \{t_l^1, l \in [|M|]\}$. Notice that triangles of S^1 are vertex disjoint. Let us now turn to phase 2. Let $\mathcal{T}^2$ be the tournament $\mathcal{T}$ where we removed all vertices $V(S^1)$. Let $(V(\mathcal{T}^2), \overleftarrow{A}(\mathcal{T}^2))$ be the linear ordering of $\mathcal{T}^2$ obtained by removing $V(S^1)$ in $(V(\mathcal{T}), \overleftarrow{A}(\mathcal{T}))$. We say that three distinct backward edges $\{a_1, a_2, a_3\} \subseteq \overleftarrow{A}(\mathcal{T}^2)$ can be packed into triangles t_1 and t_2 iff $V(\{t_1, t_2\}) = V(\{a_1, a_2, a_3\})$ and the t_i are vertex disjoint. For example, if $h(a_1) < h(a_2) < t(a_1) < h(a_3) < t(a_2) < t(a_3)$, then $\{a_1, a_2, a_3\}$ can be packed into $(h(a_1), h(a_2), t(a_1))$ and $(h(a_3), t(a_2), t(a_3))$ (recall that when $\overleftarrow{A}(\mathcal{T})$ form a matching, (u, v, w) is triangle iff $wu \in \overleftarrow{A}(\mathcal{T})$ and $u < v < w$), and if $h(a_1) < h(a_2) < t(a_2) < h(a_3) < t(a_3) < t(a_1)$, then $\{a_1, a_2, a_3\}$ cannot be packed into two triangles. In phase 2, while it is possible, Φ finds a triplet of arcs of $Y \subseteq \overleftarrow{A}(\mathcal{T}^2)$ that can be packed into triangles, create the two corresponding triangles, and remove $V(Y)$. Let S^2 be the triangle created in phase 2 and let $S = S^1 \cup S^2$.

▶ **Observation 8.** *For any $a \in \overleftarrow{A}(\mathcal{T})$, either $V(a) \subseteq V(S)$ or $V(a) \cap V(S) = \emptyset$. Equivalently, no backward arc has one endpoint in $V(S)$ and the other outside $V(S)$.*

According to Observation 8, we can partition $\overleftarrow{A}(\mathcal{T}) = \overleftarrow{A}_0 \cup \overleftarrow{A}_1 \cup \overleftarrow{A}_2$, where for $i \in \{1, 2\}$, $\overleftarrow{A}^i = \{a \in \overleftarrow{A}(\mathcal{T}) : V(a) \subseteq V(S^i)$ is the set of arcs used in phase i, and $\overleftarrow{A}_0 =_{def} \{b_i, i \in [x]\}$ are the remaining unused arcs. Let $\overleftarrow{A}_\Phi = \overleftarrow{A}_1 \cup \overleftarrow{A}_2$, $m_i = |\overleftarrow{A}_i|$, $m = m_0 + m_1 + m_2$ and $m_\Phi = m_1 + m_2$ the number of arcs (entirely) consumed by Φ. To prove the $1 + \frac{6}{c-1}$ desired approximation ratio, we will first prove in Lemma 9 that Φ uses at most all the arcs

($m_A \geq (1-\epsilon(c))m$), and in Theorem 10 that the number of triangles made with these arcs is "optimal". Notice that the latter condition is mandatory as if Φ used its m_Φ arcs to only create $\frac{2}{3}(m_\Phi)$ triangles in phase 2 instead of creating $m' \approx m_\Phi$ triangle with m' backward arcs and m' vertices of degree $(0,0)$, we would have a $\frac{3}{2}$ approximation ratio.

▶ **Lemma 9** ($\star$). *For any $c \geq 2$, $m_\Phi \geq (1-\frac{6}{c+5})m$*

▶ **Theorem 10.** *For any $c \geq 2$, Φ is a polynomial $(1+\frac{6}{c-1})$ approximation algorithm for C_3-Packing-T$^{D_M}_{\geq c}$.*

Proof. Let OPT be an optimal solution. Let us define $OPT_i \subseteq OPT$ and $\overleftarrow{A}^*_i \subseteq \overleftarrow{A}(\mathcal{T})$ as follows. Let $t=(u,v,w) \in OPT$. As the FAS of the instance is a matching, we know that $wu \in \overleftarrow{A}(\mathcal{T})$ as we cannot have a triangle with two backward arcs. If $d(v)=(0,0)$ then we add t to OPT_1 and wu to $\overleftarrow{A}^*_1$. Otherwise, let v' be the other endpoint of the unique arc a containing v. If $v' \notin V(OPT)$, then we add t to OPT_3 and $\{wu, a\}$ to $\overleftarrow{A}^*_3$. Otherwise, let $t' \in OPT$ such that $v' \in V(t')$. As the FAS of the instance is a matching we know that v' is the middle point of t', or more formally that $t'=(u',v',w')$ with $u'w' \in \overleftarrow{A}(\mathcal{T})$. We add $\{t,t'\}$ to OPT_2 and $\{wu, a, w'u'\}$ to $\overleftarrow{A}^*_2$. Notice that the OPT_i form a partition of OPT, and that the $\overleftarrow{A}^*_i$ have pairwise empty intersection, implying $|\overleftarrow{A}^*_1| + |\overleftarrow{A}^*_2| + |\overleftarrow{A}^*_3| \leq m$. Notice also that as triangles of OPT_1 correspond to a matching of size $|OPT_1|$ in the bipartite graph defined in phase 1 of algorithm Φ, we have $|OPT_1| = |\overleftarrow{A}^*_1| \leq |\overleftarrow{A}_1|$.

Putting pieces together we get (recall that S is the solution computed by Φ) $|OPT| = |OPT_1| + |OPT_2| + |OPT_3| = |\overleftarrow{A}^*_1| + \frac{2}{3}|\overleftarrow{A}^*_2| + \frac{1}{2}|\overleftarrow{A}^*_3| \leq |\overleftarrow{A}^*_1| + \frac{2}{3}(|\overleftarrow{A}^*_2| + |\overleftarrow{A}^*_3|) \leq |\overleftarrow{A}^*_1| + \frac{2}{3}(m - |\overleftarrow{A}^*_1|) \leq \frac{1}{3}|\overleftarrow{A}_1| + \frac{2}{3}m$ and $|S| = |S^1| + |S^2| = |\overleftarrow{A}_1| + \frac{2}{3}|\overleftarrow{A}_2| \geq |\overleftarrow{A}_1| + \frac{2}{3}((1-\frac{6}{c+5})m - |\overleftarrow{A}_1|) = \frac{1}{3}|\overleftarrow{A}_1| + \frac{2}{3}(1-\frac{6}{c+5})m$ which implies the desired ratio. ◀

4 Kernelization

In all this section we consider the decision problem C_3-Packing-T parameterized by the size of the solution. Thus, an input is a pair $I=(\mathcal{T},k)$ and we say that I is positive iff there exists a set of k vertex disjoint triangles in $\mathcal{T}$.

4.1 Positive results for sparse instances

Observe first that the kernel in $\mathcal{O}(k^2)$ vertices for 3-Set Packing of [1] directly implies a kernel in $\mathcal{O}(k^2)$ vertices for C_3-Packing-T. Indeed, given an instance $(\mathcal{T}=(V,A),k)$ of C_3-Packing-T, we create an instance $(I'=(V,C),k)$ of 3-Set Packing by creating an hyperedge $c \in C$ for each triangle of $\mathcal{T}$. Then, as the kernel of [1] only removes vertices, it outputs an induced instance $(\overline{I'}=I'[V'],k')$ of I with $V' \subseteq V$, and thus this induced instance can be interpreted as a subtournament, and the corresponding instance $(\mathcal{T}[V'],k')$ is an equivalent tournament with $\mathcal{O}(k^2)$ vertices.

As shown in the next theorem, as we could expect it is also possible to have kernel bounded by the number of backward arcs.

▶ **Theorem 11.** *C_3-Packing-T admits a polynomial kernel with $\mathcal{O}(m)$ vertices, where m is the number of arcs in a given FAS of the input.*

Proof. Let $I=(\mathcal{T},k)$ be an input of the decision problem associated to C_3-Packing-T. Observe first that we can build in polynomial time a linear ordering $\sigma(\mathcal{T})$ whose backward arcs $\overleftarrow{A}(\mathcal{T})$ correspond to the given FAS. We will obtain the kernel by removing useless vertices

of degree $(0,0)$. Let us define a bipartite graph $G = (V_1, V_2, E)$ with $V_1 = \{v_a^1 : a \in \overleftarrow{A}(\mathcal{T})\}$ and $V_2 = \{v_l^2 : v_l \in V_{(0,0)}\}$. Thus, to each backward arc we associate a vertex in V_1 and to each vertex v_l with $d(v_l) = (0,0)$ we associate a vertex in V_2. Then, $\{v_a^1, v_l^2\} \in E$ iff $(h(a), v_l, t(a))$ is a triangle in $\mathcal{T}$. By Hall's theorem, we can in polynomial time partition V_1 and V_2 into $V_1 = A_1 \cup A_2$, $V_2 = B_0 \cup B_1 \cup B_2$ such that $N(A_2) \subseteq B_2$, $|B_2| \leq |A_2|$, and there is a perfect matching between vertices of A_1 and B_1 (B_0 is simply defined by $B_0 = V_2 \setminus (B_1 \cup B_2)$).

For any $i, 0 \leq i \leq 2$, let $X_i = \{v_l \in V_{(0,0)} : v_l^2 \in B_i\}$ be the vertices of $\mathcal{T}$ corresponding to B_i. Let $V_{\neq(0,0)} = V(\mathcal{T}) \setminus V_{(0,0)}$. Notice that $|V_{\neq(0,0)}| \leq 2m$. We define $\mathcal{T}' = \mathcal{T}[V_{\neq(0,0)} \cup X_1 \cup X_2]$ the sub-tournament obtained from $\mathcal{T}$ by removing vertices of X_0, and $I' = (\mathcal{T}', k)$. We point out that this definition of $\mathcal{T}'$ is similar to the final step of the kernel of [1] as our partition of V_1 and V_2 (more precisely $(A_1, B_0 \cup B_1)$) corresponds in fact to the crown decomposition of [1]. Observe that $|V(\mathcal{T}')| \leq 2m + |A_1| + |A_2| \leq 3m$, implying the desired bound of the number of vertices of the kernel.

It remains to prove that I and I' are equivalent. Let $k \in \mathbb{N}$, and let us prove that there exists a solution S of $\mathcal{T}$ with $|S| \geq k$ iff there exists a solution S' of $\mathcal{T}'$ with $|S'| \geq k$. Observe that the $\Leftarrow$ direction is obvious as $\mathcal{T}'$ is a subtournament of $\mathcal{T}$. Let us now prove the $\Rightarrow$ direction. Let S be a solution of $\mathcal{T}$ with $|S| \geq k$. Let $S = S_{(0,0)} \cup S_1$ with $S_{(0,0)} = \{t \in S : t = (h(a), v, t(a)) \text{ with } v \in V_{(0,0)}, a \in \overleftarrow{A}(\mathcal{T})\}$ and $S_1 = S \setminus S_{(0,0)}$. Observe that $V(S_1) \cap V_{(0,0)} = \emptyset$, implying $V(S_1) \subseteq V_{\neq(0,0)}$. For any $i \in [2]$, let $S^i_{(0,0)} = \{t \in S_{(0,0)} : t = (h(a), v, t(a)) \text{ with } v \in V_{(0,0)}, v_a^1 \in A_i\}$ be a partition of $S_{(0,0)}$. We define $S' = S_1 \cup S^2_{(0,0)} \cup S'^1_{(0,0)}$, where $S'^1_{(0,0)}$ is defined as follows. For any $v_a^1 \in A_1$, let $v^2_{\mu(a)} \in B_1$ be the vertex associated to v_a^1 in the (A_1, B_1) matching. To any triangle $t = (h(a), v, t(a)) \in S^1_{(0,0)}$ we associate a triangle $f(t) = (h(a), v_{\mu(a)}, t(a)) \in S'^1_{(0,0)}$, where by definition $v_{\mu(a)} \in X_1$. For the sake of uniformity we also say that any $t \in S_1 \cup S^2_{(0,0)}$ is associated to $f(t) = t$.

Let us now verify that triangles of S' are vertex disjoint by verifying that triangles of $S'^1_{(0,0)}$ do not intersect another triangle of S'. Let $f(t) = (h(a), v_{\mu(a)}, t(a)) \in S'^1_{(0,0)}$. Observe that $h(a)$ and $t(a)$ cannot belong to any other triangle $f(t')$ of S' as for any $f(t'') \in S'$, $V(f(t'')) \cap V_{\neq(0,0)} = V(t'') \cap V_{\neq(0,0)}$ (remember that we use the same notation $V_{\neq(0,0)}$ to denote vertices of degree $(0,0)$ in $\mathcal{T}$ and $\mathcal{T}'$). Let us now consider $v_{\mu(a)}$. For any $f(t') \in S_1$, as $V(f(t')) \cap V_{(0,0)} = \emptyset$ we have $v_{\mu(a)} \notin V(f(t'))$. For any $f(t') = (h(a'), v_l, t(a')) \in S^2_{(0,0)}$, we know by definition that $v^1_{a'} \in A_2$, implying that $v_l^2 \in B_2$ (and $v_l \in X_2$) as $N(A_2) \subseteq B_2$ and thus that $v_l \neq v_{\mu(a)}$. Finally, for any $f(t') = (h(a'), v_l, t(a')) \in S'^1_{(0,0)}$, we know that $v_l = v_{\mu(a')}$, where $a \neq a'$, leading to $v_l \neq v_{\mu(a)}$ as μ is a matching. ◀

Using the previous result we can provide a $\mathcal{O}(k)$ vertices kernel for C_3-Packing-T restricted to sparse tournaments.

▶ **Theorem 12** ($\star$). *C_3-Packing-T restricted to sparse tournaments admits a polynomial kernel with $\mathcal{O}(k)$ vertices, where k is the size of the solution.*

4.2 No (generalised) kernel in $\mathcal{O}(k^{2-\epsilon})$

In this section we provide an OR-cross composition (see [4] where we recall the definition) from C_3-Perfect-Packing-T restricted to instances of Theorem 7 to C_3-Perfect-Packing-T parameterized by the total number of vertices.

Definition of the instance selector. The next lemma build a special tournament, called an *instance selector* that will be useful to design the cross composition.

▶ **Lemma 13** (⋆). *For any $\gamma = 2^{\gamma'}$ and ω we can construct in polynomial time (in γ and ω) a tournament $P_{\omega,\gamma}$ such that*

- *there exists γ subsets of ω vertices $X^i = \{x^i_j : j \in [\omega]\}$, that we call the distinguished set of vertices, such that*
  - *the X^i have pairwise empty intersection*
  - *for any $i \in [\gamma]$, there exists a packing S of triangles of $P_{\omega,\gamma}$ such that $V(P_{\omega,\gamma}) \setminus V(S) = X^i$ (using this packing of $P_{\omega,\gamma}$ corresponds to select instance i)*
  - *for any packing S of triangles of $P_{\omega,\gamma}$ with $|V(S)| = |V(P_{\omega,\gamma})| - \omega$ there exists $i \in [\gamma]$ such that $V(P_{\omega,\gamma}) \setminus V(S) \subseteq X^i$*
- $|V(P_{\omega,\gamma})| = \mathcal{O}(\omega\gamma)$.

Definition of the reduction. We suppose given a family of t instances $F = \{\mathcal{I}_l, l \in [t]\}$ of C_3-Perfect-Packing-T restricted to instances of Theorem 7 where $\mathcal{I}_l$ asks if there is a perfect packing in $\mathcal{T}_l = L_l K_l$. We chose our equivalence relation of the cross-composition such that there exist n and m such that for any $l \in [t]$ we have $|V(L_l)| = n$ and $|V(K_l)| = m$. We can also copy some of the t instances such that t is a square number and $g = \sqrt{t}$ is a power of two. We reorganize our instances into $F = \{\mathcal{I}_{(p,q)} : 1 \leq p, q \leq g\}$ where $\mathcal{I}_{(p,q)}$ asks if there is a perfect packing in $\mathcal{T}_{(p,q)} = L_p K_q$. Remember that according to Theorem 7, all the L_p are equals, and all the K_q are equals. We point out that the idea of using a problem on "bipartite" instances to allow encoding t instances on a "meta" bipartite graph $G = (A, B)$ (with $A = \{A_i, i \in \sqrt{t}\}$, $B = \{B_i, i \in \sqrt{t}\}$) such that each instance p, q is encoded in the graph induced by $G[A_i \cup B_i]$ comes from [8]. We refer the reader to Figure 3 which represents the different parts of the tournament. We define a tournament $G = LM_G\tilde{L}\tilde{M}_G P_{(n,g)}$, where $L = L_1 \dots L_g$, $\tilde{M}_G$ is a set of n vertices of degree $(0, 0)$, M_G is a set of $(g-1)n$ vertices of degree $(0, 0)$, $\tilde{L} = \tilde{L}_1 \dots \tilde{L}_g$ where each $\tilde{L}_p$ is a set of n vertices, and $P_{(n,g)}$ is a copy of the instance selector of Lemma 13. Then, for every $p \in [g]$ we add to G all the possible n^2 backward arcs going from $\tilde{L}_p$ to L_p. Finally, for every distinguished set X^p of $P_{(n,g)}$ (see in Lemma 13), we add all the possible n^2 backward arcs from X^p to $\tilde{L}_p$.

Now, in a symmetric way we define a tournament $D = KM_D\tilde{K}\tilde{M}_D P'_{(m,g)}$, where $K = K_1 \dots K_g$, $\tilde{M}_D$ is a set of m vertices of degree $(0, 0)$, M_D is a set of $(g-1)m$ vertices of degree $(0, 0)$, $\tilde{K} = \tilde{K}_1 \dots \tilde{K}_g$ where each $\tilde{K}_q$ is a set of m vertices, and $P'_{(m,g)}$ is a copy of the instance selector of Lemma 13. Then, for every $q \in [g]$ we add to G all the m^2 possible backward arcs going from $\tilde{K}_p$ to K_p. For every distinguished set X'^q of $P'_{(m,g)}$ we also add all the possible m^2 backward arcs from X'^q to $\tilde{K}_q$. Finally, we define $\mathcal{T} = GD$. Let us add some backward arcs from D to G. For any p and q with $1 \leq p, q \leq g$, we add backward arcs from K_q to L_p such that $\mathcal{T}[K_q L_p]$ corresponds to $\mathcal{T}_{(p,q)}$. Notice that this is possible as for any fixed p, all the $\mathcal{T}_{(p,q)}, q \in [g]$ have the same left part L_p, and the same goes for any fixed right part.

Restructuration lemmas. Given a set of triangles S we define $S_{\subseteq P'} = \{t \in S | V(t) \subseteq P'_{(m,g)}\}$, $S_{\subseteq P} = \{t \in S : V(t) \subseteq P_{(n,g)}\}$, $S_{\tilde{M}_D} = \{t \in S : V(t)$ intersects $\tilde{K}$, $\tilde{M}_D$ and $P'_{m,g}\}$, $S_{M_D} = \{t \in S : V(t)$ intersects K, M_D and $\tilde{K}\}$, $S_{\tilde{M}_G} = \{t \in S : V(t)$ intersects $\tilde{L}$, $\tilde{M}_G$ and $P_{n,g}\}$, $S_{M_G} = \{t \in S : V(t)$ intersects L, M_G and $\tilde{L}\}$, $S_D = \{t \in S : V(t) \subseteq V(D)\}$, $S_G = \{t \in S : V(t) \subseteq V(G)\}$, and $S_{GD} = \{t \in S : V(t)$ intersects $V(G)$ and $V(D)\}$. Notice that S_G, S_{GD}, S_D is a partition of S.

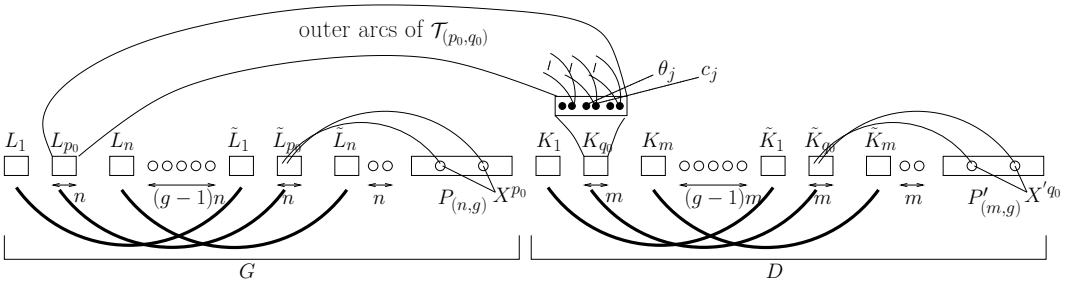


Figure 3 A example of the weak composition. All depicted arcs are backward arcs. Bold arcs represents the n^2 (or m^2) possible arcs between the two groups.

▶ **Claim 14.** *If there exists a perfect packing S of $\mathcal{T}$, then $|S_{\tilde{M}_D}| = m$ and $|S_{M_D}| = (g-1)m$. This implies that $V(S_{\tilde{M}_D} \cup S_{M_D}) \cap V(\tilde{K}) = V(\tilde{K})$, meaning that the vertices of $\tilde{K}$ are entirely used by $S_{\tilde{M}_D} \cup S_{M_D}$.*

Proof. We have $|S_{\tilde{M}_D}| \leq m$ since $|\tilde{M}_D| = m$. We obtain the equality since the vertices of $\tilde{M}_D$ only lie in the span of backward arcs from $P'_{m,g}$ to $\tilde{K}$, and they are not the head or the tail of a backward arc in $\mathcal{T}$. Thus, the only way to use vertices of $\tilde{M}_D$ is to create triangles in $S_{\tilde{M}_D}$, implying $|S_{\tilde{M}_D}| \geq m$. Using the same kind of arguments we also get $|S_{M_D}| = (g-1)m$. As $|V(\tilde{K})| = gm$ we get the last part of the claim. ◀

▶ **Claim 15.** *If there exists a perfect packing S of $\mathcal{T}$, then there exists $q_0 \in [g]$ such that $\tilde{K}_S = \tilde{K}_{q_0}$, where $\tilde{K}_S = \tilde{K} \cap V(S_{\tilde{M}_D})$.*

Proof. Let $S_{P'}$ be the triangles of S with at least one vertex in $P'_{m,g}$. As according to Claim 14 vertices of $\tilde{K}$ are entirely used by $S_{\tilde{M}_D} \cup S_{M_D}$, the only way to consume vertices of $P'_{m,g}$ is by creating local triangles in $P'_{m,g}$ or triangles in $S_{\tilde{M}_D}$. In particular, we cannot have a triangle (u,v,w) with $\{u,v\} \subseteq \tilde{K}$ and $w \in P'_{m,g}$, or with $u \in \tilde{K}$ and $\{v,w\} \subseteq P'_{m,g}$. More formally, we get the partition $S_{P'} = S_{\subseteq P'} \cup S_{\tilde{M}_D}$. As S is a perfect packing and uses in particular all vertices of $P'_{m,g}$ we get $|V(S_{P'})| = |V(P'_{m,g})|$, implying $|V(S_{\subseteq P'})| = |V(P'_{m,g})| - m$ by Claim 14. By Lemma 13, this implies that there exists $q_0 \in [g]$ such that $X' \subseteq X'^{q_0}$ where $X' = V(P'_{m,g}) \setminus V(S_{\subseteq P'})$. As X' are the only remaining vertices that can be used by triangles of $S_{\tilde{M}_D}$, we get that the m triangles of $S_{\tilde{M}_D}$ are of the form (u,v,w) with $u \in \tilde{K}_{q_0}$, $v \in \tilde{M}_D$, and $w \in X'$. ◀

▶ **Claim 16.** *If there exists a perfect packing S of $\mathcal{T}$, then there exists $q_0 \in [g]$ such that $V(S_{P'} \cup S_{\tilde{M}_D} \cup S_{M_D}) = V(D) \setminus K_{q_0}$.*

Proof. By Claim 14 we know that $|S_{M_D}| = (g-1)m$. As by Claim 15 there exists $q_0 \in [g]$ such that $\tilde{K}_S = \tilde{K}_{q_0}$, we get that the $(g-1)m$ triangles of S_{M_D} are of the form (u,v,w) with $u \in K \setminus K_{q_0}$, $v \in M_D$, and $w \in \tilde{K} \setminus \tilde{K}_{q_0}$. ◀

▶ **Lemma 17 (⋆).** *If there exists a perfect packing S of $\mathcal{T}$, then $V(S_{GD}) \cap V(G) \subseteq V(L)$. Informally, triangles of S_{GD} do not use any vertex of $M_G, \tilde{L}, \tilde{M}_T$ and $P_{n,g}$.*

Lemma 17 will allow us to prove Claims 18, 19 and 20 using the same arguments as in the right part (D) of the tournament as all vertices of $M_G, \tilde{L}, \tilde{M}_T$ and $P_{n,g}$ must be used by triangles in S_G.

▶ **Claim 18 (⋆).** *If there exists a perfect packing S of $\mathcal{T}$, then $|S_{\tilde{M}_G}| = n$ and $|S_{M_G}| = (g-1)n$. This implies that $V(S_{\tilde{M}_G} \cup S_{M_G}) \cap V(\tilde{L}) = V(\tilde{L})$, meaning that vertices of $\tilde{L}$ are entirely used by $S_{\tilde{M}_G} \cup S_{M_G}$.*

▶ **Claim 19** (⋆). *If there exists a perfect packing S of $\mathcal{T}$, then there exists $p_0 \in [g]$ such that $\tilde{L}_S = \tilde{L}_{p_0}$, where $\tilde{L}_S = \tilde{L} \cap V(S_{\tilde{M}_G})$.*

▶ **Claim 20** (⋆). *If there exists a perfect packing S of $\mathcal{T}$, then there exists $p_0 \in [g]$ such that $V(S_P \cup S_{\tilde{M}_G} \cup S_{M_G}) = V(G) \setminus L_{p_0}$.*

We are now ready to state our final claim is now straightforward as according Claim 16 and 20 we can define $S_{(p_0,q_0)} = S \setminus ((S_{P'} \cup S_{\tilde{M}_D} \cup S_{M_D}) \cup (S_P \cup S_{\tilde{M}_G} \cup S_{M_G}))$.

▶ **Claim 21.** *If there exists a perfect packing S of $\mathcal{T}$, there exists $p_0, q_0 \in [g]$ and $S_{(p_0,q_0)} \subseteq S$ such that $V(S_{(p_0,q_0)}) = V(\mathcal{T}_{(p_0,q_0)})$ (or equivalently such that $S_{(p_0,q_0)}$ is a perfect packing of $\mathcal{T}_{(p_0,q_0)}$).*

Proof of the weak composition

▶ **Theorem 22.** *For any $\epsilon > 0$, C_3-Perfect-Packing-T (parameterized by the total number of vertices N) does not admit a polynomial (generalized) kernelization with size bound $\mathcal{O}(N^{2-\epsilon})$ unless NP $\subseteq$ coNP / Poly.*

Proof. Given t instances $\{\mathcal{I}_l\}$ of C_3-Perfect-Packing-T restricted to instances of Theorem 7, we define an instance $\mathcal{T}$ of C_3-Perfect-Packing-T as defined in Section 4. We recall that $g = \sqrt{t}$, and that for any $l \in [t]$, $|V(L_l)| = n$ and $|V(K_l)| = m$. Let $N = |V(\mathcal{T})|$. As $N = |V(P'_{(m,g)})| + m + (g-1)m + 2mg + |V(P_{(n,g)})| + n + (g-1)n + 2ng$ and $|V(P_{(\omega,\gamma)})| = O(\omega\gamma)$ by Lemma 13, we get $N = \mathcal{O}(g(n+m)) = \mathcal{O}(t^{\frac{1}{2+o(1)}} \max(|\mathcal{I}_l|))$. Let us now verify that there exists $l \in [t]$ such that $\mathcal{I}_l$ admits a perfect packing iff $\mathcal{T}$ admits a perfect packing. First assume that there exist $p_0, q_0 \in [g]$ such that $\mathcal{I}_{(p_0,q_0)}$ admits a perfect packing. By Lemma 21, there is a packing $S_{P'}$ of $P'_{(m,g)}$ such that $V(S_{P'}) = V(P'_{(m,g)}) \setminus X'^{q_0}$. We define a set $S_{\tilde{M}_D}$ of m vertex disjoint triangles of the form (u,v,w) with $u \in \tilde{L}_{q_0}, v \in \tilde{M}_D, w \in X'^{q_0}$. Then, we define a set S_{M_D} of $(g-1)m$ vertex disjoint triangles of the form (u,v,w) with $u \in L \setminus L_{q_0}, v \in M_D, w \in \tilde{L} \setminus \tilde{L}_{q_0}$. In the same way we define S_P, $S_{\tilde{M}_G}$ and S_{M_G}. Observe that $V(\mathcal{T}) \setminus ((S_{P'} \cup S_{\tilde{M}_D} \cup S_{M_D}) \cup (S_P \cup S_{\tilde{M}_G} \cup S_{M_G})) = K_{q_0} \cup L_{p_0}$, and thus we can complete our packing into a perfect packing of $\mathcal{T}$ as $\mathcal{I}_{(p_0,q_0)}$ admits a perfect packing. Conversely if there exists a perfect packing S of $\mathcal{T}$, then by Claim 21 there exists $p_0, q_0 \in [g]$ and $S_{(p_0,q_0)} \subseteq S$ such that $V(S_{(p_0,q_0)}) = V(\mathcal{T}_{(p_0,q_0)})$, implying that $\mathcal{I}_{(p_0,q_0)}$ admits a perfect packing. ◀

▶ **Corollary 23.** *For any $\epsilon > 0$, C_3-Packing-T (parameterized by the size k of the solution) does not admit a polynomial kernel with size $\mathcal{O}(k^{2-\epsilon})$ unless NP $\subseteq$ coNP / Poly.*

5 Conclusion and open questions

Concerning approximation algorithms for C_3-Packing-T restricted to sparse instances, we have provided a $(1 + \frac{6}{c+5})$-approximation algorithm where c is a lower bound of the *minspan* of the instance. On the other hand, it is not hard to solve by dynamic programming C_3-Packing-T for instances where *maxspan* is bounded above. Using these two opposite approaches it could be interesting to derive an approximation algorithm for C_3-Packing-T with factor better than 4/3 even for sparse tournaments.

Concerning FPT algorithms, the approach we used for sparse tournament (reducing to the case where $m = \mathcal{O}(k)$ and apply the $\mathcal{O}(m)$ vertices kernel) cannot work for the general case. Indeed, if we were able to sparsify the initial input such that $m' = \mathcal{O}(k^{2-\epsilon})$, applying the kernel in $\mathcal{O}(m')$ would lead to a tournament of total bit size (by encoding the two endpoint

of each arc) $\mathcal{O}(m'log(m')) = \mathcal{O}(k^{2-\epsilon})$, contradicting Corollary 23. Thus the situation for C_3-Packing-T could be as in vertex cover where there exists a kernel in $\mathcal{O}(k)$ vertices, derived from [15], but the resulting instance cannot have $\mathcal{O}(k^{2-\epsilon})$ edges [8]. So it is challenging question to provide a kernel in $\mathcal{O}(k)$ vertices for the general C_3-Packing-T problem.

References

1. Faisal N. Abu-Khzam. A quadratic kernel for 3-set packing. In *International Conference on Theory and Applications of Models of Computation*, pages 81–87. Springer, 2009.
2. Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 2012.
3. Piotr Berman and Marek Karpinski. On some tighter inapproximability results. In *International Colloquium on Automata, Languages, and Programming*, pages 200–209. Springer, 1999.
4. Stephane Bessy, Marin Bougeret, and Jocelyn Thiebaut. Triangle packing in (sparse) tournaments: approximation and kernelization. Technical report, HAL LIRMM, lirmm-01550313, v1, 2017. URL: `https://hal-lirmm.ccsd.cnrs.fr/lirmm-01550313`.
5. Mao-Cheng Cai, Xiaotie Deng, and Wenan Zang. A min-max theorem on feedback vertex sets. *Mathematics of Operations Research*, 27(2):361–371, 2002.
6. Pierre Charbit, Stéphan Thomassé, and Anders Yeo. The minimum feedback arc set problem is NP-hard for tournaments. *Combinatorics, Probability and Computing*, 16(01):1–4, 2007.
7. Marek Cygan. Improved approximation for 3-dimensional matching via bounded pathwidth local search. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 509–518. IEEE, 2013.
8. Holger Dell and Dániel Marx. Kernelization of packing problems. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete algorithms, SODA'12*, 2012.
9. Venkatesan Guruswami, C. Pandu Rangan, Maw-Shang Chang, Gerard J. Chang, and C. K. Wong. The vertex-disjoint triangles problem. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 26–37. Springer, 1998.
10. Danny Hermelin and Xi Wu. Weak compositions and their applications to polynomial lower bounds for kernelization. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 104–113. Society for Industrial and Applied Mathematics, 2012.
11. Bart M. P. Jansen and Astrid Pieterse. Sparsification upper and lower bounds for graph problems and Not-All-Equal SAT. *Algorithmica*, pages 1–26, 2015.
12. Claire Kenyon-Mathieu and Warren Schudy. How to rank with few errors. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 95–103. ACM, 2007.
13. Matthias Mnich, Virginia Vassilevska Williams, and László A. Végh. A 7/3-Approximation for Feedback Vertex Sets in Tournaments. In *24th Annual European Symposium on Algorithms, ESA 2016*, pages 67:1–67:14, 2016.
14. Hannes Moser. A problem kernelization for graph packing. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 401–412. Springer, 2009.
15. George L. Nemhauser and Leslie E. Trotter Jr. Properties of vertex packing and independence system polyhedra. *Mathematical Programming*, 6(1):48–61, 1974.

Improved Algorithm for Dynamic b-Matching

Sayan Bhattacharya[1], Manoj Gupta[2], and Divyarthi Mohan[3]

1 **University of Warwick, Coventry, UK**
S.Bhattacharya@warwick.ac.uk
2 **IIT Gandhinagar, Gandhinagar, India**
gmanoj@iitgn.ac.in
3 **Princeton University, Princeton, NJ, USA**
dm23@cs.princeton.edu

Abstract

Recently there has been extensive work on maintaining (approximate) maximum matchings in dynamic graphs. We consider a generalisation of this problem known as the *maximum b-matching*: Every node v has a positive integral capacity b_v, and the goal is to maintain an (approximate) maximum-cardinality subset of edges that contains at most b_v edges incident on every node v. The maximum matching problem is a special case of this problem where $b_v = 1$ for every node v.

Bhattacharya, Henzinger and Italiano [ICALP 2015] showed how to maintain a $O(1)$ approximate maximum b-matching in a graph in $O(\log^3 n)$ amortised update time. Their approximation ratio was a large (double digit) constant. We significantly improve their result both in terms of approximation ratio as well as update time. Specifically, we design a randomised dynamic algorithm that maintains a $(2+\epsilon)$-approximate maximum b-matching in expected amortised $O(1/\epsilon^4)$ update time. Thus, for every constant $\epsilon \in (0,1)$, we get expected amortised $O(1)$ update time. Our algorithm generalises the framework of Baswana, Gupta, Sen [FOCS 2011] and Solomon [FOCS 2016] for maintaining a maximal matching in a dynamic graph.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases dynamic data structures, graph algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.15

1 Introduction

In dynamic graph algorithms, we want to maintain a solution to an optimisation problem on an input graph that is undergoing a sequence of edge insertions/deletions. The time taken to modify the solution after an *update* (edge insertion/deletion) is called the *update time* of the dynamic data structure. The challenge is to design dynamic data structures whose update times are significantly faster than recomputing the solution from scratch after each update using the best static algorithm. In recent years, there has been extensive work on dynamic graph algorithms for (approximate) maximum matching [10, 1, 12, 8, 3, 9, 2, 11, 6, 5]. A matching $M \subseteq E$ in a graph $G = (V, E)$ is a subset of edges that do not share a common endpoint. In this problem, the goal is to maintain a matching of (approximately) maximum size in an input graph undergoing a sequence of edge insertions/deletions.

The first significant result on dynamic matching was due to Onak and Rubinfeld [10], who gave a randomised data structure with approximation ratio $O(1)$ and amortised update time of $O(\log^2 n)$. Baswana, Gupta and Sen [1] improved this approximation ratio to 2 and the amortised update time to $O(\log n)$. Very recently, in a breakthrough result Solomon [12] built upon the algorithmic framework of Baswana, Gupta and Sen [1] to present a randomised data structure with approximation ratio 2 and *constant amortised update time*. This is the

first paper to achieve constant (and hence optimal) update time for *any* graph problem. All the data structures described so far are randomised. There are deterministic data structures for this problem, with $(2+\epsilon)$-approximation ratio and $O(\text{poly}\log n)$ amortised update time, that are due to Bhattacharya, Henzinger and Italiano [5] and Bhattacharya, Henzinger and Nanongkai [6]. They show how to maintain a $(2+\epsilon)$-approximate maximum *fractional* matching in $O(\log n/\epsilon^2)$ amortised update time. Then they *round* these fractional weights deterministically in a dynamic setting to get an integral matching.

We focus on a generalisation of the dynamic matching problem. We are given an input graph $G=(V,E)$ with $n=|V|$ nodes. At each time-step, an edge is inserted into/deleted from the graph. Each node $v \in V$ has a positive integral *capacity* b_v. The node-set V and the capacities $\{b_v\}$ remain unchanged over time. A b-matching is a subset of edges $\mathcal{M} \subseteq E$ that contains at most b_v edges incident on every node $v \in V$. Note that if $b_v = 1$ for every node $v \in V$, then $\mathcal{M}$ is a matching. Our goal is to maintain a b-matching of (approximately) maximum size in this dynamic graph G. This problem was first considered by Bhattacharya, Henzinger and Italiano [4]. Extending the dynamic data structure for fractional matching [5], they first showed how to maintain a fractional b-matching. Then they randomly sampled the edges from this fractional b-matching, with probabilities proportional to their edge-weights, and got an integral b-matching. This leads to a randomised data structure with a (large, double digit) constant approximation ratio and $O(\log^3 n)$ amortised update time with high probability. Recently, Gupta et. al [7] improved the update time in [4] for the set cover problem. However, their result does not imply our b-matching result.

Our result. We significantly improve upon the previous result on dynamic b-matching [4], both in terms of the approximation ratio and update time. Specifically, we present a randomised data structure with approximation ratio $(2+\epsilon)$ and expected amortised update time $O(1/\epsilon^4)$. Note that our update time does *not* depend on the capacities $\{b_v\}$. On the other hand, similar to the previous paper [4], we assume an oblivious adversary.

Our technique. We build upon the work of Baswana, Gupta, Sen [1] and Solomon [12]. Before proceeding any further, we briefly review their framework for dynamic matching. A matching $M \subseteq E$ is *maximal* if every unmatched edge $(u,v) \in E \setminus M$ has at least one matched endpoint. The size of a maximal matching is a 2-approximation to the size of the maximum matching. Note that we can easily maintain a maximal matching in a dynamic graph in $O(d)$ update time, where d is an upper bound on the maximum degree of a node. This holds since after the insertion/deletion of an edge (u,v), we only need to check the neighbours of u and v to preserve maximality, and there are at most $O(d)$ such neighbours. To improve the update time, the papers [1, 10, 12] use the following idea. We pick a neighbour y of x uniformly at random from the set N_x, where N_x is the set of all neighbours of x, and add the edge (x,y) to the matching M. If the sequence of edge insertions/deletions in G is oblivious to the random bits used by the algorithm, then in expectation this edge (x,y) will not be deleted from G before half of the current edges in N_x gets deleted. And when such an edge $(x,z) \in N_x \setminus \{(x,y)\}$ gets deleted, the algorithm does not need to perform any computation for the node x (since the node remains matched in M). Thus, although the algorithm takes $O(|N_x|)$ time to find a new *mate* for x, it does not need to do anything for the next $|N_x|/2$ edge deletions incident on x, in expectation. This helps bring down the amortised update time to $O(1)$. This is the high level idea behind the dynamic algorithms in [1, 12]. The actual algorithms, however, are significantly more complicated, for the following reason. In the preceding summary, we said that the node x picks a neighbour y uniformly at random from

the set N_x. But what if the node y itself is matched to some other node z (i.e., $(y,z) \in M$)? In that case, in order to add the edge (x,y) to the matching M, we first need to unmatch the edge (y,z) by removing it from M. But this means that we will now need to find a new mate for z. In general, this can lead to a long chain of edges alternately being matched and unmatched. To address this concern, the papers [1, 12] construct a *hierarchical partition* of the node-set into $O(\log n)$ levels. Suppose that a node x is at level i, and want to find a new mate for x. The papers [1, 12] ensure that x has roughly α^i many neighbours at strictly lower levels $\{0, \ldots, i-1\}$, for some constant $\alpha > 1$. We now pick a node y uniformly at random *only from* these neighbours of x at levels $j < i$. The papers further ensure that every matched edge has both its endpoints at the same level. Hence, the node (say z) that is matched to y will have $\ell(z) = \ell(y) < i = \ell(x)$. We now match the edge (x,y), bring the node y up to level i, and unmatch the edge (y,z). The node z now has to find a new mate for itself. But note that the level of z is strictly less than the level of x. Hence, this chain of alternate matchings and unmatchings of edges cannot go on for more than $O(\log n)$ levels.

In order to extend the above framework to maximum b-matching, we have to overcome several technical difficulties. First, since a node can have multiple matched edges incident on it in a b-matching, we can no longer ensure that both the endpoints of every matched edge are on the same level. Instead, we maintain an invariant which states that if a node v has b_v matched edges incident on it, then at least one of these matched edges, say (u,v), must have its other endpoint u at a level that is not larger than the level of v (see Invariant 3). Second, unlike the papers [1, 12], we can no longer ensure that if a node z becomes unmatched while we are finding a new mate for a different node x, then $\ell(x) > \ell(z)$. Indeed, there are instances in our algorithm where both x and z can be at the same level, and we need to show that this situation does not occur too often. Finally, in the papers [1, 12], at any point in time there can be multiple nodes in the hierarchical partition that are *dirty*, meaning that they violate one of the invariants. The algorithms in [1, 12] run a WHILE loop, and each iteration of the WHILE loop picks *any arbitrary* dirty node and restores all the invariants it ought to satisfy (this can lead to other nodes becoming dirty). In sharp contrast, the WHILE loop in our dynamic algorithm has to pick the dirty nodes *in a specific order*, preferring one type of dirty nodes over another. This preferential ordering among the dirty nodes turns out to be crucial in analysing the amortised update time of our algorithm. See Sections 2 and 3 for details.

2 Our Dynamic Algorithm for Maximum b-Matching

Let $G = (V, E)$ denote the input graph whose edge-set E changes dynamically. Every node $v \in V$ has a positive integral *capacity* b_v associated with it. The node-set V and the capacities $\{b_v\}$ do not change over time. Let $n = |V|$ be the number of nodes in G. A subset of edges $\mathcal{M} \subseteq E$ is a *b-matching* iff for all nodes $v \in V$, we have $|\{(u,v) \in \mathcal{M} \mid (u,v) \in E\}| \leq b_v$. We say that an edge $e \in E$ is *matched* in $\mathcal{M}$ iff $e \in \mathcal{M}$.

Throughout this paper, we fix any constant $\epsilon \in (0, 1/2)$ and define $\alpha = 5/\epsilon$. We will maintain a partition of the node-set V into $L+2$ *levels* $\{-1, 0, \ldots, L\}$, where $L = \lceil \log_\alpha n \rceil$. Let $\ell(v) \in \{-1, \ldots, L\}$ denote the level of a node v. The value of $\ell(v)$ changes over time. In our dynamic algorithm, whenever a node v moves to a level j, we will require that it has at most $2b_v \cdot \alpha^{j+1}$ neighbours u with $\ell(u) \leq j$. Further, any node v at level -1 has at most $O(b_v \cdot \alpha^0) = O(b_v)$ many neighbours at level -1.

The *level of an edge* is the maximum level among its endpoints, i.e., $\ell(u,v) = \max(\ell(u), \ell(v))$ for all $(u,v) \in E$. For every node $v \in V$ and every level $j \in [-1, L]$, we let $\mathcal{E}_v^j = \{(u,v) \in E : \ell(u,v) = j\}$ be the set of edges incident on v that are at level j. Since $\ell(u,v) \geq \ell(v)$ for all

edges $(u, v) \in E$, it follows that $\mathcal{E}_v^j = \emptyset$ for all levels $j < \ell(v)$. We say that an edge $(u, v) \in E$ is *owned by* its endpoint v iff $\ell(u) < \ell(v)$. Equivalently, we say that v is the *owner* of the edge (u, v). We let $\mathcal{O}_v = \{(u, v) \in E : \ell(u) < \ell(v)\}$ denote the set of edges owned by $v \in V$ (Note that if both the end-points are at the same level then no vertex owns the edge). In a b-matching $\mathcal{M} \subseteq E$, we let $\mathcal{M}_v = \{(u, v) \in \mathcal{M}\}$ denote the set of all matched edges incident on a node $v \in V$. For any level $i \in [-1, L]$, we define $\mathcal{M}_v(i) = \{(u, v) \in \mathcal{M}_v : \ell(u, v) = i\}$ as the set of all matched edges incident on a node v at level i. We say that a node $v \in V$ is FULL in a b-matching $\mathcal{M} \subseteq E$ iff $|\mathcal{M}_v| = b_v$, and v is DEFICIENT in $\mathcal{M}$ iff $|\mathcal{M}_v| < (1 - \epsilon)b_v$. The node v is NON-DEFICIENT in $\mathcal{M}$ iff $|\mathcal{M}_v| \geq (1 - \epsilon)b_v$. Finally, for every node $v \in V$, if v is FULL, then we define $\text{BASE}(v)$ to be the smallest level j for which there is a matched edge $(u, v) \in \mathcal{M}_v$ at level $\ell(u, v) = j$. Else if v is not FULL, then we set $\text{BASE}(v) = \ell(v)$.

We maintain a b-matching $\mathcal{M} \subseteq E$ in G satisfying the three invariants below. The approximation guarantee of the algorithm follows from the first two invariants (see Theorem 4). Invariant 3 will be useful in bounding the amortised update time. The main result in this paper follows from Theorems 4, 5.

▶ **Invariant 1.** *Every node $v \in V$ at level $\ell(v) \geq 0$ is* NON-DEFICIENT.

▶ **Invariant 2.** *Every unmatched edge with level -1 has at least one* NON-DEFICIENT *endpoint.*

▶ **Invariant 3.** *For every node $v \in V$, we have $|\{(u, v) \in \mathcal{M} : \ell(u) > \ell(v)\}| < b_v$.*

We will assume that the initial graph is empty. So, all the three invariants hold. We now compare our invariants with those in ([1], [12]), that is in the maximal matching case when $b_v = 1$ for all $v \in V$. NON-DEFICIENT vertices are *free* vertices in this simple case. The first invariant then say that all the vertices at level ≥ 0 are matched . The second invariant says that each un-matched edge at level -1 has both endpoints free. In ([1],[12]), there can be no edge at level -1, since only *free* vertices settle at level -1. That is why the corresponding invariant for maximal matching is that *there is no edge with level* -1. We have generalised this invariant for b-matching. The last invariant implies that if at any point of time all the other endpoints of the matched edges incident on v have levels greater than v, then v has to move to a higher level. In the simple maximal matching case, this implies that end-points of the matched edges should always be at the same level. The reader can check that these three invariants are maintained in ([1],[12]), and here we generalise them for b-matching.

▶ **Theorem 4.** *Invariants 1, 2 imply that $\mathcal{M}$ is a $(2 + \epsilon)$-approximate maximum b-matching.*

Proof. (Sketch) Invariants 1, 2, ensure that $\mathcal{M}$ is *almost maximal*, as every unmatched edge has one endpoint x with $|\mathcal{M}_x| \geq (1 - \epsilon)b_x$. This implies $(2 + \epsilon)$-approximation. ◀

▶ **Theorem 5.** *There is a randomised algorithm that maintains a b-matching in a dynamic graph satisfying Invariants 1, 2, 3. It has an amortised update time of $O(1/\epsilon^4)$ in expectation.*

2.1 Handling the insertion/deletion of an edge

Data structures. Each node $x \in V$ maintains the edge-sets $\mathcal{M}_x, \mathcal{M}_x(j), \mathcal{E}_x^j$ and $\mathcal{O}_x$ as doubly linked lists, and the node also maintains the sizes of these sets. From the size of $\mathcal{M}_x$, we can infer whether the node x is DEFICIENT, NON-DEFICIENT or FULL. With appropriate pointers, an edge can be inserted into or deleted from a linked list in $O(1)$ time. Further, each node $x \in V$ maintains its level $\ell(x)$ and the value of $\text{BASE}(x)$.

```
01. WHILE the set of DIRTY nodes is nonempty
02.     IF the set of FULL-FROM-ABOVE nodes is nonempty, THEN
03.         Let x be a FULL-FROM-ABOVE node with the largest value of BASE(x).
04.         Call the subroutine FIX-DIRTY(x).   // See Section 2.1.1.
05.     ELSE
06.         Let x be any DIRTY node that is DEFICIENT at level ℓ(x) ≥ 0.
07.         Call the subroutine FIX-DIRTY(x).   // See Section 2.1.2.
```

Figure 1 Fixing the dirty nodes.

Insertion/deletion of an edge (u, v). When an edge (u, v) is inserted into or deleted from the graph $G = (V, E)$, we first update the relevant data structures in $O(1)$ time. For the time being, to highlight the main idea behind our algorithm, we assume that the insertion/deletion of the edge (u, v) does not lead to a violation of Invariant 2 (which pertains to the subgraph induced by the nodes at level -1). In Section 2.1.4, we will show how to handle the nodes in level -1 and how to maintain Invariant 2. Right now we focus on maintaining the remaining two invariants. Accordingly, suppose that after the insertion/deletion of the edge (u, v), at least one of its endpoints violates Invariant 1 or 3. In general, if a node $x \in V$ violates Invariant 1 or 3, then we say that the node x is DIRTY. A dirty node is either DEFICIENT (if it violates Invariant 1) or FULL-FROM-ABOVE (if it violates Invariant 3). Thus, a node x is FULL-FROM-ABOVE iff $\text{BASE}(x) > \ell(x)$ and $|\mathcal{M}_x| = b_x$. Due to the insertion/deletion of an edge, its endpoints can become DIRTY. To ensure that no node remains DIRTY, we call the following subroutine in Figure 1.

We highlight two important aspects of this subroutine. First, during an iteration of the WHILE loop in Figure 1, the node x can move to a new level due to the call to FIX-DIRTY(x). Immediately after the call to FIX-DIRTY(x), the node x is no longer DIRTY. But the call to FIX-DIRTY(x) might lead to some other nodes becoming DIRTY, and these newly DIRTY nodes will be handled in subsequent iterations of the WHILE loop. Second, the WHILE loop in Figure 1 always gives preference to the FULL-FROM-ABOVE nodes over the DIRTY nodes that are DEFICIENT at a nonnegative level. Furthermore, among the FULL-FROM-ABOVE nodes, the WHILE loop picks the one with the largest BASE(.) value. The WHILE loop picks a DIRTY and DEFICIENT node only if there is no FULL-FROM-ABOVE node. This aspect of our algorithm will ensure that Lemma 9 holds, which, in turn, will play a crucial role in the analysis of the amortised update time.

2.1.1 The subroutine FIX-DIRTY(x) on a Full-from-above node x

Suppose that the subroutine FIX-DIRTY(x) is called on a FULL-FROM-ABOVE node x at level $\ell(x) = i$ with $\text{BASE}(x) = j$ (say). Since the node x is FULL-FROM-ABOVE, we have $j > i$. We first move the node x up from level i to level j, and update all the relevant data structures (as described in the beginning of Section 2.1). We next check the number of edges in $\mathcal{E}_x^j$, and, accordingly, we consider two cases.

Case 1. $|\mathcal{E}_x^j| \leq 2b_x \cdot \alpha^{j+1}$. In this case, the node x stays at level j and we terminate the subroutine. At this stage the node x has $|\mathcal{M}_x| = b_x$ and $\text{BASE}(x) = j = \ell(x)$. Hence, the node x is no longer DIRTY.

Case 2. $|\mathcal{E}_x^j| > 2b_x \cdot \alpha^{j+1}$. In this case, we find the *minimum* level $k \in [j+1, L]$ such that the number of edges (x, y) with $\ell(y) \leq k$ lies in the range $(2b_x \cdot \alpha^k, 2b_x \cdot \alpha^{k+1}]$. Such a level exists since $b_x \cdot \alpha^{L+1} \geq n >$ degree of x. We now move the node x from level j to level k. In doing so, x becomes the owner of all the edges whose other endpoints are at level $< k$ and we update all the relevant data structures. Next, we *unmatch* all edges $(x, y) \in \mathcal{M}_x$ whose other endpoints are at levels $\ell(y) < k$, and update all the relevant data structures. Let λ_x be the value of $|\mathcal{M}_x|$ at this stage. We now select $(b_x - \lambda_x)$ edges from $\mathcal{O}_x$ uniformly at random, and add them to $\mathcal{M}$ by calling RANDOM-SETTLE(x)(See Section 2.1.3). At this stage we have $|\mathcal{M}_x| = b_x$ and BASE(x) $= k = \ell(x)$, and hence x is no longer DIRTY. We terminate the subroutine at this point. This procedure can lead to other nodes becoming DIRTY, for two reasons. (1) When we unmatch an edge (x, y) with $\ell(y) < k$, the node y might become DEFICIENT and DIRTY. (2) Suppose that while executing RANDOM-SETTLE(x), we add an edge $(x, y) \in \mathcal{O}_x$ to $\mathcal{M}$. Then the node y can become FULL-FROM-ABOVE (and DIRTY). Else, it might happen that $|\mathcal{M}_y|$ becomes equal to $(b_y + 1)$ after we add the edge (x, y) to $\mathcal{M}$. Then we will need to unmatch some edge $(y, z) \in \mathcal{M}_y$, so as to ensure that $\mathcal{M}$ remains a valid b-matching. But this might lead to z becoming DEFICIENT and DIRTY. These newly DIRTY nodes will be handled in subsequent iterations of the WHILE loop in Figure 1.

▶ **Lemma 6.** *Suppose that we call FIX-DIRTY(x) on a* FULL-FROM-ABOVE *node x, and this moves the node x up from level i to level $k > i$. Then it takes $O(b_x \cdot \alpha^{k+1})$ time.*

Proof (Sketch). Only the edges $(x, y) \in E$ with $\ell(y) \leq k$ get affected as the node x moves to level k. The data structure associated with every other edge remains unchanged. Thus, the total runtime of the subroutine is proportional to the number of edges in $\{(x, y) \in E : \ell(y) \leq k\}$, plus the time taken by the potential call to RANDOM-SETTLE(x). The latter quantity is bounded in Lemma 10. It is easy to check that the former quantity is at most $2b_x\alpha^{k+1}$, for otherwise the node x would have moved up to a level larger than k. ◀

2.1.2 The subroutine FIX-DIRTY(x) on a Deficient node x

Suppose that the subroutine FIX-DIRTY(x) is called on a DEFICIENT node x at level $\ell(x) = j \geq 0$. We first check the number of edges in $\mathcal{E}_x^j$. Accordingly, we consider two cases.

Case 1. $|\mathcal{E}_x^j| > 2b_x \cdot \alpha^{j+1}$. Here, we perform the same steps as in Case 2 in Section 2.1.1.

Case 2. $|\mathcal{E}_x^j| \leq 2b_x \cdot \alpha^{j+1}$. In this case, we try to find a maximal level k in range $[0, j]$ such that the number of edges (x, y) with $\ell(y) < k$ lies in the range $(2b_x \cdot \alpha^k, 2b_x \cdot \alpha^{k+1}]$. If no such level exists, then we set $k = -1$. Next, we move the node x from level j to level k. In doing so, x dis-owns all the edges (and the other endpoints becomes the owner) that are at level $[k, j]$ and we update the relevant data structures. We then unmatch all the edges (x, y) whose other endpoints are at levels $\ell(y) < j$, and update the relevant data structures. Let λ_x be the value of $|\mathcal{M}_x|$ at this stage. If $k \geq 0$, then we call RANDOM-SETTLE(x), which selects $(b_x - \lambda_x)$ edges uniformly at random from $\mathcal{O}_x = \{(x, y) \in E : \ell(y) < k\}$, and adds those edges to $\mathcal{M}$. At this point $|\mathcal{M}_x| = b_x$ and BASE(x) $= k = \ell(x)$, and hence x is no longer DIRTY. So we terminate the subroutine. If $k = -1$, then instead of calling RANDOM-SETTLE(x), we follow the procedure in Section 2.1.4.

▶ **Lemma 7.** *Suppose that we call FIX-DIRTY(x) on a* DEFICIENT *node x at level $j \geq 0$. If the subroutine moves the node x to a level $k > j$, then it takes $O(b_x \cdot \alpha^{k+1})$ time.*

Proof. Exactly similar to the proof of Lemma 6. ◀

01. WHILE $|\mathcal{M}_x| < b_x$
02. Pick a uniformly random edge $(x, y) \in \mathcal{O}_x \setminus \mathcal{M}_x$, and add it to the b-matching. Specifically, set $\mathcal{M} \leftarrow \mathcal{M} \cup \{(x, y)\}$, and update the relevant data structures.
03. IF $|\mathcal{M}_y| = b_y + 1$, THEN
04. Select an edge $(y, z) \in \mathcal{M}_y \setminus \{(x, y)\}$ which minimises the value of $\ell(y, z)$, and unmatch the edge (y, z). Specifically, set $\mathcal{M} \leftarrow \mathcal{M} \setminus \{(y, z)\}$, and update the relevant data structures.

Figure 2 RANDOM-SETTLE(x).

▶ **Lemma 8.** *Suppose that we call FIX-DIRTY(x) on a* DEFICIENT *node x at level $j \geq 0$. If the subroutine moves the node x to a level $k \leq j$, then it takes $O(b_x \cdot \alpha^{j+1})$ time.*

Proof (Sketch). Only the edges $(x, y) \in E$ with $\ell(y) \leq j$ get affected as the node x moves to level k. The data structure associated with every other edge remains unchanged. Thus, the total running time of the subroutine is proportional to the number of edges (x, y) with $\ell(y) \leq j$, plus the time for the call to RANDOM-SETTLE(x) or the procedure in Section 2.1.4. The latter quantity is bounded in Lemma 10 and in Section 2.1.4. As we are in Case 2, the former quantity is by definition at most $2b_x \cdot \alpha^{j+1}$. ◀

2.1.3 The subroutine Random-Settle(x)

Suppose that we call RANDOM-SETTLE(x) on a node x at level $\ell(x) = k \geq 0$. From Sections 2.1.1 and 2.1.2, we infer that $2b_x \cdot \alpha^k < |\mathcal{O}_x| \leq 2b_x \cdot \alpha^{k+1}$ at this point in time. Let $\lambda_x = |\mathcal{M}_x|$ be the number of matched edges incident on x just before the call to the subroutine. The subroutine selects $(b_x - \lambda_x)$ edges uniformly at random from $\mathcal{O}_x$ and adds them to $\mathcal{M}$. Thus, at the end of the subroutine we have $|\mathcal{M}_x| = b_x$. The following lemma crucially uses our preferential treatment to FULL-FROM-ABOVE nodes over DEFICIENT nodes.

▶ **Lemma 9.** *If a call to RANDOM-SETTLE(x) matches an edge (x, y) and unmatches an edge (y, z) (Step (04) in Figure 2), then we have $\ell(z) < \ell(x)$ during that call.*

Proof. Consider two possible cases, depending on the state of the node x.

Case 1. The node x is DEFICIENT just before the call to FIX-DIRTY(x) in Figure 1. Then at this stage there is no FULL-FROM-ABOVE node. This holds since the WHILE loop in Figure 1 always picks a FULL-FROM-ABOVE node over a DEFICIENT node. In particular, the node y is not FULL-FROM-ABOVE. Since (y, z) is a matched edge that minimises the value of $\ell(y, z)$, and since y is not FULL-FROM-ABOVE, we must have $\ell(z) \leq \ell(y, z) = \ell(y)$. Finally, note that $(x, y) \in \mathcal{O}_x$, and hence we get: $\ell(z) \leq \ell(y) < \ell(x)$.

Case 2. The node x is FULL-FROM-ABOVE just before the call to FIX-DIRTY(x) in Figure 1. Then at this stage the node x has the highest BASE(x) value among all the FULL-FROM-ABOVE nodes. Suppose that BASE(x) $= j$ at this point in time. Since the subroutine FIX-DIRTY(x) called the subroutine RANDOM-SETTLE(x), we must have been in Case 2 of Section 2.1.1. This means that the node x moves to a level strictly larger than j. Thus, we have $\ell(x) > j$ during the call to RANDOM-SETTLE(x). We now consider two cases, depending on the state of the node y. (a) If the node y is not FULL-FROM-ABOVE during the call to RANDOM-SETTLE(x), then using an argument exactly similar to the one used

in Case 1, we infer that: $\ell(z) \leq \ell(y,z) = \ell(y) < \ell(x)$. This concludes the proof. (b) Else if the node y is FULL-FROM-ABOVE during the call to RANDOM-SETTLE(x), then we claim that BASE(y) $\leq j$ at the same time-instant. This is true since the WHILE loop in Figure 1 picked the node x over y, and so we must have $j = \text{BASE}(x) \geq \text{BASE}(y)$ just before the call to FIX-DIRTY(x). Since (y,z) is an edge in $\mathcal{M}$ that minimises the value $\ell(y,z)$, we get: $\ell(z) \leq \ell(y,z) = \text{BASE}(y) \leq j < \ell(x)$. This concludes the proof. ◀

▶ **Lemma 10.** *A call to the subroutine RANDOM-SETTLE(x) takes $O(b_x \cdot \alpha^{\ell(x)+1})$ time.*

Proof (Sketch). Let $\lambda_x = |\mathcal{M}_x|$ be the number of matched edges incident on x just before the call to the RANDOM-SETTLE(x). The subroutine picks $(b_x - \lambda_x)$ edges uniformly at random from its set of owned edges $\mathcal{O}_x$. This takes $O(|\mathcal{O}_x|)$ time. Further, steps (03) – (04) in each iteration of the WHILE loop in Figure 2 takes $O(1)$ time. Thus, the total time taken by the subroutine is $O(|\mathcal{O}_x| + b_x) = O(b_x \cdot \alpha^{\ell(x)+1})$. The last equality holds since $|\mathcal{O}_x| \leq 2b_x \cdot \alpha^{\ell(x)+1}$ (see the discussion in the beginning of Section 2.1.3). ◀

2.1.4 Handling the nodes in level -1: Maintaining Invariant 2

Every node at level -1 can be in two states: *dead* or *alive*. A node x becomes alive when:

(1) x moves down to level -1 from a higher level. In this case, it scans all its incident edges at level -1 and tries to add them to $\mathcal{M}$ till $|\mathcal{M}_x|$ becomes equal to b_x. If it cannot find sufficient number of edges at level -1 to be FULL, then it remains alive, otherwise it becomes dead. The total computation cost for this step is $O(b_x)$ since x has at most $O(b_x \cdot \alpha^0)$ neighbours in level -1. This cost is charged to Lemma 8.

(2) x is dead at level -1, and it becomes DEFICIENT due to either an unmatching or a deletion of an incident edge. In case (1), x became dead only when it was FULL. Thus, at least ϵb_x edges incident on x have been either deleted from the graph or removed from $\mathcal{M}$. In the former event, we give one unit of credit to x for each edge deletion incident on it. In the latter event, we give one unit of credit to x for every unmatching of an edge incident on it at level $j \geq 0$ from the FIX-DIRTY($\cdot$) procedure that removes this edge from the matching. Thus, node x accumulates at least ϵb_x worth of credit by the time it becomes DEFICIENT. Hence, it can now scan all its incident edges at level -1, as there are $O(b_x)$ such edges.

Finally, if x is not FULL, and an edge (x,y) is added at level -1 where y is also not FULL, then we add the edge (x,y) to the b-matching. This takes $O(1)$ time. A node x at level -1 becomes dead when it is FULL. Once x becomes dead, we wait till it becomes DEFICIENT again. So there is no processing done on x as long as it remains dead. The only non-trivial processing done on x is when it becomes alive, and this takes $O(b_x)$ time. We amortise this cost over the ϵb_x many edge deletions/unmatchings that lead to the change in the state of x.

3 Bounding the Amortised Update Time of Our Algorithm

We fix a sequence of T edge insertions/deletions starting from an empty graph. We show that our algorithm takes $O(T/\epsilon^4)$ time in expectation to handle these T edge insertions/deletions. This implies an amortised update time of $O(1/\epsilon^4)$. Without any loss of generality, we assume that the graph G becomes empty at the end of this sequence of edge insertion/deletions.[1]

[1] Otherwise, we can ourselves delete the existing edges from G one after the other, and get a new sequence of at most $2T$ edge insertions/deletions. We then bound the total update time on this new sequence.

Further, to highlight the main ideas, we only focus on bounding the time spent on handling the nodes at levels $j \geq 0$. From the discussion in Section 2.1.4, it is easy to infer that the amortised time spent on the nodes at level -1 is at most $O(1/\epsilon)$ per update.

Epochs. The notion of an *epoch* will play a key role in our analysis of the amortised update time. We say that a node $x \in V$ *creates an epoch* when it matches a new edge (x, y) during a given iteration of the WHILE loop in RANDOM-SETTLE(v). Note that this is the only way an edge can be added to the b-matching at levels ≥ 0. At most b_v matched edges can be created in RANDOM-SETTLE(v). For the j-th edge selected by v, define the epoch of j-th edge, γ_j to be the contiguous time interval for which this edge remains in the matching. While creating an epoch γ_j, we select an edge (x, y) uniformly at random from $\mathcal{O}_x \setminus \mathcal{M}$ and add this edge to $\mathcal{M}$ (see Step (02) in Figure 2). This random edge (x, y) is called the *candidate edge* of epoch γ_j. We denote the *level of epoch* γ_j by $\ell(\gamma_j)$. This is same as the level of x during the call to RANDOM-SETTLE(x) in Figure 2. We say that an epoch is *destroyed* when its candidate edge is removed from the b-matching $\mathcal{M}$. This happens either when the candidate edge gets deleted from the graph, or when the candidate edge gets unmatched during the course of our algorithm. As the graph G is empty at the end of T edge insertions/deletions, every epoch gets destroyed at some point in time.

Overview of the analysis. For every node $v \in V$ and level $i \in [0, L]$, let $X_{v,i}$ be a random variable that denotes the total number of epochs created by v at level i, as we handle the fixed sequence of T edge insertions/deletions. In Lemma 11, we express the total update time as a function of these random variables $\{X_{v,i}\}$. In Lemma 12, we upper bound the expected value of this function in terms of T. The bound on the expected amortised update time of our algorithm follows from Lemmas 11 and 12 (see Corollary 13).

▶ **Lemma 11.** *It takes* $\sum_{v,i} X_{v,i} \cdot O(\alpha^{i+1}/\epsilon)$ *time to handle* T *edge insertions/deletions.*

Proof (Sketch). The time taken is dominated by the WHILE loop in Figure 1. Hence, it suffices to bound the time spent on the calls to FIX-DIRTY(v) over all $v \in V$. We will charge the total time spent in FIX-DIRTY to the start or an end of an epoch in the following way:

Scenario 1. We have $\ell(v) = i$, v is DEFICIENT, and FIX-DIRTY(v) moves the node v up to some level $j > i$ and calls RANDOM-SETTLE. By Lemma 7 the runtime of FIX-DIRTY(v) is $O(b_v \alpha^{j+1})$. When v comes to level j, it calls RANDOM-SETTLE(v) and starts at least ϵb_v new epochs at level j. So, the computation cost charged per new epoch created is $O(\alpha^{j+1}/\epsilon)$.

Scenario 2. We have $\ell(v) = i$, v is DEFICIENT, and FIX-DIRTY(v) moves v down to some level $j \leq i$. By Lemma 8 the runtime of FIX-DIRTY(v) is $O(b_v \alpha^{i+1})$. Since, v became deficient at level i, at least ϵb_v epochs involving v have been destroyed (since the last time v came to level i, FIX-DIRTY makes it FULL). Note that all such epochs have level $\geq i$. So, the computation cost charged to all epochs destroyed is $O(\alpha^{i+1}/\epsilon)$.

Scenario 3. The hard case is as follows: $\ell(v) = i$, v is FULL-FROM-ABOVE, and FIX-DIRTY(v) moves the node v up to some level $j =$ BASE(v) (or at $j >$BASE(v), Case 2, Section 2.1.1). By Lemma 6 the runtime of FIX-DIRTY(v) is $O(b_v \alpha^{j+1})$. However, it is not necessary that v starts ϵb_v epochs when it arrives at level j. So, we cannot charge this computation cost new epoch of v as no new epochs are created at level j. To overcome this problem, we crucially use the fact that v will eventually become deficient at some higher level. This holds since there are only $\log n$ levels in our partition and v cannot always be

FULL-FROM-ABOVE. Formally, define a phase ϕ of v to be the maximal time interval in which FIX-DIRTY is not called on v. A phase of v is called DEFICIENT if v becomes deficient during the phase, otherwise it is called FULL-FROM-ABOVE. Define a directed graph $\mathcal{H}_v = (\mathcal{U}_v, \mathcal{E}_v)$ where $\mathcal{U}_v$ is the set of all phases of v. For two phases $\phi_1, \phi_2 \in \mathcal{U}_v$, we have $(\phi_1, \phi_2) \in \mathcal{E}_v$ iff ϕ_1 is a FULL-FROM-ABOVE phase and ϕ_2 is the phase that begins just after ϕ_1 ends. If $l(\phi)$ denote the level of v when the phase ϕ starts, then $\ell(\phi_1) < \ell(\phi_2)$. The graph $\mathcal{H}_v$ is a collection of paths and each path ends with a DEFICIENT phase. Let $\Phi = (\phi_1, \phi_2, \ldots, \phi_k)$ be a *maximal* path in $\mathcal{H}_v$. So ϕ_k is a DEFICIENT phase and ϕ_i is a FULL-FROM-ABOVE phase for all $i < k$, and hence $\ell(\phi_1) < \cdots < \ell(\phi_k)$. Let Y_Φ be the total computation cost (due to FIX-DIRTY(v)) in phases in Φ. We get: $Y_\Phi \leq \sum_{i=1}^{k} 2b_v \cdot \alpha^{\ell(\phi_i)+1} = O(b_v \cdot \alpha^{\ell(\phi_k)+1})$. So all the computation cost can be charged to the epochs destroyed during the deficient phase of v, that is $\phi(k)$. When v move to ϕ_k, it is full (or it calls RANDOM-SETTLE(v) to become full. So at least ϵb_v epochs involving v must be destroyed for v to become deficient (All such epochs have level $\geq i$). So we can charge the computation cost Y_ϕ to these ϵb_v epochs and the cost associated with each epoch is $O(\alpha^{i+1}/\epsilon)$.

From Scenario 1,2 and 3, the computation cost charged to each epoch at level i is $O(\alpha^{i+1}/\epsilon)$. So the total time taken by our algorithm is $\sum_{v,i} X_{v,i} O(\alpha^{i+1}/\epsilon)$ ◀

▶ **Lemma 12.** *We have* $\sum_{v,i} E[X_{v,i}].O(\alpha^{i+1}/\epsilon) = O(\alpha^4 T)$.

▶ **Corollary 13.** *The expected amortised update time of our algorithm is* $O(1/\epsilon^4)$.

We now focus on justifying Lemma 12. In Section 3.1, we classify the epochs into three types. Due to space constraints, in Section 3.2 we present a (hand-wavy) intuition behind the proof of Lemma 12. A complete proof appears in the full version of the paper.

3.1 Natural, induced and forced epochs

We say that an epoch is *natural* iff it gets destroyed when its candidate edge is deleted from the graph. If an epoch is not natural, then it gets destroyed when its candidate edge is removed from $\mathcal{M}$ (but does not get deleted from G). This can happen under four possible situations: (1) In Case 2 of Section 2.1.1, when a node x moves up from level j to level $k > j$, we unmatch all its incident edges $(x, y) \in E$ whose other endpoints are at levels $\ell(y) < k$. (2) In Case 1 of Section 2.1.2, when a node x moves up from a level j to a level $k > j$, we unmatch all its incident edges $(x, y) \in E$ whose other endpoints are at levels $\ell(y) < k$. (3) During a call to the subroutine RANDOM-SETTLE(x), we unmatch an edge (y, z) because y gets matched to x. See Step (04) in Figure 1. (4) In Case 2 of Section 2.1.2, when a node x moves down from a level j, we unmatch all its incident edges (x, y) whose other endpoints are at levels $\ell(y) < j$. The epochs whose candidate edges get unmatched under situations (1), (2) and (3) are called *induced* epochs. In contrast, the epochs whose candidate edges get unmatched under situation (4) are called *forced* epochs.

3.2 Intuition behind the proof of Lemma 12: A token based argument

Let $\mathcal{V}$ be the set of all epochs. We assign $\mathcal{T}(\gamma) = \alpha^{\ell(\gamma)+1}$ tokens to every epoch $\gamma \in \mathcal{V}$. Since the computation cost charged to γ (by Lemma 11) is $O(\alpha^{l(\gamma)+1}/\epsilon)$, each token is charged with $O(1/\epsilon)$ amount of computation cost. In Lemma 14, we show that the sum $\sum_{\gamma \in \mathcal{V}} \mathcal{T}(\gamma)$ is at most $O(1/\epsilon)$ times the total number of tokens assigned to the natural epochs,

which is given by the sum $\sum_{\gamma \in \mathcal{N}} \mathcal{T}(\gamma)$. In the paragraph below, we intuitively explain that $E\left[\sum_{\gamma \in \mathcal{N}} \mathcal{T}(\gamma)\right] = O(\alpha \cdot T)$. These observations, taken together, justify Lemma 12.[2]

Suppose that each time an edge (x, y) gets deleted from G, we generate 2 *dollars* and give 1 dollar to each of the endpoints $\{x, y\}$. For the sequence of T edge insertions/deletions in G, the total number of dollars generated is $O(T)$. We will try to convince the reader that a natural epoch γ at level $\ell(\gamma) = i$ receives $\Theta(\alpha^{\ell(\gamma)})$ dollars under this scheme, in expectation. This will give us: $E\left[\sum_{\gamma \in \mathcal{N}} \mathcal{T}(\gamma)\right] = O(\alpha \cdot T)$. Accordingly, suppose that a node x at level $\ell(x) = i$ creates an epoch γ as per Step (02) in Figure 2. From the discussion in the beginning of Section 2.1.3, we get $|\mathcal{O}_x \setminus \mathcal{M}| > b_x \cdot \alpha^i$ at this point in time. We say that each edge $e \in \mathcal{O}_x \setminus \mathcal{M}$ is a *witness* for the epoch γ. Since the sequence of edge insertions/deletions in G is oblivious to the random bits used by the algorithm, in expectation half of these *witness-edges* will get deleted before the candidate edge (x, y) of the epoch γ. Thus, intuitively, if the epoch is natural, then in expectation at least $(b_x/2) \cdot \alpha^i$ of its witness edges get deleted during the lifespan of the epoch. For each of these deletions of the witness edges, the node x receives 1 dollar. Note that each such edge can be a witness to at most b_x epochs of x (since $|\mathcal{M}_x| \leq b_x$). Consider the time-instant when an edge (x, y) gets deleted from G and the node x receives 1 dollar. If we distribute this 1 dollar received by x evenly among all the epochs of x which have (x, y) as a witness, then each of those epochs will receive at least $1/b_x$ dollars. From the preceding discussion, recall that at least $(b_x/2) \cdot \alpha^i$ many witness edges get deleted from G during the lifespan of a natural epoch γ of x at level i. Hence, as promised, we conclude that such an epoch will receive at least $(1/b_x) \cdot (b_x/2) \cdot \alpha^i = \Theta(\alpha^i)$ dollars during its lifespan, in expectation.

▶ **Lemma 14.** *Let $\mathcal{V}$ be the set of all epochs, and let $\mathcal{N} \subseteq \mathcal{V}$ be the set of all natural epochs. Assign $\mathcal{T}(\gamma) = \alpha^{\ell(\gamma)+1}$ tokens to every epoch $\gamma \in \mathcal{V}$. Then $\sum_{\gamma \in \mathcal{V}} \mathcal{T}(\gamma) = O(1/\epsilon) \cdot \sum_{\gamma \in \mathcal{N}} \mathcal{T}(\gamma)$.*

We devote the rest of this section to the proof of Lemma 14. Towards this end, we construct a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. To distinguish this from the input graph $G = (V, E)$, we refer to each vertex in $\mathcal{V}$ as a *meta-node* and each edge in $\mathcal{E}$ as a *meta-edge*. The set of meta-nodes $\mathcal{V}$ corresponds to the set of all epochs. The set of meta-edges $\mathcal{E}$ is defined as follows. Each natural epoch has no incoming meta-edge. Each induced or forced epoch has exactly one incoming meta-edge. Thus, without any loss of generality, for every meta-edge $(x, y) \in \mathcal{E}$ directed from x towards y, we say that y is the parent of x and x is a child of y. Note that each meta-node has at most one parent, but it can have multiple children. We will now describe the rules that specify the parent of each induced or forced epoch.

The parent of an induced epoch is specified as follows. Note that an induced epoch is destroyed under three possible scenarios: These are situations (1), (2) and (3) as described in Section 3.1. In situation (3), we define the parent of the induced epoch (with (y, z) as the candidate edge) to be the epoch of x which lead to its destruction (i.e., the epoch with (x, y) as the candidate edge). Both in situation (1) and situation (2), a node x moves up to a higher level, unmatches some incident edges (thereby destroying some induced epochs), and adds some new incident edges to $\mathcal{M}$ (thereby creating some new epochs). As per the descriptions in Sections 2.1.1 and 2.1.2, the node x becomes FULL after these operations. Hence, the number of new epochs created during these steps is at least the number of induced epochs that get destroyed. Accordingly, for each induced epoch γ that gets destroyed, we

[2] This intuitive argument implies that the total expected running time of the algorithm is $O(\alpha \cdot T/\epsilon^2)$. A technically precise argument will lose an extra α multiplicative factor in the running time.

find a unique epoch γ' that gets created, and make γ' the parent of γ. This way of defining parents for the induced epochs ensures the following property.

▶ **Claim 15.** *If an epoch γ' is the parent of an induced epoch γ, then $\ell(\gamma') > \ell(\gamma)$. Further, an epoch γ' can have at most two children that are induced epochs.*

Proof. Let γ' be the parent of an induced epoch γ. Let x and z respectively be the nodes that created the epochs γ' and γ. If γ is destroyed during situation (3) in Section 3.1, then Lemma 9 implies that $\ell(\gamma') = \ell(x) > \ell(y, z) = \ell(\gamma)$ when γ is destroyed. Else if γ is destroyed during situation (1) or (2) in Section 3.1, then the descriptions in Sections 2.1.1, 2.1.2 lead us to the following conclusion: The node x moves up from a level l to some higher level $l' > l$. The epoch γ' is created at level l'. Further, the candidate edge of the epoch γ had both endpoints at a level strictly less than l' when γ was created. Thus, we again get $\ell(\gamma') > \ell(\gamma)$. Finally, an epoch γ' can have at most one child that is an induced epoch which gets destroyed during situation (1) or (2) in Section 3.1, and at most one child that is an induced epoch which gets destroyed during situation (3) in Section 3.1. The claim follows. ◀

We now define the parents of forced epochs. The forced epochs are destroyed during situation (4) in Section 3.1. Thus, we consider the following scenario. A node x is at level $\ell(x) = j \geq 0$, and it becomes DEFICIENT at time t_1 (say). At this stage we have $|\mathcal{M}_x| < (1 - \epsilon) \cdot b_x$. Hence, as the node x moves down, at most $(1 - \epsilon) \cdot b_x$ many forced epochs get destroyed. Let F denote the set of these forced epochs. We have $|F| \leq (1 - \epsilon) \cdot b_x$. Also note that the node x must have been FULL the last time (say t_0) it *settled* at level j after a call to FIX-DIRTY(x). Thus, for situation (4) to occur, the node x must have lost at least ϵb_x many edges in $\mathcal{M}_x$ during the time-interval $[t_0, t_1]$. Let H denote the set of these epochs that were alive at time t_0, were destroyed before time t_1, and whose candidate edges were part of $\mathcal{M}_x$. We have $|H| \geq \epsilon \cdot b_x$.

We claim that $\ell(h) > j$ for every forced epoch $h \in H$. To see why this is true, let (x, y) be the candidate edge for h. Clearly, when h was destroyed the node x stayed put at level j. It follows that since h is a forced epoch, the node y must have created h. The claim is equivalent to the statement that $\ell(y) > j$ when h gets created, and this in turn is equivalent to the statement that $\ell(y) > j$ when h gets destroyed (since $\ell(y)$ does not change during the lifespan of epoch h). To see why the latter statement is true, note that by definition y moves down to a lower level when the epoch h gets destroyed. Thus, as per Case 2 of Section 2.1.2, the node y can unmatch the edge (x, y) only if $\ell(y) > j$ at that time-instant. Hence, we get:

▶ **Property 16.** *Every forced epoch $h \in H$ has level $\ell(h) > j$.*

We now *assign* the epochs in F *evenly* among the epochs in H. To be more specific, after this step, each epoch $h \in H$ gets $|F|/|H| \leq (1 - \epsilon)/\epsilon$ many epochs $f \in F$ *assigned to it*. We are now ready to define the parents of the forced epochs F. Suppose that an epoch $f \in F$ is assigned to an epoch $h \in H$. If $\ell(h) > j$, then h becomes the parent of f. Else if $\ell(h) = j$, then from Property 16 it follows that either h is a natural epoch or h is an induced epoch. In the former event, again h becomes the parent of f. In the latter event, Claim 15 guarantees that h has a parent, say $p(h)$, and this epoch $p(h)$ becomes the parent of f. Claim 15 also implies that $\ell(p(h)) > \ell(h) = j$. Since $\ell(f) = j$, we immediately get the following claim.

▶ **Claim 17.** *Suppose that an epoch γ' is the parent of a forced epoch γ. Then either (1) $\{\ell(\gamma') > \ell(\gamma)\}$ or (2) $\{\gamma'$ is a natural epoch and $\ell(\gamma') \geq \ell(\gamma)\}$.*

Claim 18 holds since: (1) An epoch $h \in H$ gets at most $(1-\epsilon)/\epsilon$ many epochs $f \in F$ assigned to it. (2) An epoch γ can have two children $h, h' \in H$ both of which are induced epochs (see Claim 15), and both h, h' can get at most $(1 - \epsilon)/\epsilon$ many epochs $f \in F$ assigned to them.

▶ **Claim 18.** *An epoch can have at most* $3(1-\epsilon)/\epsilon$ *many forced epochs as its children.*

Proof of Lemma 14. In the meta-graph $\mathcal{G}$, every induced or forced epoch has exactly one incoming edge, and every natural epoch has zero incoming edge. Hence, the meta-graph $\mathcal{G}$ is a collection of rooted directed trees, where the tree-edges are directed away from the roots. The root of each tree is a natural epoch, and any internal node is either an induced epoch or a forced epoch. Let $\mathcal{V}'$ be the set of meta-nodes in any such tree with $r \in \mathcal{V}'$ as its root. To prove the lemma, it suffices to show that $\sum_{\gamma \in \mathcal{V}'} \mathcal{T}(\gamma) = O(1/\epsilon) \cdot \mathcal{T}(r)$.

Only the root r is a natural epoch in $\mathcal{V}'$. Hence, Claims 15, 17, 18 imply that: (1) $\ell(\gamma') > \ell(\gamma)$ whenever an *internal* meta-node $\gamma' \in \mathcal{V}'$ is the parent of $\gamma \in \mathcal{V}'$. (2) Any internal meta-node in $\mathcal{V}'$ has at most $2 + 3(1-\epsilon)/\epsilon \leq 3/\epsilon$ children. (3) The root also has at most $2 + 3(1-\epsilon)/\epsilon \leq 3/\epsilon$ children. Let C be the set of children of the root. Then for all $\gamma \in C$, we have $\ell(r) \geq \ell(\gamma)$.

Consider a meta-node $\gamma \in C$. Let $\mathcal{T}^*(\gamma)$ denote the total number of tokens assigned to the meta-nodes in the subtree rooted at γ. From the above discussions, we get: $\mathcal{T}^*(\gamma) \leq \sum_{j=0}^{\ell(\gamma)} \left(\frac{3}{\epsilon}\right)^{\ell(\gamma)-j} \cdot \alpha^{j+1} = O(\alpha^{\ell(\gamma)+1}) = O(\alpha^{\ell(r)+1})$. The first inequality holds since there are at most $(3/\epsilon)^{\ell(\gamma)-j}$ descendants of γ at level $j \leq \ell(\gamma)$, and each of these descendants get α^{j+1} tokens. The second equality holds since $\alpha = 5/\epsilon >> (3/\epsilon)$. The third equality holds since $\ell(\gamma) \leq \ell(r)$. Hence, the total number of tokens assigned to the tree is given by: $\sum_{\gamma \in \mathcal{V}'} \mathcal{T}(\gamma) = \mathcal{T}(r) + \sum_{\gamma \in C} \mathcal{T}^*(\gamma) \leq \alpha^{\ell(r)+1} + (3/\epsilon) \cdot O(\alpha^{\ell(r)+1}) = O(1/\epsilon) \cdot \mathcal{T}(r)$.

References

1. S. Baswana, M. Gupta, and S. Sen. Fully dynamic maximal matching in $O(\log n)$ update time. In *FOCS 2011*, 2011.
2. Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *ICALP 2015*, 2015.
3. Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *SODA 2016*, 2016.
4. Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Design of dynamic algorithms via primal-dual method. In *ICALP 2015*, 2015.
5. Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *SODA 2015*, 2015.
6. Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *STOC 2016*, 2016.
7. Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *STOC 2017*, 2017.
8. Manoj Gupta and Richard Peng. Fully dynamic $(1+\epsilon)$-approximate matchings. In *FOCS 2013*, 2013.
9. Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *STOC 2013*, 2013.
10. Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *STOC 2010*, 2010.
11. David Peleg and Shay Solomon. Dynamic $(1+\epsilon)$-approximate matchings: A density-sensitive approach. In *SODA 2016*, 2016.
12. Shay Solomon. Fully dynamic maximal matching in constant update time. In *FOCS 2016*, 2016.

Fast Dynamic Arrays

Philip Bille[1], Anders Roy Christiansen[2], Mikko Berggren Ettienne[3], and Inge Li Gørtz[4]

1 **The Technical University of Denmark, Lyngby, Denmark**
`phbi@dtu.dk`
2 **The Technical University of Denmark, Lyngby, Denmark**
`aroy@dtu.dk`
3 **The Technical University of Denmark, Lyngby, Denmark**
`miet@dtu.dk`
4 **The Technical University of Denmark, Lyngby, Denmark**
`inge@dtu.dk`

Abstract

We present a highly optimized implementation of tiered vectors, a data structure for maintaining a sequence of n elements supporting access in time $O(1)$ and insertion and deletion in time $O(n^\epsilon)$ for $\epsilon > 0$ while using $o(n)$ extra space. We consider several different implementation optimizations in C++ and compare their performance to that of *vector* and *set* from the standard library on sequences with up to 10^8 elements. Our fastest implementation uses much less space than set while providing speedups of 40× for access operations compared to set and speedups of 10.000× compared to vector for insertion and deletion operations while being competitive with both data structures for all other operations.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, E.1 Data Structures

Keywords and phrases Dynamic Arrays, Tiered Vectors

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.16

1 Introduction

We present a highly optimized implementation of a data structure solving the *dynamic array problem*, that is, maintain a sequence of elements subject to the following operations:

access(i)**:** return the i^{th} element in the sequence.
access(i, m)**:** return the i^{th} through $(i + m - 1)^{th}$ elements in the sequence.
insert(i, x)**:** insert element x immediately after the i^{th} element.
delete(i)**:** remove the i^{th} element from the sequence.
update(i, x)**:** exchange the i^{th} element with x.

This is a fundamental and well studied data structure problem [2, 4, 7, 8, 3, 1, 5, 6] solved by textbook data structures like arrays and binary trees. Many dynamic trees provide all the operations in $O(\lg n)$ time including 2-3-4 trees, AVL trees, splay trees, etc. while Dietz [2] gives a data structure that matches the lower bound of $\Omega(\lg n / \lg \lg n)$ showed by Fredman and Saks [4]. In this paper however, we focus on the problem where access must run in $O(1)$ time. Goodrich and Kloss present what they call *tiered vectors* [5] with a time complexity of $O(1)$ for access and update and $O(n^{1/l})$ for insert and delete for any constant integer $l \geq 2$, similar to the ideas presented by Frederickson in [3]. The data structure only uses $o(n)$ extra space beyond that required to store the actual elements. At the core, the data structure is a tree with out degree $n^{1/l}$ and *constant* height $l - 1$.

Goodrich and Kloss compare the performance of an implementation with $l = 2$ to that of *vector* from the standard library of Java and show that the structure is competitive for access operations while being significantly faster for insertions and deletions. Tiered vectors provide a performance trade-off between standard arrays and balanced binary trees for the dynamic array problem.

Our Contribution. In this paper, we present what we believe is the first implementation of tiered vectors that supports more than 2 tiers. Our C++ implementation supports access and update in times that are competitive with the vector class from C++'s standard library while insert and delete run more than 10.000× faster. It performs access and update more than 40× faster than the set class from the standard library while insert and delete is only a few percent slower. Furthermore set uses more than 10× more space than our implementation. All of this when working on large sequences of 10^8 32-bit integers.

To obtain these results, we significantly decrease the number of memory probes of the original tiered vector. Our best variant requires only half as many memory probes as the original tiered vector for access and update operations which is critical for the practical performance. Our implementation is cache efficient which makes all operations run fast in practice even on tiered vectors with several tiers.

We experimentally compare the different variants of tiered vectors. Besides the comparison to the two commonly used C++ data structures, vector and set, we compare the different variants of tiered vectors to find the best one. We show that the number of tiers have a significant impact on the performance which underlines the importance of tiered vectors supporting more than 2 tiers.

Our implementations are parameterized and thus support any number of tiers ≥ 2. It uses a number of tricks like *template recursion* to keep the code rather simple while enabling the compiler to generate highly optimized code.

2 Preliminaries

The first and i^{th} element of a sequence A are denoted $A[0]$ and $A[i-1]$ respectively and the i^{th} through j^{th} elements are denoted $A[i-1, j-1]$. Let $A_1 \cdot A_2$ denote the concatenation of the sequences A_1 and A_2. $|A|$ denotes the number of elements in the sequence A. A circular shift of a sequence A by x is the sequence $A[|A| - x, |A| - 1] \cdot A[0, |A| - x - 1]$. Define the remainder of division of a by b as $a \mod b = a - qb$ where q is the largest integer such that $q \cdot b \leq a$. Define $A[i, j] \mod w$ to be the elements $A[i \mod w], A[(i+1) \mod w], \ldots, A[j \mod w]$, i.e. $A[4, 7] \mod 5 = A[4], A[0], A[1], A[2]$. Let $\lfloor x \rfloor$ denote the largest integer smaller than x.

3 Tiered Vectors

In this section we will describe how the tiered vector data structure from [5] works.

Data Structure. An l-tiered vector can be seen as a tree T with root r, fixed height $l - 1$ and out-degree w for any $l \geq 2$. A node $v \in T$ represents a sequence of elements $A(v)$ where $A(r)$ is the sequence represented by the tiered vector. The capacity $\text{cap}(v)$ of a node v is $w^{\text{height}(v)+1}$. For a node v with children $c_1, c_2, \ldots, c_w$, $A(v)$ is a circular shift of the concatenation of the elements represented by its children, $A(c_1) \cdot A(c_2) \cdot \ldots \cdot A(c_w)$. The circular shift is determined by an integer $\text{off}(v) \in [\text{cap}(v)]$ that is explicitly stored for all

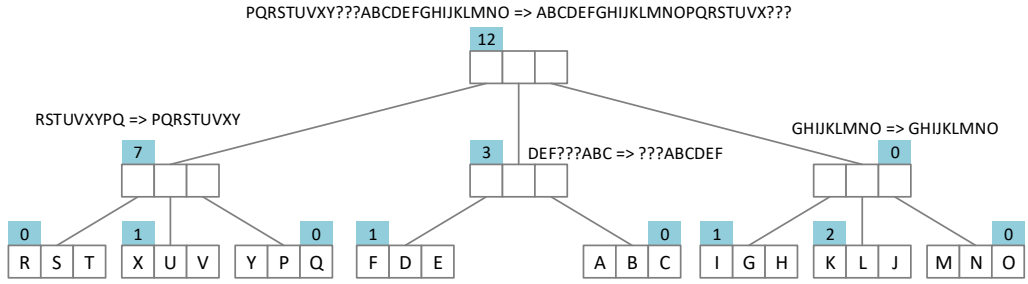


Figure 1 An illustration of a tiered vector with $l = w = 3$. The elements are letters, and the tiered vector represents the sequence ABCDEFGHIJKLMNOPQRSTUVX. The elements in the leaves are the elements that are actually stored. The number above each node is its offset. The strings above an internal node v with children c_1, c_2, c_3 is respectively $A(c_1) \cdot A(c_2) \cdot A(c_3)$ and $A(v)$, i.e. the elements v represents before and after the circular shift. ? specifies an empty element.

nodes. Thus the sequence of elements $A(v)$ of an internal node v can be reconstructed by recursively reconstructing the sequence for each of its children, concatenating these and then circular shifting the sequence by $\text{off}(v)$. See Figure 1 for an illustration. A leaf v of T explicitly stores the sequence $A(v)$ in a circular array $\text{elems}(v)$ with size w whereas internal nodes only store their offsets. Call a node v full if $|A(v)| = \text{cap}(v)$ and empty if $|A(v)| = 0$. In order to support fast access, for all nodes v the elements of $A(v)$ are located in consecutive children of v that are all full, except the children containing the first and last element of $A(v)$ which may be only partly full.

Access & Update. To access an element $A(r)[i]$ at a given index i; one traverses a path from the root down to a leaf in the tree. In each node the offset of the node is added to the index to compensate for the cyclic shift, and the traversing is continued in the child corresponding to the newly calculated index. Finally the desired element is returned from the elements array of that leaf. Let $\text{access}(v, i)$ return the element $A(v)[i]$, it can recursively be computed as:

- **v is internal:** Compute $i' = (i + \text{off}(v)) \mod \text{cap}(v)$, let v' be the $\lfloor i'/w \rfloor^{th}$ child of v and return the element $\text{access}(v', i' \mod \text{cap}(v'))$.
- **v is leaf:** Compute $i' = (i + \text{off}(v)) \mod w$ and return the element $\text{elems}(v)[i']$.

The time complexity is $\Theta(l)$ as we visit all nodes on a root-to-leaf path in T. To navigate this path we must follow $l - 1$ child pointers, lookup l offsets, and access the element itself. Therefore this requires $l - 1 + l + 1 = 2l$ memory probes.

The update operation is entirely similar to access, except the element found is not returned but substituted with the new element. The running time is therefore $\Theta(l)$ as well. For further use, let $\text{update}(v, i, e)$ be the operation that sets $A(v)[i] = e$ and returns the element that was substituted.

Range Access. Accessing a range of elements, can obviously be done by using the access-operation multiple times, but this results in redundant traversing of the tree, since consecutive elements of a leaf often – but not always due to circular shifts – corresponds to consecutive elements of $A(r)$. Let $\text{access}(v, i, m)$ report the elements $A(v)[i \dots i + m - 1]$ in order. The operation can recursively be defined as:

v is internal: Let $i_l = (i + \text{off}(v)) \mod \text{cap}(v)$, and let $i_r = (i_l + m) \mod \text{cap}(v)$. The children of v that contains the elements to be reported are in the range $[\lfloor i_l \cdot w/\text{cap}(v) \rfloor, \lfloor i_r \cdot w/\text{cap}(v) \rfloor] \mod w$, call these $c_l, c_{l+1}, \ldots, c_r$. In order, call $\mathsf{access}(c_l, i_l, \min(m, \text{cap}(c_l) - i_l))$, $\mathsf{access}(c_i, 0, \text{cap}(c_i))$ for $c_i = c_{l+1}, \ldots, c_{r-1}$, and $\mathsf{access}(c_r, e_{r-1}, 0, i_r \mod \text{cap}(c_r))$.

v is leaf: Report the elements $\text{elems}(v)[i, i + m - 1] \mod w$.

The running time of this strategy is $O(lm)$, but saves a constant factor over the naive solution.

Insert & Delete. Inserting an element in the end (or beginning) of the array can simply be achieved using the update-operation. Thus the interesting part is fast insertion on an arbitrary position; this is where we utilize the offsets.

Consider a node v, the key challenge is to shift a big chunk of elements $A(v)[i, i + m - 1]$ one index right (or left) to $A(v)[i+1, i+m]$ to make room for a new element (without actually moving each element in the range). Look at the range of children $c_l, c_{l+1}, \ldots, c_r$ that covers the range of elements $A(v)[i, i + m - 1]$ to be shifted. All elements in $c_{l+1}, \ldots, c_{r-1}$ must be shifted. These children are guaranteed to be full, so make a circular shift by decrementing each of their offsets by one. Afterwards take the element $A(c_{i-1})[0]$ and move it to $A(c_i)[0]$ using the update operation for $l < i \leq r$. In c_l and c_r only a subrange of the elements might need shifting, which we do recursively. In the base case of this recursion, namely when v is a leaf, shift the elements by actually moving the elements one-by-one in $\text{elems}(v)$.

Formally we define the $\mathsf{shift}(v, e, i, m)$ operation that (logically) shifts all elements $A(v)[i, i + m - 1]$ one place right to $A[i + 1, i + m]$, sets $A[i] = e$ and returns the value that was previously on position $A[i + m]$ as:

v is internal: Let $i_l = (i + \text{off}(v)) \mod \text{cap}(v)$, and let $i_r = (i_l + m) \mod \text{cap}(v)$. The children of v that must be updated are in the range $[\lfloor i_l \cdot w/\text{cap}(v) \rfloor, \lfloor i_r \cdot w/\text{cap}(v) \rfloor] \mod w$ call these $c_l, c_{l+1}, \ldots, c_r$. Let $e_l = \mathsf{shift}(c_l, e, i_l, \min(m, \text{cap}(c_l) - i_l))$. Let $e_i = \mathsf{update}(c_i, size(c)-1, e_{i-1})$ and set $\text{off}(c_i) = (\text{off}(c_i)-1) \mod \text{cap}(c)$ for $c_i = c_{l+1}, \ldots, c_{r-1}$. Finally call $\mathsf{shift}(c_r, e_{r-1}, 0, i_r \mod \text{cap}(c_r))$.

v is leaf: Let $e_o = \text{elems}(v)[(i + m) \mod w]$. Move the elements $\text{elems}(v)[i, (i + m - 1) \mod w]$ to $\text{elems}(v)[i + 1, (i + m) \mod w]$, and set $\text{elems}(v)[i] = e$. Return e_o.

An insertion $\mathsf{insert}(i, e)$ can then be performed as $\mathsf{shift}(root, e, i, size(root) - i - 1)$. The running time of an insertion is $T(l) = 2T(l - 1) + w \cdot l \Rightarrow T(l) = O(2^l w)$.

A deletion of an element can basically be done as an inverted insertion, thus deletion can be implemented using the shift-operation from before. A $\mathsf{delete}(i)$ can be performed as $\mathsf{shift}(r, \perp, 0, i)$ followed an update of the root's offset to $(\text{off}(r) + 1) \mod \text{cap}(r)$.

Space. There are at most $O(w^{l-1})$ nodes in the tree and each takes up constant space, thus the total space of the tree is $O(w^{l-1})$. All leaves are either empty or full except the two leaves storing the first and last element of the sequence which might contain less than w elements. Because the arrays of empty leaves are not allocated the space overhead of the arrays is $O(w)$. Thus beyond the space required to store the n elements themselves, tiered vectors have a space overhead of $O(w^{l-1})$.

To obtain the desired bounds w is maintained such that $w = \Theta(n^\epsilon)$ where $\epsilon = 1/l$ and n is the number of elements in the tiered vector. This can be achieved by using global rebuilding to gradually increase/decrease the value of w when elements are inserted/deleted without asymptotically changing the running times. We will not provide the details here. We sum up the original tiered vector data structure in the following theorem:

▶ **Theorem 1** ([5]). *The original l-tiered vector solves the dynamic array problem for $l \geq 2$ using $\Theta(n^{1-1/l})$ extra space while supporting* access *and* update *in $\Theta(l)$ time and $2l$ memory probes. The operations* insert *and* delete *take $O(2^l n^{1/l})$ time.*

4 Improved Tiered Vectors

In this paper, we consider different new variants of the tiered vector. This section considers the theoretical properties of these approaches. In particular we are interested in the number of memory accesses that are required for the different memory layouts, since this turns out to have an effect on the experimental running time. In Section 5.1 we analyze the actual impact in practice through experiments.

4.1 Implicit Tiered Vectors

As the degree of all nodes is always fixed to the same value w (it may be changed for all nodes when the tree is rebuilt due to a full root), it is possible to layout the offsets and elements such that no pointers are necessary to navigate the tree. Simply number all nodes from left-to-right level-by-level starting in the root with number 0. Using this numbering scheme, we can store all offsets of the nodes in a single array and similarly all the elements of the leaves in another array.

To access an element, we only have to lookup the offset for each node on the root-to-leaf path which requires $l - 1$ memory probes plus the final element lookup, i.e. in total l which is half as many as the original tiered vector. The downside with this representation is that it must allocate the two arrays in their entirety from the beginning (or when rebuilding). This results in a $\Theta(n)$ space overhead which is worse than the $\Theta(n^{1-\epsilon})$ space overhead from the original tiered vector.

▶ **Theorem 2.** *The implicit l-tiered vector solves the dynamic array problem for $l \geq 2$ using $O(n)$ extra space while supporting* access *and* update *in $O(l)$ time requiring l memory probes. The operations* insert *and* delete *take $O(2^l n^{1/l})$ time.*

4.2 Lazy Tiered Vectors

We now combine the original and the implicit representation, to get both few memory probes and small space overhead. Instead of having one array storing all the elements of the leaves, we store for each leaf a pointer to a location with an array containing the leaf's elements. The array is lazily allocated in memory when elements are actually inserted into it.

The total size of the offset-array and the element pointers in the leaves is $O(n^{1-\epsilon})$. At most two leaves are only partially full, therefore the total space is now again reduced to $O(n^{1-\epsilon})$. To navigate a root-to-leaf path, we now need to look at $l - 1$ offsets, follow a pointer from a leaf to its array and access the element in the array, giving a total of $l + 1$ memory accesses.

▶ **Theorem 3.** *The lazy l-tiered vector solves the dynamic array problem for $l \geq 2$ using $\Theta(n^{1-1/l})$ extra space while supporting* access *and* update *in $\Theta(l)$ time requiring $l+1$ memory probes. The operations* insert *and* delete *take $O(2^l n^{1/l})$ time.*

5 Implementation

We have implemented a generic version of the tiered vector data structure such that the number of tiers and the size of each tier can be specified at compile time. To the best of our knowledge,

all prior implementations of the tiered vector are limited to the considerably simpler 2-tier version. Most of the performance optimizations applied in the 2-tier implementation do not easily generalize. We have implemented the following variants of tiered vectors:

- *Original.* The data structure described in Theorem 1.
- *Optimized Original.* As described in Theorem 1 but with the offset of a node v located in the parent of v, adjacent in memory to the pointer to v. Leaves only consists of an array of elements (since their parent store their offset) and the root's offset is maintained separately as there is no parent to store it in.
- *Implicit.* This is the data structure described in Theorem 2 where the tree is represented implicitly in an array storing the offsets and the elements of the leaves are located in a single array.
- *Packed Implicit.* This is the data structure described in Theorem 2 with the following optimization; The offsets stored in the offset array are are packed together and stored in as little space possible. The maximum offset of a node v in the tree is $n^{\epsilon(\text{height}(v)+1)}$ and the number of bits needed to store all the offsets is therefore $\sum_{i=0}^{l} n^{1-i\epsilon} \log(n^{i\epsilon}) = \log(n) \sum_{i=0}^{l} i\epsilon n^{1-i\epsilon} \approx \epsilon n^{1-\epsilon} \log(n)$. Thus the $n^{1-\epsilon}$ offsets can be stored in approximately $\epsilon n^{1-\epsilon}$ words giving a space reduction of a constant factor ϵ. The smaller memory footprint could lead to better cache performance.
- *Lazy.* This is the data structure described in Theorem 3 where the tree is represented implicitly in an array storing the offsets and every leaf store a pointer to an array storing only the elements of that leaf.
- *Packed Lazy.* This is the data structure described in Theorem 3 with the following optimization; The offset and the pointer stored in a leaf is packed together and stored at the same memory location. On most modern 64-bit system – including the one we are testing on – a memory pointer is only allowed to address 48 bits. This means we have room to pack a 16 bit offset in the same memory location as the elements pointer, which results in one less memory probe during an access operation.
- *Non-Templated.* All other implementations used C++ templating for recursive functions in order to let the compiler do significant code optimizations. This implementation is template free and serves as a baseline to compare the performance gains given by templating.

In Section 7 we compare the performance of these implementations.

5.1 C++ Templates

As almost all other general purpose data structures in C++, we have used templates to support storing different types of data in our tiered vector. This is a well-known technique which we will not describe in detail.

However, we have also used *template recursion* which is basically like a normal recursion except that the recursion parameter must be a compile-time constant. This allows the compiler to unfold the recursion at compile-time eliminating all (recursive) function calls by inlining code, and allowing for better local code optimizations. In our case, we exploit that the height of a tiered vector is constant and therefore can be used for this.

To show the rather simple code resulting from this approach (disregarding the template stuff itself), we have included a snippet of the internals of our access operation:

```
template <class T, class Layer>
struct helper {
        static T& get(size_t node, size_t idx) {
            idx = (idx + get_offset(node)) % Layer::capacity;
            auto child = get_child(node, idx / Layer::child::capacity);
            return helper<T, typename Layer::child>::get(child, idx);
        }
}

template <class T, size_t W>
struct helper<T, Layer<W, LayerEnd> > {
        static T& get(size_t node, size_t idx) {
            idx = (idx + get_offset(node)) % L::capacity;
            return get_elem(node, idx);
        }
}
```

We also briefly show how to use the data structure. To specify the desired height of the tree, and the width of the nodes on each tier, we also use templating:

```
Tiered<int, Layer<8, Layer<16, Layer<32>>>> tiered;
```

This will define a tiered vector containing integers with three tiers. The height of the underlying tree is therefore 3 where the root has 8 children, each of which has 16 children each of which contains 32 elements. We call this configuration 8-16-32.

In this implementation of tiered vectors we have decided to let the number of children on each level be a fixed number as described above. This imposes a maximum on the number of elements that can be inserted. However, in a production ready implementation, it would be simple to make it grow-able by maintaining a single growth factor that should be multiplied on the number of children on each level. This can be combined with the templated solution since the growing is only on the number of children and not the height of the tree (per definition of tiered vectors the height is constant). This will obviously increase the running time for operations when growing/shrinking is required, but will only have minimal impact on all other operations (they will be slightly slower because computations now must take the growth factor into account).

In practice one could also, for many uses, simply pick the number of children on each level sufficiently large to ensure the number of elements that will be inserted is less than the maximum capacity. This would result in a memory overhead when the tiered vector is almost empty, but by choosing the right variant of tiered vectors and the right parameters this overhead would in many cases be insignificant.

6 Comparison with C++ STL Data Structures

In the following we have compared our best performing tiered vector (see next section) to the vector and the multiset class from the C++ standard library. The vector class directly supports the operations of a dynamic array. The multiset class is implemented as a red-black tree and is therefore interesting to compare with our data structure. Unfortunately, multiset does not directly support the operations of a dynamic array (in particular it has no notion of positions of elements). To simulate an access operation we instead find the successor of an element in the multiset. This requires a root-to-leaf traversal of the red-black tree, just as

an access operation in a dynamic array implemented as a red-black tree would. Insertion is simulated as an insertion into the multiset, which again requires the same computations as a dynamic array implemented as a red-black tree would.

Besides the random access, range access and insertion tests considered in the previous sections, we have also tested the operations *data dependent access*, insertion in the end, deletion, and *successor*. In the *data dependent access* tests, the next index to lookup depends on the value at the prior lookups. This ensures that the processor cannot successfully pipeline consecutive lookups, but must perform them in sequence. We test insertion in the end, since this is a very common use case. Deletion is performed by deleting elements at random positions. The *successor* operation returns the successor of an element and is not actually part of the dynamic array problem, but is included since it is a commonly used operation on a set in C++. It is simply implemented as a binary search over the elements in both the vector and tiered vector tests where the elements are now inserted in sorted order. The number of tests and operations is the same as in the other tests.

The results are summarized in Table 1 which shows that the vector performs slightly better than the tiered vector on all access and successor tests. As expected from the $\Theta(n)$ running time, it performs extremely poor on random insertion and deletion. For insertion in the end of the sequence, vector is also slightly faster than the tiered vector. The interesting part is that even though the tiered vector requires several extra memory lookups and computations, we have managed to get the running time down to less than the double of the vector for access, even less for data dependent and only a few percent slowdown for range access. As discussed earlier, this is most likely because the entire tree structure (without the elements) fits within the CPU cache, and because the computations required has been minimized.

Comparing our tiered vector to set, we would expect access operations to be faster since they run in $O(1)$ time compared to $O(\log n)$. On the other hand, we would expect insertion/deletion to be significantly slower since it runs in $O(n^{1/l})$ time compared to $O(\log n)$ (where $l = 4$ in these tests). We see our expectations hold for the access operations where the tiered vector faster by more than an order of magnitude. In random insertions however, the tiered vector is only 8% slower – even when operating on 100.000.000 elements. Both the tiered vector and set requires $O(\log n)$ time for the successor operation. In our experiment the tiered vector is 3 times faster for the successor operation.

Finally, we see the memory usage of vector and tiered vector is almost identical. This is expected since in both cases it is primarily the elements themselves that take up space. The set uses more than 10 times as much space, so this is also a considerable drawback of the red-black tree behind this structure.

To sum up, the tiered vectors performs better on all tests but insertion, but is even here highly competitive.

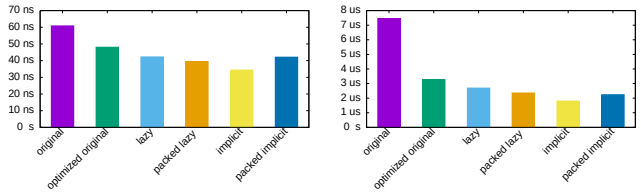
7 Tiered Vector Experiments

In this section we compare different variants of the tiered vector. We first consider how the performance of the different representations of the data structure listed in Section 5, and also how the height of tree and the capacity of the leaves affects the running time. Afterwards we compare it to some widely used C++ standard library containers.

Environment. All experiments have been performed on a Intel Core i7-4770 CPU @ 3.40GHz with 32 GB RAM. The code has been compiled with GNU GCC version 5.4.0 with flags "-O3". The reported times are an average over 10 test runs.

Table 1 The table summarizes the performance of the implicit tiered vector compared to the performance of set and vector from the C++ standard library. dd-access refers to data dependent access.

	tiered vector	*set*	*set / tiered*	*vector*	*vector / tiered*
access	34.07 ns	1432.05 ns	42.03	21.63 ns	0.63
dd-access	99.09 ns	1436.67 ns	14.50	79.37 ns	0.80
range access	0.24 ns	13.02 ns	53.53	0.23 ns	0.93
insert	1.79 μs	1.65 μs	0.92	21675.49 μs	12082.33
insertion in end	7.28 ns	242.90 ns	33.38	2.93 ns	0.40
successor	0.55 μs	1.53 μs	2.75	0.36 μs	0.65
delete	1.92 μs	1.78 μs	0.93	21295.25 μs	11070.04
memory	408 MB	4802 MB	11.77	405 MB	0.99



(a) *access* **(b)** *insert*

Figure 2 Figures (a) and (b) show the performance of the *original* (—), *optimized original* (—), *lazy* (—) *packed lazy* (—), *implicit* (—) and *packed implicit* (—) layouts.

Procedure. In all tests 10^8 32-bit integers are inserted in the data structure as a preliminary step to simulate that it has already been used[1]. For all the access and successor operations 10^9 elements have been accessed and the time reported is the average time per element. For range access, blocks of 10.000 elements have been used. For insertion/deletion 10^6 elements have been (semi-)randomly[2] added/deleted, though in the case of "vector" only 10.000 elements were inserted/deleted to make the experiments run within reasonable time.

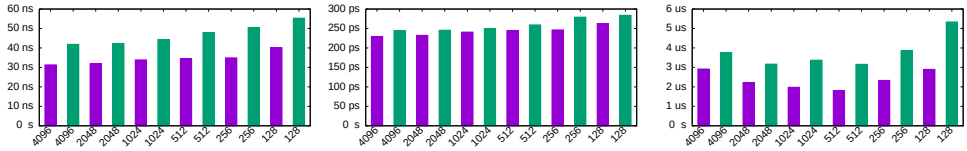
7.1 Tiered Vector Variants Experiments

In this test we compare the performance of the implementations listed in Section 5 to that or the original data structure as described in Theorem 1.

Optimized Original. By co-locating the child offset and child pointer, the two memory lookups are at adjacent memory locations. Due to the cache lines in modern processors, this means the second memory lookup will often be answered directly by the fast L1-cache. As can be seen on Figure 2, this small change in the memory layout results in a significant improvement in performance for both access and insertion. In the latter case, the running time is more than halved.

[1] In order to minimize the overall running time of the experiments, the elements were not added randomly, but we show this does not give our data structure any benefits

[2] In order to not impact timing, a simple access pattern has been used instead of a normal pseudo-random generator.



(a) *access* **(b)** *range access* **(c)** *insert*

Figure 3 Figures (a), (b) and (c) show the performance of the *implicit* (—) and the *optimized original* tiered vector (—) for different tree widths.

Lazy and Packed Lazy. Figure 2 shows how the fewer memory probes required by the *lazy* implementation in comparison to the *original* and *optimized original* results in better performance. Packing the offset and pointer in the leaves results in even better performance for both access and insertion even though it requires a few extra instructions to do the actual packing and unpacking.

Implicit. From Figure 2, we see the implicit data structure is the fastest. This is as expected because it requires fewer memory accesses than the other structures except for the packed lazy which instead has slight computational overhead due to the packing and unpacking.

As shown in Theorem 2 the implicit data structure has a bigger memory overhead than the lazy data structure. Therefore the packed lazy representation might be beneficial in some settings.

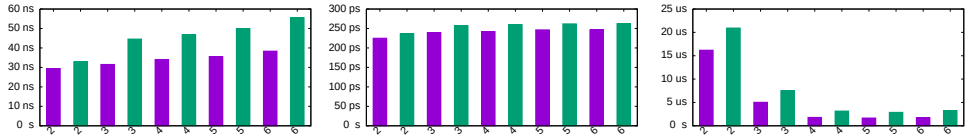
Packed Implicit. Packing the offsets array could lead to better cache performance due to the smaller memory footprint and therefore yield better overall performance. As can be seen on Figure 2, the smaller memory footprint did not improve the performance in practice. The simple reason for this, is that the strategy we used for packing the offsets required extra computation. This clearly dominated the possible gain from the hypothesized better cache performance. We tried a few strategies to minimize the extra computations needed at the expense of slightly worse memory usage, but none of these led to better results than when not packing the offsets at all.

7.2 Width Experiments

This experiment was performed to determine the best capacity ratio between the leaf nodes and the internal nodes. The six different width configurations we have tested are: 32-32-32-4096, 32-32-64-2048, 32-64-64-1024, 64-64-64-512, 64-64-128-256, and 64-128-128-128. All configurations have a constant height 4 and a capacity of approximately 130 mio.

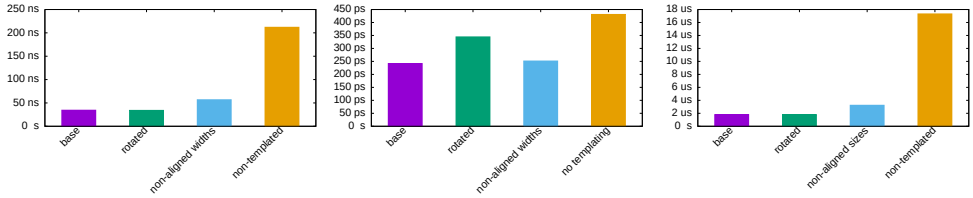
We expected the performance of access operations to remain unchanged, since the number of operations it must perform only depends on the height of the tree, and not the widths. We expect range access to perform better when the leaf size is increased, since more elements will be located in consecutive memory locations. For *insertion* there is not a clearly expected behavior as the time used to physically move elements in a leaf will increase with leaf size, but then less operations on the internal nodes of the tree has to be performed.

On Figure 3 we see access times are actually decreasing slightly when leaves get bigger. This is a bit unexpected, but is most likely due to small changes in the memory layout that results in slightly better cache performance. The same is the case for range access, but this



(a) *access(i)* **(b)** *access(i, m)* **(c)** *insert*

Figure 4 Figures (a),(b) and (c) show the performance of the *implicit* (—) and the *optimized original* tiered vector (—) for different tree heights.



(a) *access* **(b)** *range access* **(c)** *insert(i,x)*

Figure 5 Figures (a) and (b) show the performance of the *base* (—), *rotated* (—), *non-aligned sizes* (—), *non-templated* (—) layouts.

was expected. For insertion, we see there is a tipping point. For our particular instance, the best performance is achieved when the leaves have size around 512.

Based on this, we have performed the remaining tests with the 64-64-64-512 configuration (unless otherwise specified).

7.3 Height Experiments

In these tests we have studied how different heights affect the performance of access and insertion operations. We have tested the configurations 8196-16384, 512-512-512, 64-64-64-512, 16-16-32-32-512, 8-8-16-16-16-512. All resulting in the same capacity, but with heights in the range 2-6.

We expect the access operations to performance better for lower trees, since the number of operations that must be performed is linear in the height. On the other hand we expect insertion to perform significantly better with higher trees, since its running time is $O(n^{1/l})$ where l is one the height plus one.

On Figure 4 we see the results follow our expectations. However, the access operations only perform slightly worse on higher trees. We expect this to be because all internal nodes fit within the L3-cache. Therefore the dominant running time comes from the lookup of the element itself. (It is highly unlikely that the element requested by an access to a random position would be among the small fraction of elements that fit in the L3-cache).

Regarding insertion, we see significant improvements up until a height of 4 after that, increasing the height does not change the running time noticeably. This is most likely due to the hidden constant in $O(n^{1/l})$ increases rapidly with the height.

7.4 Configuration Experiments

In these experiments, we test a few hypotheses about how different changes impact the running time. The results are shown on Figure 5, the leftmost result (base) is our final and best implementation to which we compare our hypotheses.

Rotated: As already mentioned, the insertions performed as a preliminary step to the tests are not done at random positions. This means that all offsets are zero when our real operations start. The purpose of this test is the ensure that there are no significant performance gains in starting from such a configuration which could otherwise lead to misleading results. To this end, we have randomized all offsets (in a way such that the data structure is still valid, but the order of elements change) after doing the preliminary insertions but before timing the operations. As can be seen on Figure 5, the difference between this and the normal procedure is insignificant, thus we find our approach gives a fair picture.

Non-Aligned Sizes: In all our previous tests, we have ensured all nodes had an out-degree that was a power of 2. This was chosen in order to let the compiler simplify some calculations, i.e. replacing multiplication/division instructions by shift/and instructions. As Figure 5 shows, using sizes that are not powers of 2 results in significantly worse performance. Besides from showing that one should always pick powers of 2, it also indicates that not only the number of memory accesses during an operation is critical for our performance, but also the amount of computation we make.

Non-Templated: The non-templated results in Figure 2 the show that the change to templated recursion has had a major impact on the running time. It should be noted that some improvements have not been implemented in the non-templated version, but it gives a good indication that this has been quite useful.

8 Conclusion

This paper presents the first implementation of a generic tiered vector supporting any constant number of tiers. We have shown a number of modified version of the tiered vector, and employed several speed optimizations to the implementation. These implementations have been compared to vector and multiset from the C++ standard library. The benchmarks show that our implementation stays on par with vector for access and on update operations while providing a considerable speedup of more than 40× compared to set. At the same time the asymptotic difference between the logarithmic complexity of multiset and the polynomial complexity of tiered vector for insertion and deletion operations only has little effect in practice. For these operations, our fastest version of the tiered vector suffers less than 10% slowdown. Arguably, our tiered array provides a better trade-off than the balanced binary tree data structures used in the standard library for most applications that involves big instances of the dynamic array problem.

References

1 Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, WADS'99, pages 37–48, London, UK, UK, 1999. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=645932.673194`.

2 Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proceedings of the Workshop on Algorithms and Data Structures*, WADS'89, pages 39–46, London, UK, UK, 1989. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=645928.672529`.

3 Greg N. Frederickson. Implicit data structures for the dictionary problem. *J. ACM*, 30(1):80–94, January 1983. `doi:10.1145/322358.322364`.

4 M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC'89, pages 345–354, New York, NY, USA, 1989. ACM. doi:10.1145/73007.73040.

5 Michael T. Goodrich and John G. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In Frank Dehne, Jörg-Rüdiger Sack, Arvind Gupta, and Roberto Tamassia, editors, *Algorithms and Data Structures: 6th International Workshop, WADS'99 Vancouver, Canada, August 11–14, 1999 Proceedings*, pages 205–216, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-48447-7_21.

6 Jyrki Katajainen. Worst-case-efficient dynamic arrays in practice. In Andrew V. Goldberg and Alexander S. Kulikov, editors, *Experimental Algorithms: 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, pages 167–183, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-38851-9_12.

7 Jyrki Katajainen and Bjarke Buur Mortensen. Experiences with the design and implementation of space-efficient deques. In Gerth Stølting Brodal, Daniele Frigioni, and Alberto Marchetti-Spaccamela, editors, *Algorithm Engineering: 5th International Workshop, WAE 2001 Århus, Denmark, August 28–31, 2001 Proceedings*, pages 39–50, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-44688-5_4.

8 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In Frank Dehne, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Algorithms and Data Structures: 7th International Workshop, WADS 2001 Providence, RI, USA, August 8–10, 2001 Proceedings*, pages 426–437, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-44634-6_39.

On the Impact of Singleton Strategies in Congestion Games

Vittorio Bilò[1] **and Cosimo Vinci**[2]

1 **Department of Mathematics and Physics, University of Salento, Lecce, Italy**
vittorio.bilo@unisalento.it
2 **Gran Sasso Science Institute, L'Aquila, Italy**
cosimo.vinci@gssi.it

Abstract

To what extent does the structure of the players' strategy space influence the efficiency of decentralized solutions in congestion games? In this work, we investigate whether better performance is possible when restricting to load balancing games in which players can only choose among single resources. We consider three different solutions concepts, namely, approximate pure Nash equilibria, approximate one-round walks generated by selfish players aiming at minimizing their personal cost and approximate one-round walks generated by cooperative players aiming at minimizing the marginal increase in the sum of the players' personal costs. The last two concepts can also be interpreted as solutions of simple greedy online algorithms for the related resource selection problem. Under fairly general latency functions on the resources, we show that, for all three types of solutions, better bounds cannot be achieved if players are either weighted or asymmetric. On the positive side, we prove that, under mild assumptions on the latency functions, improvements on the performance of approximate pure Nash equilibria are possible for load balancing games with weighted and symmetric players in the case of identical resources. We also design lower bounds on the performance of one-round walks in load balancing games with unweighted players and identical resources (in this case, solutions generated by selfish and cooperative players coincide).

1998 ACM Subject Classification F.7.2 Algorithmic Game Theory and Mechanism Design, F.7.2.6 Quality of Equilibria, F.5.5 Online Algorithms

Keywords and phrases Congestion games, Nash equilibrium, price of anarchy, online load balancing, greedy algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.17

1 Introduction

Congestion games [25] are non-cooperative games in which there is a set of selfish players competing for a set of resources, and each resource incurs a certain latency, expressed by a congestion-dependent function, to the players using it. Each player has a certain weight and an available set of strategies, where each strategy is a non-empty subset of resources, and aims at choosing a strategy minimizing her personal cost which is defined as the sum of the latencies experienced on all the selected resources. We speak of *weighted* games/players when players have arbitrary non-negative weights and of *unweighted* games/players when all players have unitary weight.

Stable outcomes in this setting are the pure Nash equilibria [24]: strategy profiles in which no player can lower her cost by unilaterally deviating to another strategy. However, they are demanding solution concepts, as they might not always exist in weighted games

[18] and, even when their existence is guaranteed, as, for instance, in unweighted games [25] and in weighted games with affine latency functions [18, 21], their computation might be an intractable problem [1, 16]. For such a reason, more relaxed solution concepts are usually considered in the literature, as ϵ-approximate pure Nash equilibria or ϵ-approximate one-round walks. An ϵ-approximate pure Nash equilibrium is the relaxation of the concept of pure Nash equilibrium in which no player can lower her cost of a factor more than $1 + \epsilon$ by unilaterally deviating to another strategy, while an ϵ-approximate one-round walk is defined as a myopic process in which players arrive in an arbitrary order and, upon arrival, each of them has to make an irrevocably strategic choice aiming at approximatively minimizing a certain cost function. In this work, we shall consider two variants of this process: in the first, players choose a strategy approximatively minimizing, up to a factor of $1 + \epsilon$, their personal cost (*selfish players*), while, in the second, players choose the strategy approximatively minimizing, up to a factor of $1 + \epsilon$, the marginal increase in the social cost (*cooperative players*) which is defined as the sum of the players' personal costs (for the case of $\epsilon = 0$, we use the term *exact one-round walk*). In particular, approximate one-round walks can be interpreted as simple greedy online algorithms for the equivalent resource selection problem associated with a given congestion game, and, in most of the cases, these algorithms are optimal in the context of online optimization of load balancing problems [9]. The worst-case efficiency of these solution concepts with respect to the optimal social cost is termed as the *ϵ-approximate price of anarchy* (for the case of pure Nash equilibria, the term *price of anarchy* [22] is adopted) and as the *competitive ratio* of ϵ-approximate one-round walks, respectively. Interesting special cases of congestion games are obtained by restricting the combinatorics of the players' strategy space. In *symmetric congestion games*, all players share the same set of strategies; in *network congestion games* the players' strategies are defined as paths in a given network; in *matroid congestion games* [1, 2], the strategy set of every player is given by the set of bases of a matroid defined over the set of available resources; in *k-uniform matroid congestion games* [15], each player can select any subset of cardinality k from a prescribed player-specific set of resources; finally, in *load balancing games*, players can only choose single resources.

To what extent does the structure of the players' strategy space influence the efficiency of decentralized solutions in congestion games? In this work, we investigate whether better performance is possible when restricting to load balancing games. Previous work established that the price of anarchy does not improve when restricting to unweighted load balancing games with polynomial latency functions [10, 20], while better bounds are possible in unweighted symmetric load balancing games with fairly general latency functions [17]. Under the assumption of identical resources with affine latency functions, improvements are also possible when restricting to both unweighted load balancing games [10, 27] and weighted symmetric load balancing games [23]. Finally, [6] proves that the price of anarchy does not improve when restricting to weighted symmetric load balancing games under polynomial latency functions. For the competitive ratio of exact one-round walks generated by cooperative players, no improvements are possible in unweighted load balancing games with affine latency functions [10, 27], while improved performance can be obtained under the additional assumption of identical resources [10] (we observe that, in this case, solutions generated by both types of players coincide); however, for weighted players, no improvements are possible even under the assumption of identical resources [9, 10]. For one-round walks generated by selfish players, instead, no specialized limitations are currently known.

Our Contribution. We obtain an almost precise picture of the cases in which improved performance can be obtained in load balancing congestion games. This is done by either solving

open problems or extending previously known results to both approximate solution concepts and more general latency functions. Specifically, we provide the following characterizations.

Let $\mathcal{C}$ be a class of non-negative and non-decreasing functions such that, for each $f \in \mathcal{C}$ and $\alpha \in \mathbb{R}_{\geq 0}$, the function g such that $g(x) = \alpha f(x)$ belongs to $\mathcal{C}$ and let $\mathcal{C}' \subset \mathcal{C}$ be the subclass of $\mathcal{C}$ such that, for each $f \in \mathcal{C}'$ and $\alpha \in \mathbb{R}_{\geq 0}$, the function h such that $h(x) = f(\alpha x)$ belongs to $\mathcal{C}'$. A function f is *semi-convex* if $xf(x)$ is convex, it is *unbounded* if $\lim_{x\to\infty} f(x) = \infty$. We prove that:

- **for weighted players:** under unbounded latency functions drawn from $\mathcal{C}'$, the approximate price of anarchy does not improve when restricting to symmetric load balancing games (this solves an open problem raised in [6], where a similar limitation was shown only with respect to pure Nash equilibria and polynomial latency functions). Under latency functions drawn from $\mathcal{C}'$, the competitive ratio of approximate one-round walks generated by selfish players does not improve when restricting to load balancing games (this solves an open problem raised in [8]). If all functions in $\mathcal{C}'$ are semi-convex, then the same limitation applies to the competitive ratio of approximate one-round walks generated by cooperative players (this generalizes results in [5, 9, 10] which hold only with respect to exact one-round walks for games with polynomial latency functions). We also provide a parametric formula for the relative bounds which we use to obtain the exact values for polynomial latency functions;
- **for unweighted players:** under latency functions drawn from $\mathcal{C}$, either the approximate price of anarchy and the competitive ratio of approximate one-round walks generated by both selfish and cooperative players do not improve when restricting to load balancing games (these generalize a result in [10, 20] which holds only with respect to pure Nash equilibria and polynomial latency functions, a result in [10, 27] which holds only with respect to exact one-round walks generated by cooperative players in games with affine latency functions, and solve an open problem raised in [8] for one-round walks generated by selfish players). Also in this case we provide a parametric formula for the relative bounds which we use to obtain the exact bounds for polynomial latency functions.

These negative results, together with the positive ones achieved by [10, 17], imply that better bounds on the approximate price of anarchy are possible only when dealing with unweighted symmetric load balancing games. However, under the additional hypothesis of identical resources, better performance is still possible. Let f be an increasing, continuous and semi-convex function. We prove that the approximate price of anarchy of weighted symmetric load balancing games with identical resources whose latency functions coincide with f is equal to $\sup_{x\in\mathbb{R}_{>0}} \sup_{\lambda\in(0,1)} \left\{ \frac{\lambda x f(x)+(1-\lambda) inv(x) f(inv(x))}{opt(x) f(opt(x))} \right\}$, where $inv(x) := \inf\{t \geq 0 : f(x) \leq (1+\epsilon) f(x/2+t)\}$ and $opt(x) := \lambda x + (1-\lambda) inv(x)$. This generalizes a result by [23] which holds only with respect to the price of anarchy under affine latency functions. Furthermore, by using the previous formula, we compute the exact price of anarchy of weighted symmetric load balancing games with identical resources and polynomial latency functions.

Finally, still for the case of identical resources, we design lower bounds on the performance of exact one-round walks in load balancing games with unweighted players (this improves and generalizes a result in [10] which holds only for affine latency functions).

Related Work. The price of anarchy in congestion games was first considered in [4] and [11] where it was independently shown that the price of anarchy is $5/2$ and $(3+\sqrt{5})/2$ for, respectively, unweighted and weighted congestion games with affine latency functions. In [11], it is also proved that no improved bounds are possible both in symmetric unweighted games

and in unweighted network games; these results were improved by [14] which shows that the price of anarchy stays the same even in symmetric unweighted network games. In [10], it is shown that the previous bounds are tight also for load balancing games. For the special case of load balancing games on identical resources, the works of [27] and [10] show that the price of anarchy is 2.012067 for unweighted games and at least 5/2 for weighted ones. In [23], it is proved that, for symmetric load balancing games, the price of anarchy drops to 4/3 if the games are unweighted, and to 9/8 if the games are weighted with identical resources. For symmetric unweighted k-uniform matroid congestion games with affine latency functions, [15] proves that the price of anarchy is at most 28/13 and at least 1.343 for a sufficiently large value of k (for $k = 5$, it is roughly 1.3428). Tight bounds on the price of anarchy of either weighted and unweighted congestion games with polynomial latency functions have been given by [3]. Under fairly general latency functions, [17] shows that the price of anarchy of unweighted symmetric load balancing games coincides with that of non-atomic congestion games (thus generalizing a first result by [19] which proves an upper bound of $\sum_{i\in[d]} \mathcal{B}_i$, where $\mathcal{B}_i$ is the ith Bell number for the case of polynomial latency functions of maximum degree equal to d), while [6] proves that assuming symmetric strategies does not lead to improved bounds in unweighted games and gives exact bounds for the case of weighted players. It also shows that, for the case of weighted players, no improvements are possible even in symmetric load balancing games with polynomial latency functions. Finally, [12] and [7] characterize the approximate price of anarchy, respectively, in unweighted and weighted games under affine latency functions.

The competitive ratio of exact one-round walks generated by cooperative players in load balancing games with polynomial latency functions has been first considered in [5], where, for the special case of affine functions, an upper bound of $3 + 2\sqrt{2}$ is provided for weighted players. For unweighted players, this result has been improved to 17/3 in [27], where it is also shown that, for identical resources, the upper bound drops to $2 + \sqrt{5}$ in spite of a lower bound of 3.0833. Finally, [10] shows matching lower bounds of $3 + 2\sqrt{2}$ and 17/3 for, respectively, weighted and unweighted players. For weighted games with polynomial latency functions, tight bounds have been given in [9]; the lower bounds, in particular, hold even for identical resources, thus improving previous results from [5]. In [10] it is also shown that, for unweighted players and identical resources, the competitive ratio lies between 4 and $\frac{2}{3}\sqrt{21} + 1$. For the case of selfish players and still under affine latency functions, [8, 13] show that the competitive ratio is $2 + \sqrt{5}$ for unweighted congestion games, while, for weighted players, [13] gives an upper bound of $4 + 2\sqrt{3}$. In this setting, no specialized results are known for restrictions to load balancing games.

2 Definitions and Notation

For two integers $0 \le k_1 \le k_2$, let $[k_1, k_2] := \{k_1, k_1 + 1, \ldots, k_2 - 1, k_2\}$ and $[k_1] := [1, k_1]$.

A *congestion game* is a tuple $\mathsf{CG} = (N, E, (\ell_e)_{e\in E}, (w_i)_{i\in N}, (\Sigma_i)_{i\in N})$, where N is a set of $n \ge 2$ players, E is a set of resources, $\ell_e : \mathbb{R}_{\ge 0} \to \mathbb{R}_{\ge 0}$ is the latency function of resource $e \in R$, and, for each $i \in N$, $w_i \ge 0$ is the weight of player i and $\Sigma_i \subseteq 2^R \setminus \emptyset$ is her set of strategies. We speak of *weighted* games/players when players have arbitrary weights and of *unweighted* games/players when $w_i = 1$ for each $i \in N$. A congestion game is *symmetric* if $\Sigma_i = \Sigma$ for each $i \in N$, i.e., if all players share the same strategy space. A *load balancing* game is a congestion game in which for each $i \in N$ and $S \in \Sigma_i$, $|S| = 1$, that is, all players' strategies are singleton sets. Given a class $\mathcal{C}$ of latency functions, let $\mathsf{W}(\mathcal{C})$ be the class of weighted congestion games, $\mathsf{U}(\mathcal{C})$ be the class of unweighted congestion games, $\mathsf{ULB}(\mathcal{C})$ be the class of unweighted load balancing games, $\mathsf{WLB}(\mathcal{C})$ be the class of weighted load balancing

games, and $\mathsf{WSLB}(\mathcal{C})$ be the class of weighted symmetric load balancing games, all having latency functions in the class $\mathcal{C}$.

A *strategy profile* is an n-tuple of strategies $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n)$, that is, a state of the game in which each player $i \in N$ is adopting strategy $\sigma_i \in \Sigma_i$, so that $\Sigma := \times_{i \in N} \Sigma_i$ denotes the set of strategy profiles which can be realized in CG. For a strategy profile $\boldsymbol{\sigma}$, the *congestion* of resource $e \in E$ in $\boldsymbol{\sigma}$, denoted as $k_e(\boldsymbol{\sigma}) := \sum_{i \in N : e \in \sigma_i} w_i$, is the total weight of the players using resource e in $\boldsymbol{\sigma}$, (observe that, in unweighted games, $k_e(\boldsymbol{\sigma})$ coincides with the number of users of resource e in $\boldsymbol{\sigma}$). The personal cost of player i in $\boldsymbol{\sigma}$ is defined as $cost_i(\boldsymbol{\sigma}) = \sum_{e \in \sigma_i} \ell_e(k_e(\boldsymbol{\sigma}))$ and each player aims at minimizing it. For the sake of conciseness, when the strategy profile $\boldsymbol{\sigma}$ is clear from the context, we write k_e in place of $k_e(\boldsymbol{\sigma})$. Fix a strategy profile $\boldsymbol{\sigma}$ and a player $i \in N$. We denote with $\boldsymbol{\sigma}_{-i}$ the restriction of $\boldsymbol{\sigma}$ to all the players other than i; moreover, for a strategy $S \in \Sigma_i$, we denote with $(\boldsymbol{\sigma}_{-i}, S)$ the strategy profile obtained from $\boldsymbol{\sigma}$ when player i changes her strategy from σ_i to S, while the strategies of all the other players are kept fixed. The quality of a strategy profile in congestion games is measured by using the *social function* $\mathsf{SUM}(\boldsymbol{\sigma}) = \sum_{i \in N} w_i cost_i(\boldsymbol{\sigma}) = \sum_{e \in E} k_e(\boldsymbol{\sigma}) \ell_e(k_e(\boldsymbol{\sigma}))$, that is, the sum of the players' personal costs. A *social optimum* is a strategy profile $\boldsymbol{\sigma}^*$ minimizing SUM. For the sake of conciseness, once a particular social optimum has been fixed, we write o_e to denote the value $k_e(\boldsymbol{\sigma}^*)$.

For any $\epsilon \geq 0$, an *ϵ-approximate pure Nash equilibrium* is a strategy profile $\boldsymbol{\sigma}$ such that, for any player $i \in N$ and strategy $S \in \Sigma_i$, $cost_i(\boldsymbol{\sigma}) \leq (1+\epsilon) cost_i(\boldsymbol{\sigma}_{-i}, S)$. We denote by $\mathsf{NE}_\epsilon(\mathsf{CG})$ the set of ϵ-approximate pure Nash equilibria of a congestion game CG. For any $\epsilon \geq 0$, an *ϵ-approximate one-round walk* is an online process in which players appear sequentially according to an arbitrary order and, upon arrival, each player irrevocably chooses a strategy approximatively minimizing a certain cost function. Let $\boldsymbol{\sigma}^i$ denote the strategy profile obtained when the first i players have performed their strategic choice, while the remaining ones have not entered the game yet (so, it may be assumed that each of them is playing the empty strategy). The i-th *selfish player* aims at minimizing her personal cost, so that $cost_i(\boldsymbol{\sigma}^i) \leq (1+\epsilon) \min_{S \in \Sigma_i} cost_i(\boldsymbol{\sigma}^{i-1}, S)$; the i-th *cooperative player* aims at minimizing the marginal increase in the social function SUM, so that $\mathsf{SUM}(\boldsymbol{\sigma}^i) - \mathsf{SUM}(\boldsymbol{\sigma}^{i-1}) \leq (1+\epsilon) \min_{S \in \Sigma_i} (\mathsf{SUM}(\boldsymbol{\sigma}^{i-1}, S) - \mathsf{SUM}(\boldsymbol{\sigma}^{i-1}))$. For $\epsilon = 0$, we speak of an *exact one-round walk*. We denote by $\mathsf{ORW}^s_\epsilon(\mathsf{CG})$ (resp. $\mathsf{ORW}^c_\epsilon(\mathsf{CG})$) the set of strategy profiles which can be constructed by an ϵ-approximate one-round walk involving selfish (resp. cooperative) players in a congestion game CG.

The *ϵ-approximate price of anarchy* of a congestion game CG is defined as $\mathsf{PoA}_\epsilon(\mathsf{CG}) = \max_{\boldsymbol{\sigma} \in \mathsf{NE}_\epsilon(\mathsf{CG})} \{\mathsf{SUM}(\boldsymbol{\sigma})/\mathsf{SUM}(\boldsymbol{\sigma}^*)\}$, where $\boldsymbol{\sigma}^*$ is a social optimum for CG. Similarly, the *competitive ratio* of ϵ-approximate one-round walks generated by selfish (resp. cooperative) players, is defined as $\mathsf{CR}^s_\epsilon(\mathsf{CG}) = \max_{\boldsymbol{\sigma} \in \mathsf{ORW}^s_\epsilon(\mathsf{CG})} \{\mathsf{SUM}(\boldsymbol{\sigma})/\mathsf{SUM}(\boldsymbol{\sigma}^*)\}$ (resp. $\mathsf{CR}^c_\epsilon(\mathsf{CG}) = \max_{\boldsymbol{\sigma} \in \mathsf{ORW}^c_\epsilon(\mathsf{CG})} \{\mathsf{SUM}(\boldsymbol{\sigma})/\mathsf{SUM}(\boldsymbol{\sigma}^*)\}$). Given a class of congestion games $\mathcal{G}$, the ϵ-approximate price of anarchy of $\mathcal{G}$ is defined as $\mathsf{PoA}_\epsilon(\mathcal{G}) = \sup_{\mathsf{CG} \in \mathcal{G}} \mathsf{PoA}_\epsilon(\mathsf{CG})$. For the case of $\epsilon = 0$, we refer to this metric simply as to the *price of anarchy*. The competitive ratio of ϵ-approximate one-round walks of $\mathcal{G}$ generated by both selfish and cooperative players is defined accordingly. Throughout the paper, we shall assume that, in any considered class of latency functions, there always exists a non-constant function, otherwise the inefficiency of all the ϵ-approximate solution concepts we consider is always equal to $1+\epsilon$.

3 Weighted Load Balancing Games

In this section, we first show that the approximate price of anarchy of weighted congestion games cannot improve even when restricting the players' strategy space to the simplest possible combinatorial structure, i.e., to the case of symmetric load balancing games.

▶ **Theorem 1.** *Let $\mathcal{C} = \{f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}\}$ be a class of non-decreasing latency functions whose members, except for the constant ones, are unbounded and such that, for any $f \in \mathcal{C}$ and $\alpha \geq 0$, the functions $g, h : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ such that $g(x) = \alpha f(x)$ and $h(x) = g(\alpha x)$ for each $x \in \mathbb{R}_{\geq 0}$ belong to $\mathcal{C}$. Then, $\mathsf{PoA}_\epsilon(\mathsf{W}(\mathcal{C})) = \mathsf{PoA}_\epsilon(\mathsf{WSLB}(\mathcal{C}))$.*

Proof Sketch. We make use of a multi-graph representation of a pair of strategy profiles for a (symmetric) load balancing game, denoted as *load balancing graph*, defined as follows: nodes are all the resources in E, and each player is associated to a weighted edge (e_1, e_2, w), where $\{e_1\}$ is denoted as her first strategy, $\{e_2\}$ is her second strategy, and w is her weight.

Let $k_1 > 0$ and $k_2 \geq 0$ be two real numbers, n be a positive integer, and f_1, f_2 be two non-constant (and so, unbounded) functions belonging to $\mathcal{C}$. Consider a load balancing graph $\mathsf{LB}(k_1, k_2)$ yielded by a directed n-ary tree, organized in $2s$ levels, numbered from 1 to $2s$, and whose edges are oriented from the root to the leaves, with the addition of n self-loops on the nodes of level $2s$. The weight of a player associated to an edge outgoing from a node at level $i \in [s]$ (resp. $i \in [s+1, 2s]$) is equal to $(k_1/n)^i$ (resp. $(k_1/n)^s(k_2/n)^{i-s}$). For $i, j \in [2]$, define $\theta_{i,j} = \frac{f_i(k_i)}{(1+\epsilon)f_j(k_j+1)}$ and $\theta_i = \theta_{i,i}$. Each resource at level i has latency $g_i(x) = \theta_1^{i-1} f_1\left(\left(\frac{n}{k_1}\right)^{i-1} x\right)$ if $i \in [1, s]$ and $g_i(x) = \theta_1^{s-1}\theta_{1,2}\theta_2^{i-s-1} f_2\left(\left(\frac{n}{k_1}\right)^{s}\left(\frac{n}{k_2}\right)^{i-s-1} x\right)$, otherwise.

For a sufficiently large n, the strategy profile $\boldsymbol{\sigma}$ in which all players select their first strategy is an ϵ-approximate pure Nash equilibrium. Towards this end, consider a player whose first strategy is a resource from level i. Since the game is symmetric, we have to consider the following cases: (1) if $i \in [1, 2s-1]$ and the player deviates to a resource from level $i+1$, her cost decreases exactly of a factor of $1+\epsilon$; (2) if $i \in [2, 2s]$ and the player deviates to a resource from level $j \leq i$, her cost does not decrease; if $i \in [1, 2s-2]$ and the player deviates to a resource from level $j > i+1$, for a sufficiently large n, her cost does not decrease. Let $\boldsymbol{\sigma}^*$ be the strategy profile in which each player plays her second strategy. We can show that, for each $M < \mathsf{PoA}_\epsilon(\mathsf{W}(\mathcal{C}))$, there exist $k_1 > 0$ and $k_2 \geq 0$ such that $\lim_{s\to\infty} \frac{\mathsf{SUM}(\boldsymbol{\sigma})}{\mathsf{SUM}(\boldsymbol{\sigma}^*)} > M$, thus proving the thesis. This technical claim, together with the full proof of the theorem, resembles a similar result used in [6, 26]. ◀

Then, we prove that no improvements are possible for approximate one-round walks when restricting to load balancing games.

▶ **Theorem 2.** *Let $\mathcal{C} = \{f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}\}$ be a class of non-decreasing latency functions such that, for any $f \in \mathcal{C}$ and $\alpha \geq 0$, the functions $g, h : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ such that $g(x) = \alpha f(x)$ and $h(x) = g(\alpha x)$ for each $x \in \mathbb{R}_{\geq 0}$ belong to $\mathcal{C}$. Then $\mathsf{CR}^s_\epsilon(\mathsf{W}(\mathcal{C})) = \mathsf{CR}^s_\epsilon(\mathsf{WLB}(\mathcal{C}))$. If all functions in $\mathcal{C}$ are semi-convex, we have that $\mathsf{CR}^c_\epsilon(\mathsf{W}(\mathcal{C})) = \mathsf{CR}^c_\epsilon(\mathsf{WLB}(\mathcal{C}))$.*

Proof Sketch. Let us start with the case of selfish players. We extend the load balancing graph $\mathsf{LB}(k_1, k_2)$ used in the proof of Theorem 1 as follows. Denote as $i(v)$ the level of resource v. For each node u in the load balancing graph, consider an arbitrary enumeration of all the n outgoing edges of u. Since each node has a unique incoming edge, we denote by $h(v) \in [n]$ the position associated to the unique edge entering v in the given ordering.

Consider the ϵ-approximate one-round walk in which players enter the game in non-increasing order of level (with respect to their first strategy) and, within the same level, players are processed in non-decreasing order of position.

For $i, j \in [2]$ and $h \in [n]$, define $\theta_{i,j}(h) = \frac{f_i\left(\frac{hk_i}{n}\right)}{(1+\epsilon)f_j(k_j+1)}$ and $\theta_i(h) = \theta_{i,i}(h)$. Resource v has latency function $g_v(x) = f_1(x)$ if $i(v) = 1$, $g_v(x) = \underbrace{\theta_1(h(v))A_u}_{A_v} f_1\left(\left(\frac{n}{k_1}\right)^{i(v)-1} x\right)$ if

d	Selfish Players	Coordinated Players	d	Selfish Players	Coordinated Players
1	7.464	5.828	6	27,089,557	7,553,550
2	90.3	56.94	7	974,588,649	222,082,591
3	1,521	780.2	8	39,729,739,895	7,400,694,480
4	32,896	13,755	9	1,809,913,575,767	275,651,917,450
5	868,567	296,476	∞	$(\Theta(d))^{d+1}$	$(\Theta(d))^{d+1}$

Figure 1 The competitive ratio of exact one-round walks generated by either selfish or cooperative players in weighted load balancing games with polynomial latency functions of maximum degree d.

$i(v) \in [2, s]$, $g_v(x) = \underbrace{\theta_{1,2}(h(v))A_u}_{A_v} f_2\left(\left(\frac{n}{k_1}\right)^s x\right)$ if $i(v) = s+1$, while, in all the other cases, $g_v(x) = \underbrace{\theta_2(h(v))A_u}_{A_v} f_2\left(\left(\frac{n}{k_1}\right)^s \left(\frac{n}{k_2}\right)^{i(v)-s-1} x\right)$, where (u, v) denotes the unique incoming edge of v and A_v is recursively defined on the basis of A_u by setting $A_v = 1$ for $i(v) = 1$, i.e., for v being the root of the tree.

The strategy profile $\boldsymbol{\sigma}$ in which all players select their first strategy is a possible outcome of an ϵ-approximate one-round walk generated by selfish players. Let $\boldsymbol{\sigma}^*$ be the strategy profile in which all players select their second strategy. As the game is not symmetric, we can assume that all players can choose among these two strategies only. We can show that, for each $M < \mathsf{CR}^s_\epsilon(\mathsf{W}(\mathcal{C}))$, there exist $k_1 > 0$ and $k_2 \geq 0$ such that $\lim_{s\to\infty} \lim_{n\to\infty} \frac{\mathsf{SUM}(\boldsymbol{\sigma})}{\mathsf{SUM}(\boldsymbol{\sigma}^*)} > M$, thus proving the thesis. Again, this technical claim, together with the full proof of the theorem, resembles a similar result used in [6, 26]. For the case of cooperative players, it suffices considering the same load balancing graph, with $n = 1$ and $\theta_{i,j}(1) = \frac{k_i f_i(k_i)}{(1+\epsilon)((k_j+1)f_i(k_j+1)-k_j f_j(k_j))}$. ◀

3.1 Polynomial Latency Functions

Consider the class $\mathcal{P}(d)$ of polynomials with non-negative coefficients and maximum degree d. Observe that this class of latency functions satisfies the hypothesis required by Theorems 1 and 2. By applying similar arguments to those used in [3], we get $\mathsf{CR}^s_\epsilon(\mathcal{P}(d)) = (\varphi_{\epsilon,d+1})^{d+1}$ and $\mathsf{CR}^c_\epsilon(\mathcal{P}(d)) = \left(\varphi'_{\epsilon,d+1}\right)^{d+1}$, where $\varphi_{\epsilon,d+1}$ and $\varphi'_{\epsilon,d+1}$ are the unique solutions of the equations $\frac{x^{d+1}}{d+1} - (1+\epsilon)(x+1)^d = 0$ and $(2+\epsilon)x^{d+1} - (1+\epsilon)(x+1)^{d+1} = 0$, respectively. Observe that $\varphi'_{\epsilon,d+1} = \frac{1}{\sqrt[d+1]{\frac{2+\epsilon}{1+\epsilon}}-1}$ which generalizes the bounds given in [9] for the case $\epsilon = 0$. Some values for the case of $\epsilon = 0$ are reported in Figure 1.

4 Unweighted Load Balancing Games

In this section, we first show that the ϵ-approximate price of anarchy of unweighted congestion games cannot improve when restricting to load balancing games.

▶ **Theorem 3.** *Let $\mathcal{C}$ be a class of non-decreasing latency functions such that $f \in \mathcal{C}, \alpha \geq 0 \Rightarrow \alpha f \in \mathcal{C}$. Then $\mathsf{PoA}_\epsilon(\mathsf{ULB}(\mathcal{C})) = \mathsf{PoA}_\epsilon(\mathsf{U}(\mathcal{C}))$.*

Proof Sketch. Let $k_1, o_1, o_2 > 0$ and $k_2 \geq 0$ be non-negative integers. Consider a load balancing game defined by a multi-partite directed graph $\mathsf{LB}(k_1, k_2, o_1, o_2)$ organized in $2s$ levels, numbered from 1 to $2s$, and defined as follows. For each $i \in [s]$ (resp. $i \in [s+1, 2s]$) there are $o_1^{s-i}k_1^{i-1}o_2^s$ (resp. $o_2^{2s-i}k_2^{i-s-1}k_1^s$) nodes/resources. Edges can only connect nodes of consecutive levels, except for nodes at level $2s$, each of which has k_2 self-loops. The

out-degree of each node at level $i \in [s]$ (resp. $i \in [s+1, 2s]$) is k_1 (resp. k_2), and the in-degree of each node at level $i \in [2, s]$ (resp. $i \in [s+1, 2s]$ without considering self-loops) is o_1 (resp. o_2); observe that this configuration can be realized since the total number of nodes at level $i \in [s-1]$ (resp. $i = s$, resp. $i \in [s+1, 2s-1]$) multiplied by k_1 (resp. k_1, resp. k_2) is equal to the number of nodes at level $i+1$ multiplied by o_1 (resp. o_2, resp. o_2). For $i, j \in [2]$, define $\theta_{i,j} = \frac{f_i(k_i)}{(1+\epsilon)f_j(k_j+1)}$ and $\theta_i = \theta_{i,i}$. Each resource at level i has latency function $g_i(x) = \theta_1^{i-1} f_1(x)$ if $i \in [s]$, and $g_i(x) = \theta_1^{s-1}\theta_{1,2}\theta_2^{i-s-1} f_2(x)$ otherwise.

Let $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma}^*$ be the strategy profiles in which all players select their first and second strategy, respectively. As the game is not symmetric, we can assume that all players can choose among these two strategies only. Analogously to Theorem 1, it is possible to show that, for any $M < \mathsf{PoA}_\epsilon(\mathsf{U}(\mathcal{C}))$, there exist suitable non-negative integers k_1, k_2, o_1, o_2 such that $\boldsymbol{\sigma}$ is an ϵ-approximate pure Nash equilibrium and $\lim_{s\to\infty} \frac{\mathsf{SUM}(\boldsymbol{\sigma})}{\mathsf{SUM}(\boldsymbol{\sigma}^*)} > M$. ◀

Then, we prove a similar limitation for approximate one-round walks.

▶ **Theorem 4.** *Let $\mathcal{C} = \{f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}\}$ be a class of non-decreasing latency functions such that $f \in \mathcal{C}, \alpha \geq 0 \Rightarrow \alpha f \in \mathcal{C}$. Then $\mathsf{CR}^s_\epsilon(\mathsf{U}(\mathcal{C})) = \mathsf{CR}^s_\epsilon(\mathsf{WLB}(\mathcal{C}))$. If functions of $\mathcal{C}$ are semi-convex, we have that $\mathsf{CR}^c_\epsilon(\mathsf{U}(\mathcal{C})) = \mathsf{CR}^c_\epsilon(\mathsf{WLB}(\mathcal{C}))$.*

Proof Sketch. Let us start with the case of selfish players. Define $j(i) = 1$ if $i \in [s]$ and $j(i) = 2$ otherwise. We extend the load balancing graph $\mathsf{LB}(k_1, k_2, o_1, o_2)$ used in the proof of Theorem 3 according to the following recursive procedure.

- **Base Case:** partition the resources of the first level (resp. second level) in $o_{j(2)}$ (resp. $k_{j(1)}$) groups of equal size, and add edges from the first level to the second one in such a way that each resource in the first level has exactly $k_{j(1)}$ outgoing edges, each ending in a different group of the second level, and each resource in the second level has exactly $o_{j(2)}$ incoming edges, each coming from a different group of the first level; number the groups of the second level from 1 to $k_{j(1)}$ and label each resource with the number associated to the group it belongs to, for an illustrating example see figure 2 where resources belonging to different groups at level 1 are represented with different colors, resources belonging to different groups at level 2 belong to different squares and they are labeled with the number of the square they belong to.
- **Inductive Case:** as inductive hypothesis, suppose that resources at level $i \in [2s-1]$ have been partitioned into $m(i)$ groups of equal size and labeled with values from 1 to $k_{j(i-1)}$, where each label is assigned to $m(i)/k_{j(i-1)}$ distinct groups, and that all the edges from level $i-1$ to level i have been added. Partition resources at level $i+1$ in a temporary partition of $m(i)$ groups of equal size, and consider a bijective correspondence between groups at level i and groups at level $i+1$ (in Figure 2, groups at levels 2 and 3 which are in bijective correspondence, have been depicted in the same dashed square). Partition each group at level i into $o_{j(i+1)}$ subgroups of equal size, and the corresponding group at level $i+1$ into $k_{j(i)}$ subgroups of equal size (this defines the final partitioning of nodes at level $i+1$ into $m(i)k_{j(i)}$ groups), and add edges from the first group to the second one in the same way as described in the basic case, i.e. each resource in the first group has exactly $k_{j(i)}$ outgoing edges, each ending in a different subgroup of the second group, and each resource in the second group has exactly $o_{j(i)}$ incoming edges, each coming from a different subgroup of the first group. For each group at level $i+1$, number its subgroups with values from 1 to $k_{j(i)}$ and label each resource with the number associated to subgroup it belongs to. For instance, in Figure 2, consider an arbitrary dashed square including two groups at levels 2 and 3 which are in bijective correspondence. Analogously to the base case, resources belonging to different subgroups of the first (resp. second) group are represented with different colors (resp. belong to different squares and are labeled with the number of the square they belong to).

Let $h(v)$ be the label of resource v. Consider the ϵ-approximate one-round walk in which players enter the game in non-increasing order of level (with respect to their first strategy) and, within the same level, players are processed in non-decreasing order of position defined by labeling function h.

For $i,j \in [2]$ and $h \in k_i$, define $\theta_{i,j}(h) = \frac{f_i(h)}{(1+\epsilon)f_j(k_j+1)}$ and $\theta_i(h) = \theta_{i,i}(h)$. Resource v has latency function $g_v(x) = f_1(x)$ if $i(v) = 1$, $g_v(x) = \underbrace{\theta_1(h(v))A_u}_{A_v} f_1(x)$ if $i(v) \in [2,s]$, $g_v(x) = \underbrace{\theta_{1,2}(h(v))A_u}_{A_v} f_2(x)$ if $i(v) = s+1$, and $g_v(x) = \underbrace{\theta_2(h(v))A_u}_{A_v} f_2(x)$ otherwise, where (u,v) is an arbitrary incoming edge of v and A_v is recursively defined on the basis of A_u by setting $A_v = 1$ for $i(v) = 1$. By using the recursive structure of the load balancing graph, one can prove, by induction on the level of each resource v, that $A_u = A_{u'}$ if (u,v) and (u',v) are both edges of the load balancing graph, so that the definition of g_v is independent of the particular incoming edge of v.

The strategy profile $\boldsymbol{\sigma}$ all players select their first strategy is a possible outcome of an ϵ-approximate one-round walk generated by selfish players. Let $\boldsymbol{\sigma}^*$ be the strategy profile in which all players select their second strategy. We can show that, for each $M < \mathsf{CR}^s_\epsilon(\mathsf{U}(\mathcal{C}))$, there exist suitable non-negative integers k_1, k_2, o_1, o_2 such that $\lim_{s\to\infty} \lim_{n\to\infty} \frac{\mathsf{SUM}(\boldsymbol{\sigma})}{\mathsf{SUM}(\boldsymbol{\sigma}^*)} > M$, thus proving the claim. For the case of cooperative players, it suffices considering the same load balancing graph with $\theta_{i,j}(h) = \frac{h_i f_i(h_i) - (h_i-1)f_i(h_i-1)}{(1+\epsilon)((k_j+1)f_i(k_j+1) - k_j f_j(k_j))}$. ◀

4.1 Polynomial Latency Functions

Consider the class $\mathcal{P}(d)$ of polynomials with non-negative coefficients and maximum degree d. For ϵ-approximate one-round walks generated by cooperative players, by using similar arguments to those exploited in [3], one can prove that $\mathsf{CR}^c_\epsilon(\mathsf{ULB}(\mathcal{P}(d)))$ is equal to $\mathsf{CR}^c_\epsilon(\mathsf{WLB}(\mathcal{P}(d)))$ if $\varphi'_{d,\epsilon}$ is an integer (see Subsection 3.1), otherwise we get $\mathsf{CR}^c_\epsilon(\mathsf{ULB}(\mathcal{P}(d))) = \gamma_{d,\epsilon}\left(\left\lfloor \varphi'_{d,\epsilon} \right\rfloor\right)$, where $\gamma_{d,\epsilon}(k) := k^{d+1} + x_{d,\epsilon}(k)\left(-k^{d+1} + (1+\epsilon)\cdot\left((k+1)^{d+1} - k^{d+1}\right)\right)$, and $x_{d,\epsilon}(k)$ is such that $\gamma_{d,\epsilon}(k) = \gamma_{d,\epsilon}(k+1)$. Some values for the case of $\epsilon = 0$ are reported in Figure 3. For the case of selfish players, by using the approach in [7], we get that $\mathsf{CR}^s_0(\mathsf{ULB}(\mathcal{P}(1))) = 2+\sqrt{5}$, $\mathsf{CR}^s_0(\mathsf{ULB}(\mathcal{P}(2))) = \frac{3383}{90}$ and $\mathsf{CR}^s_0(\mathsf{ULB}(\mathcal{P}(3))) = \frac{17929}{34}$.

5 The Case of Identical Resources

In this section, we characterize the approximate price of anarchy of weighted symmetric load balancing games with identical resources having semi-convex latency functions. We start by showing the upper bound.

▶ **Theorem 5** (Upper bound). *Let $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ be a non-decreasing and semi-convex latency function. Let $\mathsf{WSILG}(f)$ be the class of weighted symmetric load balancing games with identical resources having latency function f. For any $\epsilon \geq 0$, let*

$$\begin{aligned} inv(x) &:= \inf\{t \geq 0 : f(x) \leq (1+\epsilon)f(x/2+t)\},\\ opt(x,\lambda) &:= \lambda x + (1-\lambda)inv(x),\\ upp(x,\lambda) &:= \frac{\lambda x f(x) + (1-\lambda)inv(x)f(inv(x))}{opt(x,\lambda)f(opt(x,\lambda))}. \end{aligned}$$

If $inv(x) \neq 0$ for each $x \in \mathbb{R}_{>0}$, then:

$$\mathsf{PoA}_\epsilon(\mathsf{WSILG}(f)) \leq \sup_{x\in\mathbb{R}_{>0}} \max_{\lambda\in(0,1)} upp(x,\lambda). \tag{1}$$

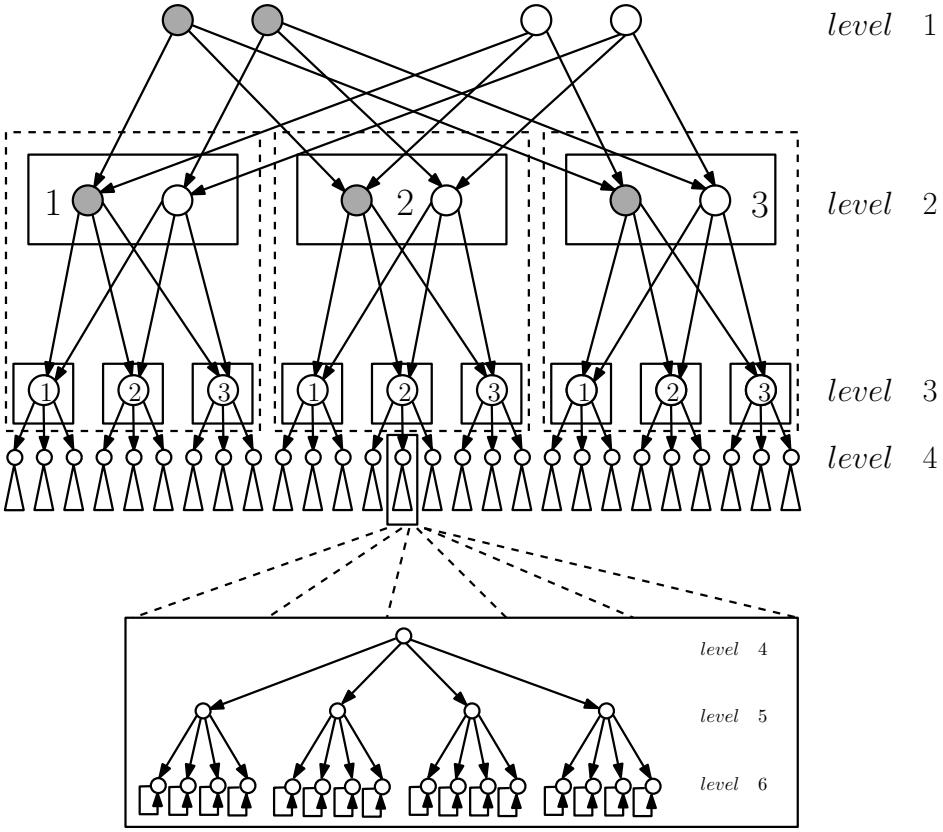


Figure 2 The load balancing graph described in the proof of Theorems 3 and 4, with $s = 3$, $k_1 = 3$, $o_1 = 2$, $k_2 = 4$ and $o_2 = 1$. We also describe the partitioning and labeling structures used in the proof of Theorem 4.

d	Competitive Ratio	d	Competitive Ratio	d	Competitive Ratio
1	5.66	4	13,170	7	220,349,064
2	55.46	5	289,648	8	7,022,463,077
3	755.2	6	7,174,495	∞	$(\Theta(d))^{d+1}$

Figure 3 The competitive ratio of exact one-round walks generated by cooperative players in unweighted load balancing games with polynomial latency functions of maximum degree d.

Proof Sketch. Let $\mathsf{WSILG}(f, W, m) \subseteq \mathsf{WSILG}(f)$ be the subclass of load balancing games having m resources and such that $\sum_{i \in N} w_i = W$. First, we prove that the optimal social cost of games in $\mathsf{WSILG}(f, W, m)$ is lower bounded by the cost of a strategy profile $\boldsymbol{\sigma}^*(W, m)$ in which all resources have the same congestion, so that $\mathsf{SUM}(\boldsymbol{\sigma}^*(W, m)) = Wf\left(\frac{W}{m}\right)$.

Furthermore, we prove that the supremum of the social cost over all ϵ-approximate pure Nash equilibria of games in $\mathsf{WSILG}(f, W, m)$ is upper bounded by the supremum of the social cost over all strategy profiles $\boldsymbol{\sigma}(m, x, h)$ in which all the resources can have three possible congestions, namely x, y, z, such that $z = inv(x) \leq y \leq x$, one resource has congestion y and $h \in [0, m-1]$ resources have congestion equal to x, so that $\mathsf{SUM}(\boldsymbol{\sigma}(m, x, h)) = hxf(x) + yf(y) + (m-h-1)inv(x)f(inv(x))$. Observe that it must be $W = hx + y + (m-h-1)inv(x)$.

One can show that that:

$$\begin{aligned}
&\mathsf{PoA}_\epsilon(\mathsf{WSILG}(f)) && (2)\\
&= \sup_{W\geq 0, m\in\mathbb{N}} \mathsf{PoA}_\epsilon(\mathsf{WSILG}(f,W,m))\\
&\leq \sup_{m\in\mathbb{N},h\in[0,m-1],x\geq 0,y:inv(x)\leq y\leq x} \frac{\mathsf{SUM}(\boldsymbol{\sigma}(m,x,h))/m}{\mathsf{SUM}(\boldsymbol{\sigma}(hx+y+(m-h-1)inv(x)))/m}\\
&= \sup_{m\in\mathbb{N},h\in[0,m-1],x\geq 0,y:inv(x)\leq y\leq x} \frac{\frac{hxf(x)+yf(y)+(m-h-1)inv(x)f(inv(x))}{m}}{\left(\frac{hx+y+(m-h-1)inv(x)}{m}\right) f\left(\frac{hx+y+(m-h-1)inv(x)}{m}\right)}\\
&= \lim_{m\to\infty} \sup_{h\in[0,m-1],x\geq 0,y:inv(x)\leq y\leq x} \frac{\frac{hxf(x)+yf(y)+(m-h-1)inv(x)f(inv(x))}{m}}{\left(\frac{hx+y+(m-h-1)inv(x)}{n}\right) f\left(\frac{hx+y+(m-h-1)inv(x)}{m}\right)} && (3)\\
&= \sup_{x\in\mathbb{R}_{>0}} \max_{\lambda\in(0,1)} \frac{\lambda x f(x)+(1-\lambda)inv(x)f(inv(x))}{opt(x,\lambda)f(opt(x,\lambda))}\\
&= \sup_{x\in\mathbb{R}_{>0}} \max_{\lambda\in(0,1)} upp(x,\lambda) && (4)
\end{aligned}$$

thus proving the claim (in (3) we have replaced h/m with λ, $(m-h-1)/m$ with $1-\lambda$ and y/m with 0). ◀

We show that, under mild assumptions, a tight lower bound can be obtained.

▶ **Theorem 6** (Lower Bound). *For any $\epsilon \geq 0$, let $\lambda^*(x) := \arg\max_{\lambda\in(0,1)} upp(x,\lambda)$ for any $x \in \mathbb{R}_{\geq 0}$. If $\lambda^*(x) \leq \frac{1}{2}$ and $opt(x,\lambda^*(x)) - x/2 \geq 0$, then*

$$\mathsf{PoA}_\epsilon(\mathsf{WSILG}(f)) = \sup_{x\in\mathbb{R}_{>0}} \max_{\lambda\in(0,1)} upp(x,\lambda). \quad (5)$$

Proof Sketch. Given $m \in \mathbb{N}$, let $h(m) \in [m]$. We prove that, if $h(m)/m$ approaches $\lambda^*(x)$ for $m \to \infty$, the strategy profiles $\boldsymbol{\sigma}^* := \boldsymbol{\sigma}^*(mx+y+(m-h-1)inv(x), m)$ and $\boldsymbol{\sigma} := \boldsymbol{\sigma}(m,x,h(m))$ defined in the proof of Theorem 5, can be enforced as an optimal strategy profile and an ϵ-approximate pure Nash equilibrium for the relative game, respectively. Thus, by using similar arguments to those exploited to obtain (4), we get

$$\lim_{m\to\infty} \sup_{x\geq 0,y:inv(x)\leq y\leq x} \frac{\mathsf{SUM}(\boldsymbol{\sigma})}{\mathsf{SUM}(\boldsymbol{\sigma}^*)} = \sup_{x\in\mathbb{R}_{>0}} upp(x,\lambda^*(x)),$$

thus concluding the proof. ◀

5.1 Polynomial Latency Functions

By exploiting (5), we derive exact bounds on the price of anarchy of weighted symmetric load balancing games with identical resources having polynomial latency functions. In Figure 4, we show a comparison between the cases of general and identical resources with respect to the price of anarchy for games with polynomial latency functions.

▶ **Theorem 7.** *Let $\mathcal{P}(d)$ be the class of polynomial latency functions of maximum degree d. Then,* $\mathsf{PoA}_0(\mathcal{P}(d)) = \frac{d^d(2^{d+1}-1)^{d+1}}{2^d(d+1)^{d+1}(2^d-1)^d} \in \Theta\left((2+o(1))^d\right)$.

d	Identical Resources	General Resources	d	Identical Resources	General Resources
1	1.125	2.618	6	7.544	14,099
2	1.412	9.909	7	12.866	118,926
3	1.946	47.82	8	22.478	1,101,126
4	2.895	277	9	39.984	11,079,429
5	4.571	1,858	∞	$\Theta\left((2+o(1))^d\right)$	$\left(\Theta\left(\frac{d}{\log d}\right)\right)^{d+1}$

Figure 4 The price of anarchy of weighted symmetric load balancing games with polynomial latency functions of maximum degree d: a comparison between the cases of identical and general resources.

5.2 Lower Bounds for Exact One-Round Walks

The following construction gives a class of lower bounding instances for exact one-round walks generated by selfish/cooperative players in load balancing games with identical resources having latency function f. Fix $n \in \mathbb{N}$ and a sequence of integers $1 = o_1 \leq o_2 \leq \ldots \leq o_n$. Let $E = E_0 \supset E_1 \supset E_2 \supset \ldots \supset E_n \supset E_{n+1} = \emptyset$ be a sequence of sets of resources such that $(|E_{i-1}| - |E_i|)o_i = |E_i|$ (observe that such a sequence exists). For any $i \in [n]$, we have $|E_i|$ players of type i whose set of strategies is E_{i-1}. Suppose that players enter the game in non-decreasing order with respect to their type. One can easily prove that the strategy profile $\boldsymbol{\sigma}$ in which each player of type i selects a different resource $e \in E_i$ is a possible outcome for an exact one-round walk generated by selfish/cooperative players. Consider the strategy profile in which, for any resource $e \in E_{i-1} \setminus E_i$, there are exactly o_i players of type i selecting e. We get:

$$\mathsf{CR}_0^s(\{f\}) \geq \frac{\mathsf{SUM}(\boldsymbol{\sigma})}{\mathsf{SUM}(\boldsymbol{\sigma}^*)} = \frac{\sum_{i=1}^{n}(|E_i| - |E_{i+1}|)if(i)}{\sum_{i=1}^{n}(|E_{i-1}| - |E_i|)o_i f(o_i)}. \tag{6}$$

For linear latency functions, by using $n = 10^{13}$ and $o_i = \left\lfloor \frac{i44411}{100000} + 1 + \left\lfloor \frac{\sqrt{i}}{7} \right\rfloor \right\rfloor$, by (6), we get a lower bound of at least 4.0009 which improves the currently known lower bound of 4 given in [10]. We conjecture that a tight class of lower bounding instances for linear and more general polynomial latency functions is given by the union of all the instances described above, over all values of $n \in \mathbb{N}$ and all sequences $(o_i)_{i\in[n]}$.

6 Open Problems

Our work leaves two open problems. The first is to understand whether better performance is possible for approximate one-round walks in weighted symmetric load balancing games (we conjecture this is not the case), while the second is to give upper bounds on the performance of one-round walks in weighted and unweighted load balancing games with identical resources.

References

1 H. Ackermann, H. Röglin, and B. Vöcking. On the impact of combinatorial structure on congestion games. *Journal of ACM*, 55(6), 2008.

2 H. Ackermann, H. Röglin, and B. Vöcking. Pure Nash equilibria in player-specific and weighted congestion games. *Theoretical Computer Science*, 410(17):1552–1563, 2009.

3 S. Aland, D. Dumrauf, M. Gairing, B. Monien, and F. Schoppmann. Exact price of anarchy for polynomial congestion games. *SIAM Journal on Computing*, 40(5):1211–1233, 2011.

4 B. Awerbuch, Y. Azar, and L. Epstein. The price of routing unsplittable flow. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 57–66, 2005.

5 B. Awerbuch, Y. Azar, E. F. Grove, M.-Y. Kao, P. Krishnan, and J. S. Vitter. Load balancing in the L_p norm. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 383–391, 1995.

6 K. Bhawalkar, M. Gairing, and T. Roughgarden. Weighted congestion games: price of anarchy, universal worst-case examples, and tightness. *ACM Transactions on Economics and Computation*, 2(4):1–23, 2014.

7 V. Bilò. A unifying tool for bounding the quality of non-cooperative solutions in weighted congestion games. In *Proceedings of the 10th Workshop on Approximation and Online Algorithms (WAOA)*, volume 7846 of *LNCS*, pages 229–241, 2012.

8 V. Bilò, A. Fanelli, M. Flammini, and L. Moscardelli. Performances of one-round walks in linear congestion games. *Theory of Computing Systems*, 49(1):24–45, 2011.

9 I. Caragiannis. Better bounds for online load balancing on unrelated machines. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 972–981, 2008.

10 I. Caragiannis, M. Flammini, C. Kaklamanis, P. Kanellopoulos, and L. Moscardelli. Tight bounds for selfish and greedy load balancing. *Algorithmica*, 61(3):606–637, 2011.

11 G. Christodoulou and E. Koutsoupias. The price of anarchy of finite congestion games. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 67–73, 2005.

12 G. Christodoulou, E. Koutsoupias, and P. G. Spirakis. On the performance of approximate equilibria in congestion games. *Algorithmica*, 61(1):116–140, 2011.

13 G. Christodoulou, V. S. Mirrokni, and A. Sidiropoulos. Convergence and approximation in potential games. *Theoretical Computer Science*, 438:13–27, 2012.

14 J. Correa, J. de Jong, B. de Keijzer, and M. Uetz. The curse of sequentiality in routing games. In *Proceedings of the 11th International Conference on Web and Internet Economics (WINE)*, volume 9470 of *LNCS*, pages 258–271, 2015.

15 J. de Jong, M. Klimm, and M. Uetz. Efficiency of equilibria in uniform matroid congestion games. In *Proceedings of the 9th International Symposium on Algorithmic Game Theory (SAGT)*, volume 9928 of *LNCS*, pages 105–116, 2016.

16 A. Fabrikant, C. H. Papadimitriou, and K. Talwar. The complexity of pure Nash equilibria. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 604–612, 2004.

17 D. Fotakis. Stackelberg strategies for atomic congestion games. *Theory of Computing Systems*, 47(1):218–249, 2010.

18 D. Fotakis, S. Kontogiannis, and P. Spirakis. Selfish unsplittable flows. *Theoretical Computer Science*, 348:226–239, 2005.

19 M. Gairing, T. Lücking, M. Mavronicolas, and B. Monien. The price of anarchy for polynomial social cost. *Theoretical Computer Science*, 369(1–3):116–135, 2006.

20 M. Gairing and F. Schoppmann. Total latency in singleton congestion games. In *Proceedings of the Third International Workshop on Internet and Network Economics (WINE)*, volume 4858 of *LNCS*, pages 381–387, 2007.

21 T. Harks and M. Klimm. On the existence of pure Nash equilibria in weighted congestion games. *Mathematics of Operations Research*, 37(3):419–436, 2012.

22 E. Koutsoupias and C. Papadimitriou. Worst-case equilibria. In *Proceedings of the 16th International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1653 of *LNCS*, pages 404–413, 1999.

23 T. Lücking, M. Mavronicolas, B. Monien, and M. Rode. A new model for selfish routing. *Theoretical Computer Science*, 406(3):187–2006, 2008.

24 J. F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Science*, 36(1):48–49, 1950.

25 R. W. Rosenthal. A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory*, 2(1):65–67, 1973.

26 T. Roughgarden. Intrinsic robustness of the price of anarchy. *Journal of ACM*, 62(5):32:1–32:42, 2015.

27 S. Suri, C. Tóth, and Y. Zhou. Selfish load balancing and atomic congestion games. *Algorithmica*, 47(1):79–96, 2007.

Tight Lower Bounds for the Complexity of Multicoloring*†

Marthe Bonamy[1], Łukasz Kowalik[2], Michał Pilipczuk[3], Arkadiusz Socała[4], and Marcin Wrochna[5]

1 **CNRS, LaBRI, Bordeaux, France**
marthe.bonamy@labri.fr
2 **University of Warsaw, Warsaw, Poland**
kowalik@mimuw.edu.pl
3 **University of Warsaw, Warsaw, Poland**
michal.pilipczuk@mimuw.edu.pl
4 **University of Warsaw, Warsaw, Poland**
arkadiusz.socala@mimuw.edu.pl
5 **University of Warsaw, Warsaw, Poland**
m.wrochna@mimuw.edu.pl

Abstract

In ¢the *multicoloring* problem, also known as *(a:b)-coloring* or *b-fold coloring*, we are given a graph G and a set of a colors, and the task is to assign a subset of b colors to each vertex of G so that adjacent vertices receive disjoint color subsets. This natural generalization of the classic coloring problem (the $b = 1$ case) is equivalent to finding a homomorphism to the Kneser graph $KG_{a,b}$, and gives relaxations approaching the fractional chromatic number.

We study the complexity of determining whether a graph has an $(a{:}b)$-coloring. Our main result is that this problem does not admit an algorithm with running time $f(b) \cdot 2^{o(\log b) \cdot n}$, for any computable $f(b)$, unless the Exponential Time Hypothesis (ETH) fails. A $(b+1)^n \cdot \mathrm{poly}(n)$-time algorithm due to Nederlof [2008] shows that this is tight. A direct corollary of our result is that the graph homomorphism problem does not admit a $2^{\mathcal{O}(n+h)}$ algorithm unless ETH fails, even if the target graph is required to be a Kneser graph. This refines the understanding given by the recent lower bound of Cygan et al. [SODA 2016].

The crucial ingredient in our hardness reduction is the usage of *detecting matrices* of Lindström [Canad. Math. Bull., 1965], which is a combinatorial tool that, to the best of our knowledge, has not yet been used for proving complexity lower bounds. As a side result, we prove that the running time of the algorithms of Abasi et al. [MFCS 2014] and of Gabizon et al. [ESA 2015] for the r-monomial detection problem are optimal under ETH.

1998 ACM Subject Classification G.2.2 Graph Theory, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases multicoloring, Kneser graph homomorphism, ETH lower bound

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.18

* A full version of the paper is available at https://arxiv.org/abs/1607.03432.

† Work supported by the National Science Centre of Poland, grants number 2013/11/D/ST6/03073 (MP, MW) and 2015/17/N/ST6/01224 (AS). The work of Ł. Kowalik is a part of the project TOTAL that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 677651). Michał Pilipczuk is supported by the Foundation for Polish Science (FNP) via the START stipend programme.

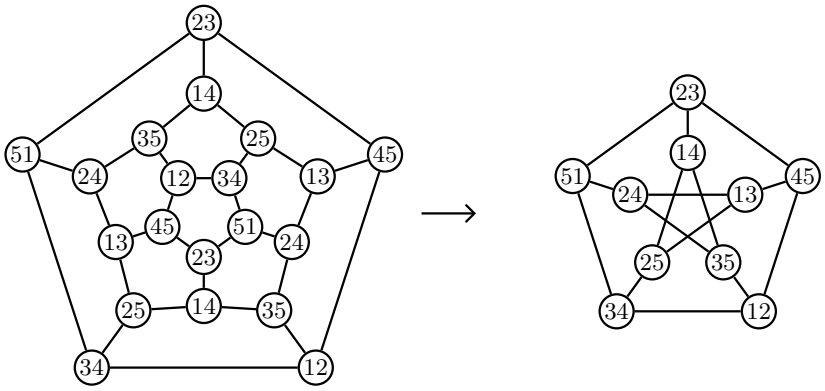


Figure 1 A (5:2)-coloring of the dodecahedron (left) which can be seen as a homomorphism to $KG_{5,2}$ (the Petersen graph, right). The homomorphism is given by identifying the pairs of opposite vertices in the corresponding regular solid.

1 Introduction

The complexity of determining the chromatic number of a graph is undoubtedly among the most intensively studied computational problems. Countless variants and generalizations of graph colorings have been introduced and investigated. Here, we focus on *multicolorings*, also known as $(a{:}b)$-colorings. In this setting, we are given a graph G, a palette of a colors, and a number $b \leq a$. An *$(a{:}b)$-coloring* of G is any assignment of b distinct colors to each vertex so that adjacent vertices receive disjoint subsets of colors. The $(a{:}b)$-COLORING problem asks whether G admits an $(a{:}b)$-coloring. For $b = 1$ we obtain the classic graph coloring problem. The smallest a for which an $(a{:}b)$-coloring exists is called the *b-fold chromatic number*, denoted by $\chi_b(G)$.

The motivation behind $(a{:}b)$-colorings can be perhaps best explained by showing the connection with the *fractional chromatic number*. For a graph G, it is denoted as $\chi_f(G)$ and defined as the optimum value of the natural LP relaxation of the problem of computing the chromatic number of G, expressed as finding a cover of the vertex set using the minimum possible number of independent sets. It can be easily seen that by relaxing the standard coloring problem by allowing b times more colors while requiring that every vertex receives b colors and adjacent vertices receive disjoint subsets, with increasing b we approximate $\chi_f(G)$ better and better. Consequently, $\lim_{b\to\infty} \chi_b(G)/b = \chi_f(G)$.

Another connection concerns *Kneser graphs*. Recall that for positive integers a, b with $b < a/2$, the Kneser graph $KG_{a,b}$ has all b-element subsets of $\{1, 2, \ldots, a\}$ as vertices, and two subsets are considered adjacent if and only if they are disjoint. For instance, $KG_{5,2}$ is the well-known Petersen graph (see Fig. 1, right). Thus, $(a{:}b)$-coloring of a graph G can be interpreted as a homomorphism from G to the Kneser graph $KG_{a,b}$ (see Fig. 1). Kneser graphs are well studied in the context of colorings, mostly due to the celebrated result of Lovász [28], who determined their chromatic number, initiating the field of topological combinatorics.

Multicolorings and $(a{:}b)$-colorings have been studied both from combinatorial [6, 12, 26] and algorithmic [5, 18, 19, 24, 25, 29, 30, 33] points of view. The main real-life motivation comes from the problem of assigning frequencies to nodes in a cellular network so that adjacent nodes receive disjoint sets of frequencies on which they can operate. This makes (near-)planar and distributed settings particularly interesting for practical applications. We refer to the survey of Halldórsson and Kortsarz [17] for a broader discussion.

In this paper we focus on the paradigm of exact exponential time algorithms: given a graph G on n vertices and numbers $a \geq b$, we would like to determine whether G is $(a{:}b)$-colorable as quickly as possible. Since the problem is already NP-hard for $a = 3$ and $b = 1$, we do not expect it to be solvable in polynomial time, and hence look for an efficient exponential-time algorithm. A straightforward dynamic programming approach yields an algorithm with running time[1] $\mathcal{O}^\star(2^n \cdot (b+1)^n)$ as follows. For each function $\eta\colon V(G) \to \{0, 1, \ldots, b\}$ and each $k = 0, 1, \ldots, a$, we create one boolean entry $D[\eta, k]$ denoting whether one can choose k independent sets in G so that every vertex $v \in V(G)$ is covered exactly $\eta(v)$ times. Then value $D[\eta, k]$ can be computed as a disjunction of values $D[\eta', k-1]$ over η' obtained from η by subtracting 1 on vertices from some independent set in G.

This simple algorithm can be improved by finding an appropriate algebraic formula for the number of $(a{:}b)$-colorings of the graph and using the inclusion-exclusion principle to compute it quickly, similarly as in the case of standard colorings [2]. Such an algebraic formula was given by Nederlof [32, Theorem 3.5] in the context of a more general Multi Set Cover problem. Nederlof also observed that in the case of $(a{:}b)$-coloring, a simple application of the inclusion-exclusion principle to compute the formula yields an $\mathcal{O}^\star((b+1)^n)$-time exponential-space algorithm. Hua et al. [21] noted that the formulation of Nederlof [32] for Multi Set Cover can be also used to obtain a polynomial-space algorithm for this problem. By taking all maximal independent sets to be the family in the Multi Set Cover problem, and applying the classic Moon-Moser upper bound on their number [31], we obtain an algorithm for $(a{:}b)$-coloring that runs in time $\mathcal{O}^\star(3^{n/3} \cdot (b+1)^n)$ and uses polynomial space. Note that by plugging $b = 1$ to the results above, we obtain algorithms for the standard coloring problem using $\mathcal{O}^\star(2^n)$ time and exponential space, or using $\mathcal{O}^\star(2.8845^n)$ time and polynomial space, which almost matches the fastest known procedures [2].

The complexity of $(a{:}b)$-coloring becomes particularly interesting in context of the Graph Homomorphism problem: given graphs G and H, with n and h vertices respectively, determine whether G admits a homomorphism to H. By the celebrated result of Hell and Nešetřil [20] the problem is in P if H is bipartite and NP-complete otherwise. For quite a while it was open whether there is an algorithm for Graph Homomorphism running in time $2^{\mathcal{O}(n+h)}$. It was recently answered in the negative by Cygan et al. [9]; more precisely, they proved that an algorithm with running time $2^{o(n \log h)}$ contradicts the Exponential Time Hypothesis (ETH) of Impagliazzo et al. [22]. However, Graph Homomorphism is a very general problem, hence researchers try to uncover a more fine-grained picture and identify families of graphs $\mathcal{H}$ such that the problem can be solved more efficiently whenever $H \in \mathcal{H}$. For example, Fomin, Heggernes and Kratsch [13] showed that when H is of treewidth at most t, then Graph Homomorphism can be solved in time $\mathcal{O}^\star((t+3)^n)$. It was later extended to graphs of cliquewidth bounded by t, with $\mathcal{O}^\star((2t+1)^{\max\{n,h\}})$ time bound by Wahlström [35]. On the other hand, H needs not be sparse to admit efficient homomorphism testing: the family of cliques admits the $\mathcal{O}^\star(2^n)$ running time as shown by Björklund et al. [2]. As noted above, this generalizes to Kneser graphs $KG_{a,b}$, by the $\mathcal{O}^\star((b+1)^n)$-time algorithm of Nederlof. In this context, the natural question is whether the appearance of b in the base of the exponent is necessary, or is there an algorithm running in time $\mathcal{O}^\star(c^n)$ for some universal constant c independent of b.

Our contribution. We show that the algorithms for $(a{:}b)$-coloring mentioned above are essentially optimal under the Exponential Time Hypothesis. Specifically, we prove the following results:

[1] The $\mathcal{O}^\star(\cdot)$ notation hides factors polynomial in the input size.

▶ **Theorem 1.** *If there is an algorithm for* $(a{:}b)$-COLORING *that runs in time* $f(b) \cdot 2^{o(\log b)\cdot n}$, *for some computable function* $f(b)$, *then ETH fails. This holds even if the algorithm is only required to work on instances where* $a = \Theta(b^2 \log b)$.

▶ **Corollary 2.** *If there is an algorithm for* GRAPH HOMOMORPHISM *that runs in time* $f(h) \cdot 2^{o(\log\log h)\cdot n}$, *for some computable* $f(h)$, *then ETH fails. This holds even if the algorithm is only required to work on instances where* H *is a Kneser graph* $KG_{a,b}$ *with* $a = \Theta(b^2 \log b)$.

The bound for $(a{:}b)$-COLORING is tight, as the straightforward $\mathcal{O}^\star(2^n \cdot (b{+}1)^n) = 2^{\mathcal{O}(\log b)\cdot n}$ dynamic programming algorithm already shows. At first glance, one might have suspected that $(a{:}b)$-COLORING, as an interpolation between classical coloring and fractional coloring, both solvable in $2^{\mathcal{O}(n)}$ time [16], should be just as easy; Theorem 1 refutes this suspicion.

Corollary 2 in particular excludes any algorithm for testing homomorphisms into Kneser graphs with running time $2^{\mathcal{O}(n+h)}$. It cannot give a tight lower bound matching the result of Cygan et al. [9] for general homomorphisms, because $h = |V(KG_{a,b})| = \binom{a}{b}$ is not polynomial in b. On the other hand, it exhibits the first explicit family of graphs H for which the complexity of GRAPH HOMOMORPHISM increases with h.

In our proof, we first show a lower bound for the list variant of the problem, where every vertex is given a list of colors that can be assigned to it (see Section 2 for formal definitions). The list version is reduced to the standard version by introducing a large Kneser graph $KG_{a+b,b}$; we need a and b to be really small so that the size of this Kneser graph does not dwarf the size of the rest of the construction. However, this is not necessary for the list version, where we obtain lower bounds for a much wider range of functions $b(n)$.

▶ **Theorem 3.** *If there is an algorithm for* LIST $(a{:}b)$-COLORING *that runs in time* $2^{o(\log b)\cdot n}$, *then ETH fails. This holds even if the algorithm is only required to work on instances where* $a = \Theta(b^2 \log b)$ *and* $b = \Theta(b(n))$ *for an arbitrarily chosen polynomial-time computable function* $b(n)$ *such that* $b(n) \in \omega(1)$ *and* $b(n) = \mathcal{O}(n/\log n)$.

The crucial ingredient in the proof of Theorem 3 is the usage of *d-detecting matrices* introduced by Lindström [27]. We choose to work with their combinatorial formulation, hence we shall talk about *d-detecting families*. Suppose we are given some universe U and there is an unknown function $f\colon U \to \{0, 1, \ldots, d-1\}$, for some fixed positive integer d. One may think of U as consisting of coins of unknown weights that are integers between 0 and $d-1$. We would like to learn f (the weight of every coin) by asking a small number of queries of the following form: for a subset $X \subseteq U$, what is $\sum_{e\in X} f(e)$ (the total weight of coins in X)? A set of queries sufficient for determining all the values of an arbitrary f is called a *d-detecting family*. Of course f can be learned by asking $|U|$ questions about single coins, but it turns out that significantly fewer questions are needed: there is a d-detecting family of size $\mathcal{O}(|U|/\log|U|)$, for every fixed d [27]. The logarithmic factor in the denominator will be crucial for deriving our lower bound.

Let us now sketch how d-detecting families are used in the proof of Theorem 3. Given an instance φ of 3-SAT with n variables and $\mathcal{O}(n)$ clauses, and a number $b \leq n/\log n$, we will construct an instance G of LIST $(a{:}b)$-COLORING for some a. This instance will have a positive answer if and only if φ is satisfiable, and the constructed graph G will have $\mathcal{O}(n/\log b)$ vertices. It can be easily seen that this will yield the promised lower bound.

Partition the clause set C of φ into groups $C_1, C_2, \ldots, C_p$, each of size roughly b; thus $p = \mathcal{O}(n/b)$. Similarly, partition the variable set V of φ into groups $V_1, \ldots, V_q$, each of size roughly $\log_2 b$; thus $q = \mathcal{O}(n/\log b)$. In the output instance we create one vertex per each variable group—hence we have $\mathcal{O}(n/\log b)$ such vertices—and one block of vertices per each

clause group, whose size will be determined in a moment. Our construction ensures that the set of colors assigned to a vertex created for a variable group misses one color from some subset of b colors. The choice of the missing color corresponds to one of $2^{\log_2 b} = b$ possible boolean assignments to the variables of the group.

Take any vertex u from a block of vertices created for some clause group C_j. We make it adjacent to vertices constructed for precisely those variable groups V_i, for which there is some variable in V_i that occurs in some clause of C_j. This way, u can only take a subset of the above missing colors corresponding to the chosen assignment on variables relevant to C_j. By carefully selecting the list of u, and some additional technical gadgeteering, we can express a constraint of the following form: the total number of satisfied literals in some subset of clauses of C_j is exactly some number. Thus, we could verify that every clause of C_j is satisfied by creating a block of $|C_j|$ vertices, each checking one clause. However, the whole graph output by the reduction would then have $\mathcal{O}(n)$ vertices, and we would not obtain any non-trivial lower bound. Instead, we create one vertex per each question in a d-detecting family on the universe $U = C_j$, which has size $\mathcal{O}(|C_j|/\log|C_j|) = \mathcal{O}(|C_j|/\log b)$. Then, the total number of vertices in the constructed graph will be $\mathcal{O}(n/\log b)$, as intended.

Finally, we observe that from our main result one can infer a lower bound for the complexity of the (r,k)-Monomial Testing problem. Recall that in this problem we are given an arithmetic circuit that evaluates a homogenous polynomial $P(x_1, x_2, \ldots, x_n)$ over some field $\mathbb{F}$; here, a polynomial is homogenous if all its monomials have the same total degree k. The task is to verify whether P has some monomial in which every variable has individual degree not larger than r, for a given parameter r. Abasi et al. [1] gave a randomized algorithm solving this problem in time $\mathcal{O}^\star(2^{\mathcal{O}(k\cdot\frac{\log r}{r})})$, where k is the degree of the polynomial, assuming that $\mathbb{F} = \mathrm{GF}(p)$ for a prime $p \leq 2r^2 + 2r$. This algorithm was later derandomized by Gabizon et al. [14] within the same running time, but under the assumption that the circuit is *non-cancelling*: it has only input, addition, and multiplication gates. Abasi et al. [1] and Gabizon et al. [14] gave a number of applications of low-degree monomial detection to concrete problems. For instance, r-Simple k-Path, the problem of finding a walk of length k that visits every vertex at most r times, can be solved in time $\mathcal{O}^\star(2^{\mathcal{O}(k\cdot\frac{\log r}{r})})$. However, for r-Simple k-Path, as well as other problems that can be tackled using this technique, the best known lower bounds under ETH exclude only algorithms with running time $\mathcal{O}^\star(2^{o(\frac{k}{r})})$. Whether the $\log r$ factor in the exponent is necessary was left open by Abasi et al. and Gabizon et al.

We observe that the List $(a{:}b)$-coloring problem can be reduced to (r,k)-Monomial Testing over the field $\mathrm{GF}(2)$ in such a way that an $\mathcal{O}^\star(2^{k\cdot o(\frac{\log r}{r})})$-time algorithm for the latter would imply a $2^{o(\log b)\cdot n}$-time algorithm for the former, which would contradict ETH. Thus, we show that the known algorithms for (r,k)-Monomial Testing most probably cannot be sped up in general; nevertheless, the question of lower bounds for specific applications remains open. However, going through List $(a{:}b)$-coloring to establish a lower bound for (r,k)-Monomial Testing is actually quite a detour, because the latter problem has a much larger expressive power. Therefore, we also give a more straightforward reduction that starts from a convenient form of Subset Sum; this reduction also proves the lower bound for a wider range of r, expressed as a function of k.

Outline. In Section 2 we set up the notation as well as recall definitions and well-known facts. We also discuss d-detecting families, the main combinatorial tool used in our reduction. In Section 3 we prove the lower bound for the list version of the problem, i.e., Theorem 3, and sketch the few steps needed for the standard version, thereby proving Theorem 1. Section 4 is

devoted to deriving lower bounds for low-degree monomial testing. Due to space constraints, proofs of statements marked with (♠) are deferred to the full version [3] of this paper.

2 Preliminaries

Notation. We use standard graph notation, see e.g. [10, 11]. All graphs we consider in this paper are simple and undirected. For an integer k, we denote $[k] = \{0, \ldots, k-1\}$. By $\uplus$ we denote the disjoint union, i.e., by $A \uplus B$ we mean $A \cup B$ with the indication that A and B are disjoint. If I and J are instances of decision problems P and R, respectively, then we say that I and J are *equivalent* if either both I and J are YES-instances, or both are NO-instances of the respective problems.

Exponential-Time Hypothesis. The Exponential Time Hypothesis (ETH) of Impagliazzo et al. [22] states that there exists a constant $c > 0$, such that there is no algorithm solving 3-SAT in time $\mathcal{O}^\star(2^{cn})$. During the recent years, ETH became the central conjecture used for proving tight bounds on the complexity of various problems. One of the most important results connected to ETH is the *Sparsification Lemma* [23], which essentially gives a reduction from an arbitrary instance of k-SAT to an instance where the number of clauses is linear in the number of variables. The following well-known corollary can be derived by combining ETH with the Sparsification Lemma.

▶ **Theorem 4** (see e.g. Theorem 14.4 in [10]). *Unless ETH fails, there is no algorithm for* 3-SAT *that runs in time* $2^{o(n+m)}$*, on formulas with* n *variables and* m *clauses.*

We need the following regularization result of Tovey [34]. Following Tovey, by (3,4)-SAT we call the variant of 3-SAT where each clause of the input formula contains exactly 3 different variables, and each variable occurs in at most 4 clauses.

▶ **Lemma 5** ([34]). *Given a* 3-SAT *formula* φ *with* n *variables and* m *clauses one can transform it in polynomial time into an equivalent* (3,4)-SAT *instance* φ' *with* $\mathcal{O}(n+m)$ *variables and clauses.*

▶ **Corollary 6.** *Unless ETH fails, there is no algorithm for* (3,4)-SAT *that runs in time* $2^{o(n)}$*, where* n *denotes the number of variables of the input formula.*

List and nonuniform list (a:b)-coloring. For integers a, b and a graph G with a function $L\colon V(G) \to 2^{[a]}$ (assigning a list of colors to every vertex), an *L-(a:b)-coloring* of G is an assignment of exactly b colors from $L(v)$ to each vertex $v \in V(G)$, such that adjacent vertices get disjoint color sets. The List (a:b)-coloring problem asks, given (G, L), whether an L-(a:b)-coloring of G exists.

As an intermediary step of our reduction, we use the following generalization of list colorings where the number of demanded colors varies with every vertex. For integers a, b, a graph G with a function $L\colon V(G) \to 2^{[a]}$ and a *demand function* $\beta\colon V(G) \to \{1, \ldots, b\}$, an *$L$-($a$:$\beta$)-coloring* of G is an assignment of exactly $\beta(v)$ colors from $L(v)$ to each vertex $v \in V(G)$, such that adjacent vertices get disjoint color sets. Nonuniform List (a:b)-coloring is then the problem in which given (G, L, β) we ask if an L-(a:β)-coloring of G exists.

***d*-detecting families.** In our reductions the following notion plays a crucial role.

▶ **Definition 7.** A *d-detecting family* for a finite set U is a family $\mathcal{F} \subseteq 2^U$ of subsets of U such that for every two functions $f, g : U \to \{0, \ldots, d-1\}$, $f \neq g$, there is a set S in the family such that $\sum_{x \in S} f(x) \neq \sum_{x \in S} g(x)$.

A deterministic construction of sublinear, d-detecting families was given by Lindström [27], together with a proof that even the constant factor 2 in the family size cannot be improved.

▶ **Theorem 8** ([27]). *For every constant $d \in \mathbb{N}$ and finite set U, there is a d-detecting family $\mathcal{F}$ on U of size $\frac{2|U|}{\log_d |U|} \cdot (1 + o(1))$. Furthermore, $\mathcal{F}$ can be constructed in* $\mathrm{poly}(|U|)$ *time.*

Other constructions, generalizations, and discussion of similar results can be found in Grebinski and Kucherov [15], and in Bshouty [4]. Note that the expression $\sum_{x \in S} f(x)$ is just the product of f as a vector in $[d]^{|U|}$ with the characteristic vector of S. Hence, instead of subset families, Lindström speaks of *detecting vectors*, while later works see them as *detecting matrices*, that is, $(0,1)$-matrices with these vectors as rows (which define an injection on $[d]^{|U|}$ despite having few rows). Similar definitions appear in the study of query complexity, e.g., as in the popular Mastermind game [7].

3 Hardness of List $(a{:}b)$-coloring

In this section we show our main technical contribution: an ETH-based lower bound for List $(a{:}b)$-coloring. We begin with key part: reducing an n-variable instance 3-SAT to an instance of Nonuniform List $(a{:}b)$-coloring with only $\mathcal{O}(\frac{n}{\log b})$ vertices. Next, it is rather easy to reduce Nonuniform List $(a{:}b)$-coloring to List $(a{:}b)$-coloring.

3.1 The nonuniform case

We prove the following theorem through the remaining part of this section.

▶ **Theorem 9.** *For any instance ϕ of* (3,4)-SAT *with n variables and any integer $2 \leq b \leq n/\log_2 n$, there is an equivalent instance (G, β, L) of* Nonuniform List $(a{:}2b)$-coloring *such that $a = \mathcal{O}(b^2 \log b)$, $|V(G)| = \mathcal{O}(\frac{n}{\log b})$ and G is 3-colorable. Moreover, the instance (G, β, L) and the 3-coloring of G can be constructed in* $\mathrm{poly}(n)$ *time.*

Consider an instance ϕ of 3-SAT where each variable appears in at most four clauses. Let V be the set of its variables and C be the set of its clauses. Note that $\frac{1}{3}|V| \leq |C| \leq \frac{4}{3}|V|$. Let $a = 12b^2 \cdot \lfloor \log_2 b \rfloor$. We shall construct, for some integers $n_V = \mathcal{O}(|V|/\log b)$ and $n_C = \mathcal{O}(|C|/b)$:

- a partition $V = V_1 \uplus \ldots \uplus V_{n_V}$ of variables into groups of size at most $\lfloor \log_2 b \rfloor$,
- a partition $C = C_1 \uplus \ldots \uplus C_{n_C}$ of clauses into groups of size at most b,
- a function $\sigma \colon \{1, \ldots, n_V\} \to [12 \cdot b \cdot \lfloor \log_2 b \rfloor]$,

such that the following condition holds:

> For any $j = 1, \ldots, n_C$, the variables occurring in clauses of C_j are all different and they all belong to pairwise different variable groups. Moreover, the indices of these groups are mapped to pairwise different values by σ. (✠)

In other words, any two literals of clauses in C_j have different variables, and if they belong to V_i and $V_{i'}$ respectively, then $\sigma(i) \neq \sigma(i')$.

▶ **Lemma 10** (♠). *Partitions $V = V_1 \uplus \ldots \uplus V_{n_V}$, $C = C_1 \uplus \ldots \uplus C_{n_C}$ and a function σ satisfying* (✠) *can be found in time $\mathcal{O}(n)$.*

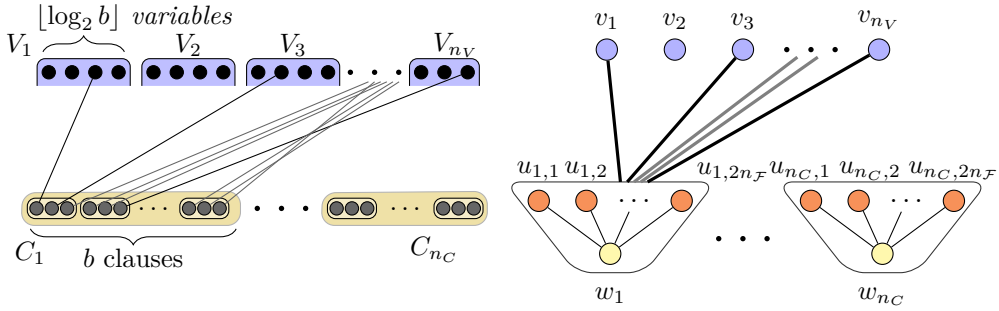


Figure 2 *(left)* The groups of variables and clauses of the formula; literals in C_1 are joined with their variables. Since no variable of V_2 occurs in C_1, we have $2 \notin I_1$ – this may allow us to make $\sigma(2)$ the same number as $\sigma(3)$, say, reducing the total number a of colors needed. *(right)* The constructed graph; thick lines represent edges to all vertices corresponding to C_1.

For every $1 \leq i \leq n_V$, the set V_i of variables admits $2^{|V_i|} \leq b$ different assignments. We will therefore say that each assignment on V_i is *given* by an integer $x \in [b]$, for example by interpreting the first $|V_i|$ bits of the binary representation of x as truth values for variables in V_i. Note that when $|V_i| < \log_2 b$, different integers from $[b]$ may give the same assignment on V_i.

For $1 \leq j \leq n_C$, let $I_j \subseteq \{1, \ldots, n_V\}$ be the set of indices of variable groups that contain some variable occurring in the clauses of C_j. Since every clause contains exactly three literals, property (✠) means that $|I_j| = 3|C_j|$ and that σ is injective over each I_j. See Fig. 2.

For $1 \leq j \leq n_C$, let $\{C_{j,1}, \ldots, C_{j,n_\mathcal{F}}\}$ be a 4-detecting family of subsets of C_j, for some $n_\mathcal{F} = \mathcal{O}(\frac{b}{\log b})$ (we can assume $n_\mathcal{F}$ does not depend on j by adding arbitrary sets when $|C_j| < b$). For every $1 \leq k \leq n_\mathcal{F}$, let $C_{j,n_\mathcal{F}+k} = C_j \setminus C_{j,k}$.

We are now ready to build the graph G, the demand function $\beta : V(G) \to \{1, \ldots, 2b\}$, and the list assignment L as follows.

1. For $1 \leq i \leq n_V$, create a vertex v_i with $\beta(v_i) = b - 1$ and $L(v_i) = \{b \cdot \sigma(i) + x \mid x \in [b]\}$.
2. For $1 \leq j \leq n_C$ and $1 \leq k \leq 2n_\mathcal{F}$, create a vertex $u_{j,k}$ adjacent to each v_i for $i \in I_j$. Let $\beta(u_{j,k}) = |C_{j,k}|$ and

$$L(u_{j,k}) = \{b \cdot \sigma(i) + x \;\mid\; 1 \leq i \leq n_V, x \in [2^{|V_i|}] \text{ such that } x \text{ gives an assignment of } V_i \text{ that satisfies some clause of } C_{j,k}\}.$$

3. For $1 \leq j \leq n_C$, create a vertex w_j, adjacent to each v_i for $i \in I_j$ and to each $u_{j,k}$ $(1 \leq k \leq 2n_\mathcal{F})$. Let $\beta(w_j) = 2|C_j|$ and $L(w_j) = \bigcup_{i \in I_j} \{b \cdot \sigma(i) + x \mid x \in [b]\}$.

Before giving a detailed proof of the correctness, let us describe the reduction in intuitive terms. Note that vertices of type v_i get all but one color from their list; this missing color, say $b \cdot \sigma(i) + x_i$, for some $x_i \in [b]$, defines an assignment on V_i. For every $j = 1, \ldots, n_C$ the goal of the gadget consisting of w_j and vertices $u_{j,k}$ is to express the constraint that every clause in C_j has a literal satisfied by this assignment. Since $w_j, u_{j,k}$ are adjacent to all vertices in $\{v_i \mid i \in I_j\}$, they may only use the missing colors (of the form $b \cdot \sigma(i) + x_i$, where $i \in I_j$). Since $|I_j| = 3|C_j|$, there are $3|C_j|$ such colors and $2|C_j|$ of them go to w_j. This leaves exactly $|C_j|$ colors for vertices of type $u_{j,k}$, corresponding to a choice of $|C_j|$ satisfied literals from the $3|C_j|$ literals in clauses of C_j. The lists and demands for vertices $u_{j,k}$ guarantee that

exactly $|C_{j,k}|$ chosen satisfied literals occur in clauses of $C_{j,k}$. The properties of 4-detecting families will ensure that every clause has exactly one chosen, satisfied literal, and hence at least one satisfied literal. We proceed with formal proofs.

▶ **Lemma 11.** *If ϕ is satisfiable then G is L-$(a{:}\beta)$-colorable.*

Proof. Consider a satisfying assignment η for ϕ. For $1 \leq i \leq n_V$, let $x_i \in [2^{|V_i|}]$ be an integer giving the same assignment on V_i as η. For every clause c of ϕ, choose one literal satisfied by η in it, and let i_c be index of the group V_{i_c} containing the literal's variable. Let $\alpha : V(G) \to \binom{[a]}{\leq 2b}$ be the L-$(a{:}\beta)$-coloring of G defined as follows, for $1 \leq i \leq n_V$, $1 \leq j \leq n_C$, $1 \leq k \leq 2n_{\mathcal{F}}$:

- $\alpha(v_i) = L(v_i) \setminus \{b \cdot \sigma(i) + x_i\}$
- $\alpha(u_{j,k}) = \{b \cdot \sigma(i_c) + x_{i_c} \mid c \in C_{j,k}\}$
- $\alpha(w_j) = \{b \cdot \sigma(i) + x_i \mid i \in I_j \setminus \{i_c \mid c \in C_j\}\}$.

Let us first check that every vertex v gets colors from its list $L(v)$ only. This is immediate for vertices v_i and w_j, while for $u_{j,k}$ it follows from the fact that x_{i_c} gives a partial assignment to V_i that satisfies some clause of $C_{j,k}$.

Now let us check that for every vertex v, the coloring α assigns exactly $\beta(v)$ colors to v. For $\alpha(v_i)$ this follows from the fact that $|L(v_i)| = b$ and $0 \leq x_i < 2^{|V_i|} \leq b$. Since by property (✠), σ is injective on I_j, and thus on $\{i_c \mid c \in C_{j,k}\} \subseteq I_j$, we have $|\alpha(u_{j,k})| = |C_{j,k}| = b(u_{j,k})$. Similarly, since σ is injective on I_j and $|I_j \setminus \{i_c \mid c \in C_j\}| = 3|C_j| - |C_j| = 2|C_j|$, we get $|\alpha(w_j)| = 2|C_j| = \beta(w_j)$.

It remains to argue that the sets assigned to any two adjacent vertices are disjoint. There are three types of edges in the graph, namely $v_i u_{j,k}$, $v_i w_j$, and $w_j u_{j,k}$. The disjointness of $\alpha(w_j)$ and $\alpha(u_{j,k})$ is immediate from the definition of α, since $C_{j,k} \subseteq C_j$. Fix $j = 1, \ldots, n_C$. Since σ is injective on I_j, for any two different $i, i' \in I_j$, we have $b \cdot \sigma(i) + x_i \notin L(v_{i'})$. Hence,

$$\bigcup_{i \in I_j} \alpha(v_i) = \{b \cdot \sigma(i) + x \mid i \in I_j \text{ and } x \in [b]\} \setminus \{b \cdot \sigma(i) + x_i \mid i \in I_j\}.$$

Since $\alpha(u_{j,k}), \alpha(w_j) \subseteq \{b \cdot \sigma(i) + x_i \mid i \in I_j\}$, it follows that edges of types $v_i u_{j,k}$ and $v_i w_j$ received disjoint sets of colors on their endpoints, concluding the proof. ◀

▶ **Lemma 12.** *If G is L-$(a{:}\beta)$-colorable then ϕ is satisfiable.*

Proof. Assume that G is L-$(a{:}\beta)$-colorable, and let α be the corresponding coloring.

For $1 \leq i \leq n_V$, we have $|L(v_i)| = b$ and $|\alpha(v_i)| = b - 1$, so v_i misses exactly one color from its list. Let $b \cdot \sigma(i) + x_i$, for some $x_i \in [b]$, be the missing color. We want to argue that the assignment x for ϕ given by x_i on each V_i satisfies ϕ.

Consider any clause group C_j, for $1 \leq j \leq n_C$. Every vertex in $\{w_j\} \cup \{u_{j,k} \mid 1 \leq k \leq 2n_{\mathcal{F}}\}$ contains $\{v_i \mid i \in I_j\}$ in its neighborhood. Therefore, the sets $\alpha(u_{j,k})$ and $\alpha(w_j)$ are disjoint from $\bigcup_{i \in I_j} \alpha(v_i)$. Since $L(u_{j,k}), L(w_j) \subseteq \{b \cdot \sigma(i) + x' \mid i \in I_j, x' \in [b]\}$, we get that $\alpha(u_{j,k})$ and $\alpha(w_j)$ are contained in the set of missing colors $\{b \cdot \sigma(i) + x_i \mid i \in I_j\}$ (corresponding to the chosen assignment). By property (✠), this set has exactly $|I_j| = 3|C_j|$ different colors. Of these, exactly $2|C_j|$ are contained in $\alpha(w_j)$. Let the remaining $|C_j|$ colors be $\{b \cdot \sigma(i) + x_i \mid i \in J_j\}$, for some subset $J_j \subseteq I_j$ of $|C_j|$ indices.

Since $\alpha(u_{j,k})$ is disjoint from $\alpha(w_j)$, we have $\alpha(u_{j,k}) \subseteq \{b \cdot \sigma(i) + x_i \mid i \in J_j\}$ for all k. By definition of I_j, for every $i \in J_j \subseteq I_j$ there is a variable in V_i that appears in some clause of C_j. By property (✠), it can only occur in one such clause, so let l_i be the literal in the clause of C_j where it appears. For every color $b \cdot \sigma(i) + x_i \in \alpha(u_{j,k})$, by definition of the lists

for $u_{j,k}$ we know that x_i gives a partial assignment to V_i that satisfies some clause of $C_{j,k}$. This means x_i makes the literal l_i true and l_i occurs in a clause of $C_{j,k}$. Therefore, for each k, at least $|\alpha(u_{j,k})| = |C_{j,k}|$ literals from the set $\{l_i \mid i \in J_j\}$ occur in clauses of $C_{j,k}$ and are made true by the assignment x.

Let $f : C_j \to \{0,1,2,3\}$ be the function assigning to each clause $c \in C_j$ the number of literals of c in $\{l_i \mid i \in J_j\}$. By the above, $\sum_{c \in C_{j,k}} f(c) \geq |C_{j,k}|$ for $1 \leq k \leq 2n_{\mathcal{F}}$. Since each literal in $\{l_i \mid i \in J_j\}$ belongs to some clause of C_j, we have $\sum_{c \in C_j} f(c) = |J_j| = |C_j|$. Then,

$$\sum_{c \in C_{j,k}} f(c) = \sum_{c \in C_j} f(c) - \sum_{c \in C_{j,n_{\mathcal{F}}+k}} f(c) \leq |C_j| - |C_{j,n_{\mathcal{F}}+k}| = |C_{j,k}|.$$

Hence $\sum_{c \in C_{j,k}} f(c) = |C_{j,k}|$ for $1 \leq k \leq 2n_{\mathcal{F}}$. Let $g : C_j \to \{0,1,2,3\}$ be the constant function $g \equiv 1$. Note that

$$\sum_{c \in C_{j,k}} g(c) = |C_{j,k}| = \sum_{c \in C_{j,k}} f(c).$$

Since $\{C_{j,1}, \ldots, C_{j,n_{\mathcal{F}}}\}$ is a 4-detecting family, this implies that $f \equiv 1$. Thus, for every clause c of C_j we have $f(c) = 1$, meaning that there is a literal from the set $\{l_i \mid i \in J_j\}$ in this clause. All these literals are made positive by the assignment η, therefore all clauses of C_j are satisfied. Since $j = 1, \ldots, n_C$ was arbitrary, this concludes the proof that η is a satisfying assignment for ϕ. ◀

The construction can clearly be made in polynomial time and the total number of vertices is $n_V + n_C \cdot \mathcal{O}(\frac{b}{\log b}) + n_C = \mathcal{O}(\frac{n}{\log b})$. Moreover, we get a proper 3-coloring of G, by coloring vertices of the type v_i by color 1, vertices of the type $u_{j,k}$ by color 2, and vertices of the type w_j by color 3. By Lemmas 11 and 12, this concludes the proof of Theorem 9.

3.2 The uniform case

In this section we reduce the nonuniform case to the uniform one, and state the resulting lower bound on the complexity of List $(a{:}b)$-coloring.

▶ **Lemma 13.** *For any instance $I = (G, \beta, L)$ of* Nonuniform List $(a{:}b)$-coloring *where the graph G is t-colorable, there is an equivalent instance (G, L') of* List $((a+tb){:}b)$-coloring. *Moreover, given a t-coloring of G the instance (G, L') can be constructed in time polynomial in $|I| + b$.*

Proof. Let $c\colon V(G) \to [t]$ be a t-coloring of G. For every vertex v, define a set of filling colors $F(v) = \{a + c(v)b + i \,:\, i = 0, \ldots, b - |\beta(v)| - 1\}$ and put $L'(v) = L(v) \cup F(v)$.

Let $\alpha\colon V(G) \to 2^{[a]}$ be an L-$(a{:}\beta)$-coloring of G. We define a coloring $\alpha'\colon V(G) \to 2^{[a+tb]}$ by setting $\alpha'(v) = \alpha(v) \cup F(v)$ for every vertex $v \in V(G)$. Observe that $\alpha'(v) \subseteq L'(v)$ and $|\alpha'(v)| = |\alpha(v)| + (b - |\beta(v)|) = b$. Since α was a proper L-$(a{:}\beta)$-coloring, adjacent vertices can only share the filling colors. However, the lists of adjacent vertices have disjoint subsets of filling colors, since these vertices are colored differently by c. It follows that α' is an L'-$(a{:}b)$-coloring of G.

Conversely, let $\alpha'\colon V(G) \to 2^{[a+tb]}$ be an L'-$(a{:}b)$-coloring of G. For every vertex v, we have $|\alpha'(v) \cap [a]| = b - |\alpha'(v) \cap F(v)| \geq b - (b - |\beta(v)|) = |\beta(v)|$. Define $\alpha(v)$ to be any cardinality $\beta(v)$ subset of $\alpha'(v) \cap [a]$. It is immediate that α is an L-$(a{:}\beta)$-coloring of G. ◀

We are now ready to prove one of our main results.

▶ **Theorem 3.** *If there is an algorithm for* List $(a{:}b)$-coloring *that runs in time* $2^{o(\log b)\cdot n}$, *then ETH fails. This holds even if the algorithm is only required to work on instances where* $a = \Theta(b^2 \log b)$ *and* $b = \Theta(b(n))$ *for an arbitrarily chosen polynomial-time computable function* $b(n)$ *such that* $b(n) \in \omega(1)$ *and* $b(n) = \mathcal{O}(n/\log n)$.

Proof. Let $b(n)$ be a function as in the statement. We can assume w.l.o.g. that $2 \leq b(n) \leq n/\log_2 n$ (otherwise, replace $b(n)$ with $b'(n) = 2 + \lfloor b(n)/c \rfloor$ in the reasoning below, for c a big enough constant; clearly $b'(n) = \Theta(b(n))$). Fix a function $f(b) = o(\log b)$ and assume there is an algorithm $\mathcal{A}$ for List $(a{:}b)$-coloring that runs in time $2^{f(b)\cdot n}$, whenever $b = \Theta(b(n))$. Consider an instance of (3,4)-SAT with n variables. Let $b = b(n)$. By Theorem 9 in poly(n) time we get an equivalent instance (G, β, L) of Nonuniform List $(a{:}(2b))$-coloring such that $a = \Theta(b^2 \log b)$, $|V(G)| = \mathcal{O}(\frac{n}{\log b})$, and a 3-coloring of G. Next, by Lemma 13 in poly(n) time we get an equivalent instance (G, L') of List $((a + 6b){:}(2b))$-coloring. Finally, we solve the instance (G, L') using algorithm $\mathcal{A}$. Since $b(n) = \omega(1)$, we have $f(b(n)) = o(\log(b(n)))$, and $\mathcal{A}$ runs in time $2^{o(\log b(n))\cdot|V(G)|}$. Thus, we solved (3,4)-SAT in time $2^{o(\log b(n))\cdot|V(G)|} = 2^{o(\log b(n))\cdot\frac{n}{\log b(n)}} = 2^{o(n)}$. By Corollary 6, this contradicts ETH. ◀

Finally, we reduce List $(a{:}b)$-coloring to $(a{:}b)$-coloring. This is done by increasing the number of colors by b, adding a Kneser graph $KG_{a+b,b}$ (which can be colored essentially only by assigning each b-set of colors to its corresponding vertex), and replacing the lists by edges to appropriate vertices of the Kneser graph.

▶ **Lemma 14 (♠).** *Given an instance of* List $(a{:}b)$-coloring *with* n *vertices, an equivalent instance of* $(a + b : b)$-coloring *with* $n + \binom{a+b}{b}$ *vertices can be computed in* poly$(n, \binom{a+b}{b})$*-time.*

For $b \in o(\frac{\log n}{\log\log n})$, $a = \mathcal{O}(b^2 \log b)$, we show that $\binom{a+b}{b} = o(n)$, allowing us to compose our reductions with Lemma 14. An analysis similar to the one in Theorem 3 then concludes the proof of Theorem 1 and Corollary 2; we refer to the full version [3] for details.

4 Low-degree testing

In this section we derive lower bounds for (r, k)-Monomial Testing. In this problem, we are given an arithmetic circuit C over some field $\mathbb{F}$ (with input, constant, addition, and multiplication gates). One gate is designated to be the output gate, and it computes some polynomial P of the variables $x_1, x_2, \ldots, x_n$ that appear in the input gates. We assume that P is a homogenous polynomial of degree k, i.e., all its monomials have total degree k. The task is to verify whether P contains an r-monomial, i.e., a monomial in which every variable has its individual degree bounded by r, for a given $r \leq k$. Abasi et al. [1] gave a very fast randomized algorithm for (r, k)-Monomial Testing.

▶ **Theorem 15** (Abasi et al. [1]). *Fix integers* r, k *with* $2 \leq r \leq k$. *Let* $p \leq 2r^2 + 2r$ *be a prime, and let* $g \in \mathrm{GF}(p)[x_1, \ldots, x_n]$ *be a homogenous polynomial of degree* k, *computable by a circuit* C. *There is a randomized algorithm running in time* $\mathcal{O}(r^{2k/r}|C|(rn)^{\mathcal{O}(1)})$ *which:*

- *with probability at least* $1/2$ *answers YES when* g *contains an* r*-monomial,*
- *always answers NO when* g *contains no* r*-monomial.*

This result was later derandomized by Gabizon et al. [14] under the assumption that the circuit is *non-cancelling*, that is, it contains only input, addition, and multiplication gates. Many concrete problems like r-Simple k-Path can be reduced to (r, k)-Monomial

Testing by encoding the set of candidate objects as monomials of some large polynomial, so that "good" objects correspond to monomials with low individual degrees.

As we show in the full version [3] of this paper, this is also the case for List $(a{:}b)$-coloring. Namely, for an instance (G, L) of List $(a{:}b)$-coloring we can construct a homogeneous polynomial p_G of degree $k = 2bn$ with $n(a+1)$ variables, together with a circuit of size $2^n \operatorname{poly}(a, n)$ evaluating it, such that G admits a list $(a{:}b)$-coloring iff p_G contains a b-monomial. Using Theorem 15 with $r = b$, we get yet another polynomial-space algorithm for List $(a{:}b)$-coloring, running in time $\mathcal{O}(b^{\mathcal{O}(n)} \operatorname{poly}(n))$. Similarly, if the running time in Theorem 15 was improved to $2^{o(\log r/r)\cdot k} \cdot |C| \operatorname{poly}(r, n)$, then we would get an algorithm for List $(a{:}b)$-coloring in time $\mathcal{O}(2^{o(\log b)n} \operatorname{poly}(n))$, which contradicts ETH by Theorem 3. However, a careful examination shows that this chain of reductions would only yield instances of (r, k)-Monomial Testing with $r = \mathcal{O}(\sqrt{k/\log k})$. Hence, this does not exclude the existence of a fast algorithm that works only for large r. Below we show a more direct reduction, which excludes fast algorithms for a wider spectrum of pairs (r, k).

In the Carry-Less Subset Sum problem, we are given $n+1$ numbers $s, a_1, \ldots, a_n$, each represented as n decimal digits. For any number x, the j-th decimal digit of x is denoted by $x^{(j)}$. It is assumed that $\sum_{i=1}^{n} a_i^{(j)} < 10$, for every $j = 1, \ldots, n$. The goal is to verify whether there is a sequence of indices $1 \leq i_1 < \ldots < i_k \leq n$ such that $\sum_{q=1}^{k} a_{i_q} = s$. Note that by the small sum assumption, this is equivalent to the statement that $\sum_{q=1}^{k} a_{i_q}^{(j)} = s^{(j)}$, for every $j = 1, \ldots, n$. The standard NP-hardness reduction from 3-SAT to Subset Sum (see e.g. [8]) in fact gives instances of Carry-Less Subset Sum of linear size, yielding the following.

▶ **Lemma 16 (♠).** *Unless ETH fails, the* Carry-Less Subset Sum *problem cannot be solved in* $2^{o(n)}$ *time.*

We proceed with a sketch of the reduction from Carry-Less Subset Sum to (r, k)-Monomial Testing; details can be found in the full version [3] of this paper. Let us choose a parameter $t \in \{1, \ldots, n\}$. Assume w.l.o.g. that t divides n (otherwise, add zeroes at the end of every input number). Let $q = n/t$. For an n-digit decimal number x, for every $j = 1, \ldots t$, let $x^{[j]}$ denote the q-digit number given by the j-th block of q digits in x, i.e., $x^{[j]} = (x^{(jq-1)} \cdots x^{((j-1)q)})_{10}$.

Let $r = 10^q - 1$. Define the following polynomial over GF(2):

$$q_S = \prod_{i=1}^{n} \left(y_i + z_i \cdot \prod_{j=1}^{t} x_j^{a_i^{[j]}} \right) \cdot \prod_{j=1}^{t} x_j^{r - s^{[j]}} = \sum_{S \subseteq \{1,\ldots,n\}} \prod_{j=1}^{t} x_j^{\sum_{i \in S} a_i^{[j]} + r - s^{[j]}} \prod_{i \notin S} y_i \prod_{i \in S} z_i.$$

Let p_S denote the polynomial obtained from q_S by filtering out all the monomials of degree different than $k = tr + n$. The first expression defining q_S gives a circuit of size $\mathcal{O}(nt)$, and thus with a standard construction, we show p_S can be evaluated by a circuit of size $\mathcal{O}(nt^2 r + n^2 t)$. It is relatively easy to see that by construction, $(s, a_1, \ldots, a_n)$ is a YES-instance of Carry-Less Subset Sum iff q_S contains the monomial $\prod_{j=1}^{t} x_j^r \prod_{i \notin S} y_i \prod_{i \in S} z_i$, for some $S \subseteq \{1, \ldots, n\}$ (the variables y_i and z_i guaranteeing that no pair of monomials cancels out). This in turn holds iff p_S contains an r-monomial (with exactly n variables y_i and z_i, and hence degree exactly r for each x_i variable).

With this reduction, we obtain our main lower bound for (r, k)-Monomial Testing. We state it in the most general, but technical form, and derive an exemplary corollary below.

▶ **Theorem 17 (♠).** *If there is an algorithm solving* (r, k)-Monomial Testing *in time* $2^{o(k \log r/r)} |C|^{\mathcal{O}(1)}$*, then ETH fails. The statement remains true even if the algorithm works*

only for instances where $r = 2^{\Theta(n/t(n))}$ *and* $k = t(n)2^{\Theta(n/t(n))}$*, for an arbitrarily chosen function* $t : \mathbb{N} \to \mathbb{N}$ *computable in* $2^{o(n)}$ *time, such that* $t(n) = \omega(1)$ *and* $t(n) \leq n$ *for every* n*.*

▶ **Theorem 18 (♠).** *Let* $\sigma \in [0, 1)$*. Then, unless ETH fails, there is no algorithm for* (r, k)-Monomial Testing *that solves instances with* $r = \Theta(k^\sigma)$ *in time* $2^{o(k \cdot \frac{\log r}{r})} \cdot |C|^{\mathcal{O}(1)}$*.*

In particular, no algorithm solves (r, k)-Monomial Testing in time $2^{o(\frac{\log r}{r}) \cdot k} \cdot |C|^{\mathcal{O}(1)}$ for all input values r, unless ETH fails.

Acknowledgements. The authors thank Andreas Björklund and Matthias Mnich for sharing the problem considered in this paper.

References

1 Hasan Abasi, Nader H. Bshouty, Ariel Gabizon, and Elad Haramaty. On r-simple k-path. In *MFCS 2015*, volume 8635 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2014.

2 Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. Set partitioning via inclusion-exclusion. *SIAM J. Comput.*, 39(2):546–563, 2009.

3 Marthe Bonamy, Lukasz Kowalik, Michal Pilipczuk, Arkadiusz Socala, and Marcin Wrochna. Tight lower bounds for the complexity of multicoloring. *CoRR*, abs/1607.03432, 2016. URL: `http://arxiv.org/abs/1607.03432`.

4 Nader H. Bshouty. Optimal algorithms for the coin weighing problem with a spring scale. In *COLT 2009*, 2009.

5 Marie G. Christ, Lene M. Favrholdt, and Kim S. Larsen. Online multi-coloring with advice. In *WAOA 2014*, volume 8952 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2014.

6 V. Chvátal, M. R. Garey, and D. S. Johnson. Two results concerning multicoloring. In *Algorithmic Aspects of Combinatorics*, volume 2 of *Annals of Discrete Math.*, pages 151–154. Elsevier, 1978.

7 Vasek Chvátal. Mastermind. *Combinatorica*, 3(3):325–329, 1983. `doi:10.1007/BF02579188`.

8 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2009.

9 Marek Cygan, Fedor V. Fomin, Alexander Golovnev, Alexander S. Kulikov, Ivan Mihajlin, Jakub Pachocki, and Arkadiusz Socała. Tight bounds for Graph Homomorphism and Subgraph Isomorphism. In *SODA 2016*, pages 1643–1649, 2016.

10 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

11 Reinhard Diestel. *Graph Theory*. Springer-Verlag Heidelberg, 2010.

12 David C. Fisher. Fractional colorings with large denominators. *J. Graph Theory*, 20(4):403–409, 1995.

13 Fedor V. Fomin, Pinar Heggernes, and Dieter Kratsch. Exact algorithms for graph homomorphisms. *Theory Comput. Syst.*, 41(2):381–393, 2007. `doi:10.1007/s00224-007-2007-x`.

14 Ariel Gabizon, Daniel Lokshtanov, and Michal Pilipczuk. Fast algorithms for parameterized problems with relaxed disjointness constraints. In *ESA 2015*, volume 9294 of *Lecture Notes in Computer Science*, pages 545–556. Springer, 2015.

15 Vladimir Grebinski and Gregory Kucherov. Optimal reconstruction of graphs under the additive model. *Algorithmica*, 28(1):104–124, 2000. URL: `https://hal.inria.fr/inria-00073517`, `doi:10.1007/s004530010033`.

16 Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981. Corrigendum available at: http://dx.doi.org/10.1007/BF02579139. doi:10.1007/BF02579273.

17 Magnús M. Halldórsson and Guy Kortsarz. Multicoloring: Problems and techniques. In *MFCS 2013*, volume 3153 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2004.

18 Magnús M. Halldórsson, Guy Kortsarz, Andrzej Proskurowski, Ravit Salman, Hadas Shachnai, and Jan Arne Telle. Multicoloring trees. *Inf. Comput.*, 180(2):113–129, 2003.

19 Frédéric Havet. Channel assignment and multicolouring of the induced subgraphs of the triangular lattice. *Discrete Math.*, 233(1-3):219–231, 2001.

20 Pavol Hell and Jaroslav Nešetřil. On the complexity of H-coloring. *J. Comb. Theory, Ser. B*, 48(1):92–110, 1990. doi:10.1016/0095-8956(90)90132-J.

21 Qiang-Sheng Hua, Yuexuan Wang, Dongxiao Yu, and Francis C. M. Lau. Dynamic programming based algorithms for set multicover and multiset multicover problems. *Theor. Comput. Sci.*, 411(26-28):2467–2474, 2010.

22 Russell Impagliazzo and Ramamohan Paturi. On the Complexity of k-SAT. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.

23 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774.

24 Mustapha Kchikech and Olivier Togni. Approximation algorithms for multicoloring planar graphs and powers of square and triangular meshes. *Discrete Math. Theor. Comput. Sci.*, 8(1):159–172, 2006.

25 Fabian Kuhn. Local multicoloring algorithms: Computing a nearly-optimal TDMA schedule in constant time. In *STACS 2009*, volume 3 of *LIPIcs*, pages 613–624. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2009. doi:10.4230/LIPIcs.STACS.2009.1852.

26 Wensong Lin. Multicoloring and Mycielski construction. *Discrete Math.*, 308(16):3565–3573, 2008.

27 Bernt Lindström. On a combinatorial problem in number theory. *Canad. Math. Bull*, 8(4):477–490, 1965. doi:10.4153/CMB-1965-034-2.

28 László Lovász. Kneser's conjecture, chromatic number, and homotopy. *J. Comb. Theory, Ser. A*, 25(3):319–324, 1978.

29 Dániel Marx. The complexity of tree multicolorings. In *MFSC 2002*, volume 2420 of *Lecture Notes in Computer Science*, pages 532–542. Springer, 2002.

30 Colin McDiarmid and Bruce A. Reed. Channel assignment and weighted coloring. *Networks*, 36(2):114–117, 2000.

31 J. W. Moon and L. Moser. On cliques in graphs. *Israel J. Math.*, 3(1):23–28, 1965.

32 Jesper Nederlof. Inclusion exclusion for hard problems. Master's thesis, Department of Information and Computer Science, Utrecht University, 2008. Available at http://www.win.tue.nl/~jnederlo/MScThesis.pdf.

33 K. S. Sudeep and Sundar Vishwanathan. A technique for multicoloring triangle-free hexagonal graphs. *Discrete Math.*, 300(1-3):256–259, 2005.

34 Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Appl. Math.*, 8(1):85–89, 1984. doi:10.1016/0166-218X(84)90081-7.

35 Magnus Wahlström. New plain-exponential time classes for graph homomorphism. *Theory Comput. Syst.*, 49(2):273–282, 2011. doi:10.1007/s00224-010-9261-z.

Exploring the Tractability of the Capped Hose Model

Thomas Bosman[1] **and Neil Olver**[2]

1 **Dept. of Econometrics & Operations Research, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands**
t.n.bosman@vu.nl

2 **Dept. of Econometrics & Operations Research, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands; and**
CWI, Amsterdam, The Netherlands
n.olver@vu.nl

Abstract

Robust network design concerns the design of networks to support uncertain or varying traffic patterns. An especially important case is the *VPN problem*, where the total traffic emanating from any node is bounded, but there are no further constraints on the traffic pattern. Recently, Fréchette et al. [10] studied a generalization of the VPN problem where in addition to these so-called hose constraints, there are individual upper bounds on the demands between pairs of nodes. They motivate their model, give some theoretical results, and propose a heuristic algorithm that performs well on real-world instances.

Our theoretical understanding of this model is limited; it is APX-hard in general, but tractable when either the hose constraints or the individual demand bounds are redundant. In this work, we uncover further tractable cases of this model; our main result concerns the case where each terminal needs to communicate only with two others. Our algorithms all involve *optimally embedding* a certain auxiliary graph into the network, and have a connection to a heuristic suggested by Fréchette et al. for the capped hose model in general.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.1.6 Optimization, G.2.2 Graph Theory

Keywords and phrases robust network design, VPN problem

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.19

1 Introduction

Robust network design (RND) [2] is concerned with designing networks that can efficiently handle uncertain or varying utilization. The motivation comes primarily (though not exclusively) from communication networks. Let $G = (V, E)$, be a graph with edge costs that describes an existing, high-capacity network. A set of *terminals* $W \subseteq V$ is required to communicate over the network, and to enable this, we must reserve capacity on the edges of G for our exclusive use (this is in order to guarantee reliable performance). On each edge, we may buy multiple units of capacity (measured, say, in Mb/s); the cost of the edge represents the per-unit cost of capacity. In the RND framework, demand uncertainty is described by a *demand universe* $\mathcal{U}$. It is simply a set containing all of the demands that need to be routed; the choice of this set will be determined by operational needs or historical data. More precisely, each $D \in \mathcal{U}$ is a matrix where entry D_{ij} describes the demand (measured again, say, in Mb/s) from terminal i to terminal j. It turns out that the universe can always be taken to be a convex set, and will frequently be a polytope.

We will consider only single-path, oblivious routing (other routing schemes are possible, but less relevant to practice). This means that a solution to the RND problem must specify, for each pair of terminals $i, j \in W$, a path P_{ij} that will be used to route the demand between this pair. This path must be fixed ahead of time, and cannot be adjusted as a function of the current demand. Once all these paths have been fixed, a capacity reservation must be made on the network. For any edge $e \in E$, the capacity $u(e)$ must be chosen so that no matter which demand matrix $D \in \mathcal{U}$ is instantiated, the total amount of traffic traversing e does not exceed $u(e)$.

The most studied case of universe is the *hose model* [5, 8]. Here, each terminal $i \in W$ has an associated marginal b_i, and the universe $\mathcal{H}(\boldsymbol{b})$ consists of all demand matrices D for which $\sum_{j \in W} D_{ij} \leq b_i$ for all i, and $D_{ij} = D_{ji}$[1]. The optimization problem for the hose model is called the VPN problem, and it was shown by Goyal et al. [11] to be polynomially solvable.

While the hose model is particularly appealing, especially given its exact solvability, it is very natural to consider generalizations with more expressive modelling power. A number of such generalizations have been considered in the literature [7, 17, 10, 9]. One such generalization, the *capped hose model* was introduced by Fréchette, Shepherd, Thottan and Winzer [10]. It is very natural: in addition to the hose constraints b_i for each $i \in W$, there is an upper bound d_{ij} on the demand between a given pair $i, j \in W$. This leads to the capped hose polytope $\mathcal{H}^{\text{cap}}(\boldsymbol{b}, \boldsymbol{d})$. If $d_{ij} = \infty$ for all pairs $i, j \in W$, then this recovers the hose model; and if $b_i = \infty$ for all $i \in W$, then this recovers the *pipe model*, the somewhat trivial case where the problem is to route a single fixed demand matrix. We refer to Fréchette et al. [10] for further discussion and motivation.

As Fréchette et al. [10] observed, the problem of finding the cheapest solution in the capped hose model generalizes Steiner tree, and hence is APX-hard. Simply consider, for an arbitrarily chosen root $r \in W$, the choice $b_i = 1$ for all $i \in W$, and $d_{ir} = d_{ri} = 1$ for all $i \in W$, $d_{ij} = 0$ otherwise. Beyond this, the complexity and approximability of this problem is poorly understood. In particular, it is open as to whether there is a constant factor approximation algorithm. (The general robust network design problem is hard to approximate within polylog factors [17], but this construction does not apply to this restricted setting.) Moreover, the RND problem under $\mathcal{H}^{\text{cap}}(\boldsymbol{b}, \boldsymbol{d})$ is polynomially solvable for some choices of $\boldsymbol{b}$ and $\boldsymbol{d}$, for example when $\boldsymbol{d}$ is sufficiently large (recovering the hose model), or $\boldsymbol{b}$ is sufficiently large (recovering the pipe model). Our goal in this work is to expand the class of exactly solvable cases.

We focus on the setting where $b_i = 1$ for all $i \in W$ and $d_{ij} \in \{0, \infty\}$ (or equivalently, $d_{ij} \in \{0, 1\}$) for all $i, j \in W$, which we call the *masked hose model*. Instead of parametrizing an instance with $\boldsymbol{d}$ and $\boldsymbol{b}$, we can describe it via the *mask graph* H, which has vertex set W and an edge between each pair of terminals which may communicate. In other words, the universe is the set of all fractional matchings in H. The resulting *masked VPN problem* is a clean generalization of the standard VPN problem (the case where H is the complete graph), and is already very rich from a theoretical standpoint. Again, it is not known if a constant approximation factor is possible for arbitrary mask graphs, and the case where H is a star is APX-hard. It is harmless to restrict to connected mask graphs, since otherwise the problem can be solved separately on each connected component, and the resulting solutions overlaid in G.

[1] This is the symmetric hose model; an asymmetric variant which does not require $D_{ij} = D_{ji}$ is also possible, and different [14, 6, 13].

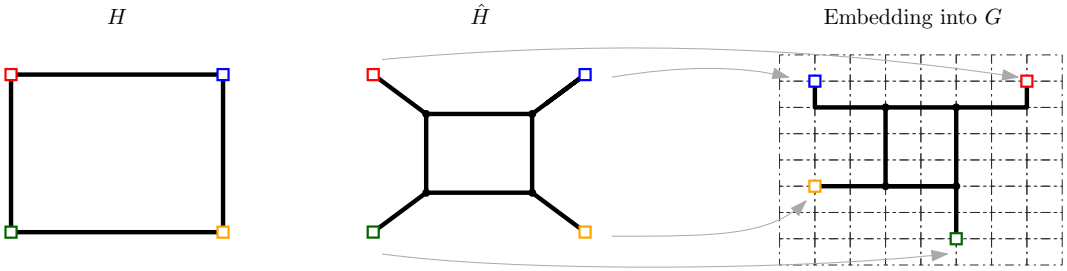


Figure 1 The embedding algorithm for H a cycle; in this example, G is a grid.

Our main result is the following

▶ **Theorem 1.** *The masked VPN problem is polynomially solvable if H is a cycle.*

The algorithm is based on embedding an appropriate auxiliary graph (see Figure 1). Let $\hat{H}$ denote the graph obtained by replacing each terminal i by a new node $\hat{i}$, and then adding back the terminal i along with the edge $\{i, \hat{i}\}$. We give each edge e of $\hat{H}$ a capacity of 1. An *embedding* of $\hat{H}$ into G is simply a mapping ϕ satisfying the following. Each node of $\hat{H}$ is mapped to a node of G, with $\phi(i) = i$ for all $i \in W$; and each edge $\{u, v\} \in E(\hat{H})$ maps to a path in G between $\phi(u)$ and $\phi(v)$. Any embedding implies a path in G between any adjacent pair of adjacent terminals $\{i, j\} \in E(H)$; simply the image under the embedding of the path $(i, \hat{i}, \hat{j}, j)$. After assigning a capacity reservation $u(e) = |\{f \in E(\hat{H}) : e \in \phi(f)\}|$, it is easy to see that this yields a feasible solution to the masked VPN problem for H. Our algorithm simply finds the cheapest possible embedding of $\hat{H}$ into G; this can easily be done by dynamic programming. We show that this is optimal; an overview of the proof strategy can be found in Section 3.1

The cycle may seem like a very specific and restricted case. But understanding cycles has historically been an important stepping stone in the area towards more general results. The VPN Conjecture on the polynomial solvability of the hose model was first solved for the case where the network is a cycle [15, 12], and ideas from [12] were crucial for the resolution of the full conjecture.

We also prove the following.

▶ **Theorem 2.** *The masked VPN problem is exactly solvable if H is a tree with bounded degree.*

Technically, this result is much more straightforward, and we give the proof in Section 4. It exploits the well-known Dreyfus-Wagner algorithm for Steiner tree on a fixed number of terminals [4], which corresponds to the case where H is a bounded degree star. We make heavy use of the dual viewpoint, discussed in Section 2, in order to argue that the solution can be efficiently decomposed into Steiner tree problems. And while our focus is on exactly solvable cases, we remark that an $O(1)$-approximation without any degree bound can readily be obtained (see Theorem 15 in Section 4).

While this result does not require major technical novelty, it yields an interesting message. The algorithm can *also* be interpreted as an embedding algorithm. This time, however, there are multiple options regarding which graph to embed, and we have to choose the best. Begin by constructing $\hat{H}$ in the same fashion as above, splitting out each terminal. But now we go further; for each node $v \in V(\hat{H})$ with degree 4 or more, consider all possible ways of "blowing up" v into a tree with only degree 3 nodes (see Figure 2). Each possible way of blowing up

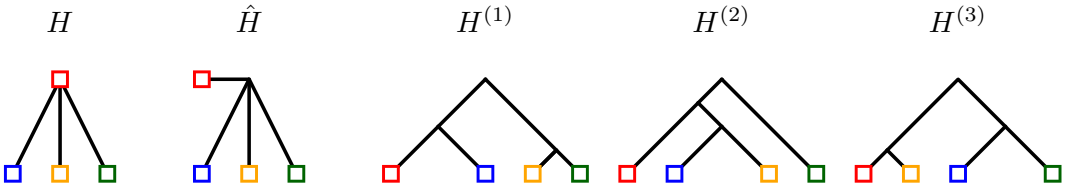


Figure 2 Potential graphs to embed in the case where H is a star.

each node v yields a graph whose embedding yields a solution. The algorithm computes the cheapest possible embedding from all of these possibilities; by using dynamic programming, combined with the assumption of bounded degree, this can be done in polynomial time. Again, this is precisely the idea of the Dreyfus-Wagner algorithm for Steiner tree [4], extended from H a star to H a tree. We also observe that if H was a path, then $\hat{H}$ has maximum degree 3, and so only $\hat{H}$ itself needs to be embedded.

Further, embedding algorithms of this form have been used before in RND, though only embeddings of *trees*. For the standard VPN problem, the optimal solution is simply the optimal embedding of a star with a leaf for each terminal [11]. This is very natural when one considers that the demand universe for the hose model – fractional matchings on the complete graph – is nothing more than the set of demands that are routable on such a star. A generalization of the hose model (different to the one discussed here) defines the universe to be the set of demands routable on a given capacitated tree (with leaf set equal to W) [17]. It has been conjectured that the optimal algorithm is given by the optimal embedding of this tree [18]; it is only known that this yields a constant factor approximation [17].

Partially motivated by this, Fréchette et al. proposed a tree embedding algorithm as a heuristic for the capped hose model. The tree they embed is chosen carefully, albeit heuristically, and they show that this performs well in practice. Our work can be seen as providing an initial theoretical basis for their approach, while suggesting that extending beyond trees may be beneficial.

2 Problem definition and preliminaries

An instance of the masked VPN (MVPN) problem consists of a graph $G = (V, E)$, with edge costs $c : E \to \mathbb{R}_+$ (where $\mathbb{R}_+$ denotes the nonnegative reals), a set of terminals $W \subseteq V$, and a second graph H which has W as its vertex set.

We use $\binom{W}{2}$ to denote the collection of unordered pairs of distinct terminals. The demand universe is defined as

$$\mathcal{H}^{\text{mask}}(H) := \left\{ D \in \mathbb{R}_+^{\binom{W}{2}} : \sum_j D_{ij} \leq 1 \ \forall i \text{ and } D_{ij} = 0 \text{ unless } \{i,j\} \in E(H) \right\}.$$

Our goal is to specify a routing template $\mathcal{P} = \{P_{ij} \ : \{i,j\} \in E(H)\}$, where P_{ij} is a fixed (possibly non-simple) i-j-path used for traffic between terminal i and j. (P_{ij} and P_{ji} refer to the same path.) Given a set of routing paths, we are required to make a capacity reservation $u : E \to \mathbb{R}$, sufficient to route any traffic vector in $\mathcal{H}^{\text{mask}}(H)$. The minimum capacity requirement on edge e is therefore

$$u(e) = \max_{D \in \mathcal{H}^{\text{mask}}(H)} \sum_{\{i,j\} \in \binom{W}{2} : e \in P_{ij}} D_{ij}. \tag{1}$$

The resulting solution has cost $\sum_e c(e)u(e)$, and this we wish to minimize.

We will often take a dual viewpoint of (1). This viewpoint has been exploited before, see [1, 15, 11]. Since (1) is a fractional matching problem, its dual is a fractional vertex cover problem:

$$\begin{aligned} u(e) = \quad & \min \quad \sum_{i \in W} y_i(e) \\ & \text{s.t.} \quad y_i(e) + y_j(e) \geq 1 \qquad \forall \{i,j\} \in E(H),\ e \in P_{ij} \\ & \qquad y_i(e) \geq 0\,. \end{aligned} \tag{2}$$

So, we may rephrase the problem as follows. Each terminal i buys a capacity vector y_i, with the property that $\{e \in E : y_i(e) + y_j(e) \geq 1\}$ contains an i-j-path, for each $\{i,j\} \in E(H)$. The goal is to minimize the total cost $\sum_i c(y_i)$, where $c(y_i) = \sum_{e \in E} c(e) y_i(e)$.

Let $\boldsymbol{y}$ denote the vector whose i'th component y_i is the capacity vector purchased by $i \in W$. At this point we note that, since (2) is a fractional vertex cover problem, $\boldsymbol{y}$ can always assumed to be half-integral. In fact, we will mainly be able to restrict ourselves to integral capacity vectors. In such a case it is convenient to express the solution as a collection of edge sets $\boldsymbol{Y} = (Y_i)_{i \in W}$ where $Y_i = \{e : y_i(e) = 1\}$.

▶ Remark. Through this dual viewpoint, a connection can be made with the work of Iglesias et al. [16]. With a completely different motivation, they consider essentially this problem, explicitly requiring integrality but also *connectivity* of the set of edges purchase by each terminal. Since we show that the optimal solutions satisfy these properties, our results apply in their setting as well.

3 The cycle case

We consider the case where H is a cycle. Let k denote the number of terminals, and assume for convenience that $W = \{1, 2, \ldots, k\}$, with the ordering corresponding to the cycle structure of H. We will interpret all references to terminals modulo k; so terminal 0 refers to terminal k, and terminal $k+1$ to terminal 1.

Our main technical theorem shows that there is always an optimal solution to the MVPN problem that satisfies a simple structure.

▶ **Theorem 3** (Hubbed solution). *There exists an optimal solution to the* MVPN *problem on cycles such that:*

- *for each terminal i there exists a* hub *vertex h_i; and*
- *the routing path $P_{i,i+1}$ is given by concatenating shortest paths from i to h_i, from h_i to h_{i+1}, and from h_{i+1} to $i+1$.*

The optimal location of the hub vertices minimizes the cost

$$\sum_{i \in W} \mathrm{sp}_c(i, h_i) + \mathrm{sp}_c(h_i, h_{i+1}),$$

where sp_c denotes the shortest path distance with respect to the edge costs c. Since H is a cycle the optimal location of hubs $h_{i+1}, \ldots, h_{j-1}$ are independent of hubs $h_{j+1}, \ldots, h_{i-1}$ given the location of h_i and h_j, so we can find these hubs in polynomial time with dynamic programming, yielding Theorem 1.

3.1 Overview

The proof of Theorem 3 involves first showing that there is always an optimal solution of a certain nice form, albeit not yet of the hubbed form we are looking for. We first argue that

we may restrict our focus to solutions $\boldsymbol{y}$ that are integral. Next, we show that there is an integral solution satisfying a certain structure theorem. Roughly speaking, this structure is similar to the hubbed structure we are looking for, but instead of a single hub h_i, there is an odd-cardinality set T_i; instead of a path between i and h_i, we have a $(\{i\} \triangle T_i)$-join; and instead of a path between h_i and h_{i+1}, we have a $(T_i \triangle T_{i+1})$-join. The final step of the argument is then to show that we can take $|T_i| = 1$ for each i, which is then precisely a hubbed solution. This step uses a rather non-obvious "rotation" of the solution to reduce the cardinality of the T_i's.

3.2 Integrality

It will be convenient to work with integral solutions. If k is even, so that H is bipartite, then each fractional matching problem in (2) has an integral optimum. We have to work a bit harder in the case where k is odd.

▶ **Lemma 4.** *There exists an integral optimal solution to the cycle* MVPN *problem.*

Proof. Since any strict subgraph of a cycle is bipartite, the only case where (2) does not have an integral optimal solution, is if it corresponds to a vertex cover problem on the complete cycle. This only happens if the routing path between *every* pair of neighbours uses the edge. So suppose $e = \{u, v\}$ is used on every routing path in a solution $\boldsymbol{y}$. We claim the integral solution $\boldsymbol{Z}$, given by taking Z_i to be the edges of a shortest i-v-path for all terminals i, costs no more than $\boldsymbol{y}$.

Let D be a traffic vector with $D_{i,i+1} = \frac{1}{2}$ for all i. Now if we route D according to the solution $\boldsymbol{y}$, the flow between i and $i+1$ can be split into half a unit of i-v flow and half a unit of v-$(i+1)$-flow, since every routing path passes through e, and thus v.

So D induces a unit i-v flow for each $i \in W$. So $\boldsymbol{y}$ has sufficient capacity to route 1 unit of flow from each $i \in W$ simultaneously. But this costs at least as much as the sum of the shortest paths from each terminal to v, as required. ◀

For the remainder of the proof we will therefore assume that each terminal buys a set of edges. It will be useful to partition these edges into a different collection of sets based on the routing paths they support.

▶ **Definition 5.** A *feasible solution* $\bar{\boldsymbol{X}}$ consists of edge sets $\bar{X}_i$ and $\bar{X}_{i,i+1}$ for each i, such that all edge sets are disjoint and for each i there exists a path $P_{i,i+1}$ connecting terminal i to $i+1$ with

$$P_{i,i+1} \subseteq \bar{X}_i \cup \bar{X}_{i,i+1} \cup \bar{X}_{i+1}.$$

The cost of the solution is $\sum_i c(\bar{X}_i) + \sum_i c(\bar{X}_{i,i+1})$.

One should think of $\bar{X}_i$ as the edges on both $P_{i,i+1}$ and $P_{i-1,i}$ and $\bar{X}_{i,i+1}$ as the remaining edges on $P_{i,i+1}$. Such a solution can be transformed into a feasible solution in original form by setting $X_i = \bar{X}_i \cup \bar{X}_{i,i+1}$ for all i. Indeed, the definition does not confer any advantage to choosing any of the sets $\bar{X}_{i,i+1}$ to be nonempty. But as we will see in the next section, this formulation provides a natural way to express the structure of a feasible solution.

3.3 Structure theorem

For the remainder we will assume that G is a complete graph satisfying the triangle inequality. We may simply replace G with its metric completion to ensure this, and it allows us to bypass some substantial technical awkwardness.

▶ **Definition 6.** Let $\boldsymbol{T}$ be a collection consisting of an odd cardinality sets $T_i \subseteq V \setminus W$ for each terminal i. Then a $\boldsymbol{T}$*-solution* $\bar{\boldsymbol{X}}$ consists of a collection of edge sets $\bar{X}_i$ and $\bar{X}_{i,i+1}$ for each terminal i satisfying:

1. $\bar{X}_i$ is a perfect matching on $T_i \triangle \{i\}$,
2. $\bar{X}_{i,i+1}$ is a perfect matching on $T_i \triangle T_{i+1}$.

The cost of the solution is $\sum_i c(\bar{X}_i) + \sum_i c(\bar{X}_{i,i+1})$.

It is good to observe that Property 1 and 2 of this definition imply that $\bar{\boldsymbol{X}}$ is a feasible solution. The odd degree vertices of $\bar{X}_i \dot\cup \bar{X}_{i,i+1} \dot\cup \bar{X}_{i+1}$ are precisely

$$T_i \triangle \{i\} \triangle T_i \triangle T_{i+1} \triangle T_{i+1} \triangle \{i+1\} = \{i, i+1\},$$

implying that i and $i+1$ are in the same component.

The restriction in the above definition that $T_i \cap W = \emptyset$ is without loss of generality, since we may always modify any given instance by replacing each terminal i in the instance and the solution with a new dummy node $\bar{i}$, and then replacing i in the instance at the same location, attaching the terminal to this dummy node by an edge of zero cost, and including $\{i, \bar{i}\}$ in $\bar{X}_i$.

The following lemma shows that we do not lose anything if we restrict ourselves to $\boldsymbol{T}$-solutions.

▶ **Lemma 7** (Weak Structure Lemma)**.** *Any feasible solution* $\bar{\boldsymbol{X}}$ *may be transformed into a* $\boldsymbol{T}$*-solution* $\bar{\boldsymbol{Y}}$ *of no higher cost for some* $\boldsymbol{T}$ *satisfying* $T_i \subseteq V(\bar{X}_i) \setminus \{i\}$ *for all* $i \in W$*.*

Proof. Define:

- $\bar{Y}'_i = \bar{X}_i \cap E(P_{i-1,i}) \cap E(P_{i,i+1})$, and
- $\bar{Y}'_{i,i+1} = E(P_{i,i+1}) \setminus (\bar{X}_i \cup \bar{X}_{i+1})$.

We then obtain $\bar{\boldsymbol{Y}}$ from $\bar{\boldsymbol{Y}}'$ by shortcutting paths, so that $\bar{Y}_i$ is a collection of vertex disjoint edges for each $i \in W$.

Now for each terminal i choose the vertex set T_i such that $T_i \triangle \{i\}$ is the set of vertices incident to an edge in $\bar{Y}_i$. We claim that $\bar{\boldsymbol{Y}}$ is a $\boldsymbol{T}$-solution.

By construction, $\bar{Y}_i$ is a perfect matching on $T_i \triangle \{i\}$. To see that $\bar{Y}_{i,i+1}$ is a perfect matching on $T_i \triangle T_{i+1}$, note that since $\bar{Y}'_i$ and $\bar{Y}'_{i+1}$ are both contained in the path $P_{i,i+1}$, we may write

$$\bar{Y}'_{i,i+1} = E(P_{i,i+1}) \triangle \bar{Y}_i \triangle \bar{Y}_{i+1}.$$

Thus the odd degree nodes of $\bar{Y}'_{i,i+1}$ are precisely

$$\{i, i+1\} \triangle T_i \triangle \{i\} \triangle T_{i+1} \triangle \{i+1\} = T_i \triangle T_{i+1}.$$

As the odd degree nodes in $\bar{Y}'_{i,i+1}$ and $\bar{Y}_{i,i+1}$ are equal, the result follows. ◀

▶ **Definition 8.** A *strong* $\boldsymbol{T}$*-solution* is a $\boldsymbol{T}$-solution with the additional properties:

(i) $\bar{X}_i \dot\cup \bar{X}_{i,i+1} \dot\cup \bar{X}_{i+1}$ consists of a single i-$(i+1)$-path, and
(ii) each edge in $\bar{X}_{i,i+1}$ is incident to one vertex in T_i and one in T_{i+1}.

Notice that in a strong $\boldsymbol{T}$-solution $\bar{\boldsymbol{X}}$, $|T_i| = |T_j|$ for all $i, j \in W$.

▶ **Lemma 9** (Strong Structure Lemma)**.** *Any* $\boldsymbol{T}$*-solution* $\bar{\boldsymbol{X}}$ *can be transformed into a strong* $\boldsymbol{R}$*-solution* $\bar{\boldsymbol{Y}}$ *of no higher cost, with* $R_i \subseteq T_i$ *for all* $i \in W$*.*

For the proof of this lemma we will need two auxilliary lemmas.

▶ **Lemma 10.** *Let $\bar{\boldsymbol{X}}$ be a $\boldsymbol{T}$-solution such that for some $i \in W$, $\bar{X}_i \dot\cup \bar{X}_{i,i+1} \dot\cup \bar{X}_{i+1}$ does not satisfy property (i) of Definition 8. Then there exists an $\boldsymbol{R}$-solution $\bar{\boldsymbol{Y}}$ of no higher cost, with $R_j \subseteq T_j$ for all $j \in W$, and $R_i \subsetneq T_i$.*

Proof. Since i and $i+1$ are the only vertices that do not have degree 2 in $\bar{X}_i \dot\cup \bar{X}_{i,i+1} \dot\cup \bar{X}_{i+1}$, the connected component containing i and $i+1$ contains an i-$(i+1)$-path; call it P.

Define $R_i = T_i \cap V(P)$, $R_{i+1} = T_{i+1} \cap V(P)$, and $R_j = T_j$ for all other $j \in W$. Note that $R_i \subsetneq T_i$. We will now construct a $\boldsymbol{R}$-solution $\bar{\boldsymbol{Y}}$ as follows. Define $\bar{Y}_j = \bar{X}_j$ for all $j \notin \{i, i+1\}$, and $\bar{Y}_{j,j+1} = \bar{X}_{j,j+1}$ for all $j \notin \{i-1, i, i+1\}$. Now define $\bar{Y}_{i,i+1} = \bar{X}_{i,i+1} \cap P$, so it is a perfect matching on $R_i \triangle R_{i+1}$ with $c(\bar{Y}_{i,i+1}) \leq c(\bar{X}_{i,i+1})$. Also let $\bar{Y}_i = \bar{X}_i \cap P$, which is a perfect matching on $R_i \triangle \{i\}$. We have $c(\bar{Y}_i) = c(\bar{X}_i) - c(Q)$, where $Q = \bar{X}_i \setminus P$ is a perfect matching on $T_i \setminus R_i$. To define $\bar{Y}_{i-1,i}$, first let

$$\bar{Y}'_{i-1,i} = \bar{X}_{i-1,i} \triangle (\bar{X}_i \setminus P).$$

Notice that the odd degree nodes of $\bar{Y}'_{i-1,i}$ are precisely

$$(T_{i-1} \triangle T_i) \triangle (T_i \setminus R_i) = T_{i-1} \triangle R_i = R_{i-1} \triangle R_i.$$

Now, by discarding any cycles and shortcutting paths, we can choose $\bar{Y}_{i-1,i}$ to be a perfect matching on $R_{i-1} \triangle R_i$ that costs no more than $\bar{Y}'_{i-1,i}$. So we have $c(\bar{Y}_{i-1,i}) \leq c(\bar{X}_{i-1,i}) + c(Q)$.

We make precisely the symmetric construction to define $\bar{Y}_{i+1}$ and $\bar{Y}_{i+1,i+2}$. We have obtained the required $\boldsymbol{R}$-solution $\bar{\boldsymbol{Y}}$. ◀

▶ **Lemma 11.** *Let $\bar{\boldsymbol{X}}$ be a $\boldsymbol{T}$-solution that satisfies Property (i) of Definition 8 but where Property (ii) fails for some terminal i. Then there exists an $\boldsymbol{R}$-solution $\bar{\boldsymbol{Y}}$ of no higher cost, with $R_j \subseteq T_j$ for all $j \in W$, and $R_i \subsetneq T_i$.*

Proof. Suppose w.l.o.g. $e = \{u, v\}$ is an edge in $\bar{X}_{i,i+1}$ with both $u, v \in T_i$. Because of Property (i) we know there exists a u-v-path in $\bar{X}_{i-1} \dot\cup \bar{X}_{i-1,i} \dot\cup \bar{X}_i$, say Q.

Let us define a new solution $\bar{\boldsymbol{Y}}$ equal to $\bar{\boldsymbol{X}}$ except for:

$$\begin{aligned} \bar{Y}_i &= (\bar{X}_i \setminus E(Q)) \cup \{e\} \\ \bar{Y}_{i,i+1} &= \bar{X}_{i,i+1} \cup (\bar{X}_i \cap E(Q)) \setminus \{e\}. \end{aligned}$$

The i-$(i+1)$-path in $\bar{\boldsymbol{X}}$ is still feasible in $\bar{\boldsymbol{Y}}$, and we can get an $(i-1)$-i-path $\bar{\boldsymbol{Y}}$ from the respective path in $\bar{\boldsymbol{X}}$, by replacing the subpath Q with the edge e. Thus, $\bar{\boldsymbol{Y}}$ is a feasible solution.

Let P be the maximal path in $\bar{Y}_i$ that contains e. Since every edge on P is used both on some $(i-1)$-i-path and i-$(i+1)$-path, we can replace P by an edge connecting the endpoints in $\bar{Y}_i$ and retain a feasible solution, with the property that

$$V(\bar{Y}_i) \setminus \{i\} \subsetneq V(\bar{X}_i) \setminus \{i\} = T_i,$$

and $V(\bar{Y}_j) = V(\bar{X}_j)$ for $j \neq i$. By Lemma 7 it now follows that we can find an $\boldsymbol{R}$-solution $\bar{\boldsymbol{Z}}$ with $R_i \subseteq V(Y_i) \setminus \{i\} \subsetneq T_i$ and $R_j \subseteq T_j$ for $j \in W$, as required. ◀

Proof. Lemma 9 We arrive at our Lemma from the fact that we can alternatingly apply Lemmas 10 and 11 to a $\boldsymbol{T}$-solution $\bar{\boldsymbol{X}}$ until we have a strong $\boldsymbol{R}$-solution $\bar{\boldsymbol{Y}}$. Since every time we apply Lemma 10, $\sum_{i \in W} |T_i|$ strictly decreases, this procedure must terminate in a finite number of steps. ◀

3.4 From a T-solution to an optimal embedding

▶ **Observation 12.** *Suppose $\bar{\boldsymbol{X}}$ is a strong $\boldsymbol{T}$-solution with $|T_i| = 1$ for all $i \in W$. Then $\bar{\boldsymbol{X}}$ is a hubbed solution.*

As we will see, as long as we have a strong $\boldsymbol{T}$-solution that is not a hubbed solution (implying that $|T_i| = \alpha > 1$ for some α and all i), we can find an $\boldsymbol{R}$-solution such that $\boldsymbol{R}$ is strictly smaller than $\boldsymbol{T}$.

▶ **Lemma 13.** *Given a strong $\boldsymbol{T}$-solution $\bar{\boldsymbol{X}}$ with $|T_i| > 1$ for all i, there exists a strong $\boldsymbol{R}$-solution $\bar{\boldsymbol{Y}}$ of no higher cost with $R_i \subsetneq T_{i+1}$ for all i.*

Proof. We claim that we can find a new solution $\bar{\boldsymbol{Y}}$ with $V(\bar{Y}_i) = T_{i+1} \setminus \{u_{i+1}\} \cup \{i\}$ for some node $u_{i+1} \in T_{i+1}$. It then follows by Lemma 7 and Lemma 9 that we can find a strong $\boldsymbol{R}$-solution $\bar{\boldsymbol{Z}}$ with

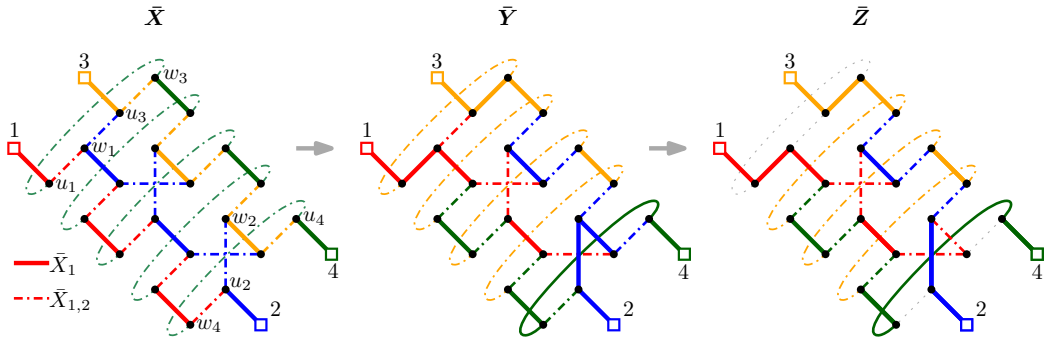
$$R_i \subseteq V(\bar{Y}_i) \setminus \{i\} = T_{i+1} \setminus \{u_{i+1}\},$$

which implies the required result.

For each terminal i we define $u_i \in T_i$ as the node matched to i in $\bar{X}_i$, and $w_i \in T_{i+1}$ as u_i if $u_i \in T_{i+1}$, or the vertex matched to u_i in $\bar{X}_{i,i+1}$ otherwise. Finally let L_i denote the i-u_i-w_i path in $\bar{X}_i \cup \bar{X}_{i,i+1}$.

Now take a solution $\bar{\boldsymbol{Y}}$ equal to $\bar{\boldsymbol{X}}$ except for

$$\bar{Y}_i = \{\{i, w_i\}\} \cup \bar{X}_{i+1} \setminus L_{i+1}$$
$$\text{and} \qquad \bar{Y}_{i,i+1} = \bar{X}_{i+1,i+2} \setminus L_{i+1} \setminus L_{i+2}.$$



We will show that $|T_i| > 1$ implies that $\bar{Y}_i \cup \bar{Y}_{i,i+1} \cup \bar{Y}_{i+1}$ contains an i-$(i+1)$-path. Note that:

$$\bar{Y}_i \cup \bar{Y}_{i,i+1} \cup \bar{Y}_{i+1} = \{\{i, w_i\}, \{i+1, w_{i+1}\}\} \cup E(P_{i+1,i+2} \setminus L_{i+1} \setminus L_{i+2}).$$

As $w_j \in T_{j+1}$ for all j, clearly $P_{i+1,i+2}$ contains a w_i-w_{i+1}-subpath. Since $|T_j| > 1$ for all $j \in W$, we must have that L_{i+1} and L_{i+2} are vertex disjoint, and therefore $E(P_{i+1,i+2} \setminus L_{i+1} \setminus L_{i+2})$ induces a single non-empty connected component.

Now

$$V(P_{i+1,i+2} \setminus L_{i+1} \setminus L_{i+2}) = \begin{cases} V(P_{i+1,i+2}) \setminus \{i+1, u_{i+1}, i+2\} & \text{if } u_{i+1} \neq w_{i+1} \\ V(P_{i+1,i+2}) \setminus \{i+1, i+2\} & \text{otherwise} \end{cases}.$$

But as L_i and L_{i+1} are vertex disjoint, $w_i \neq u_{i+1}$. Therefore $P_{i+1,i+2} \setminus L_{i+1} \setminus L_{i+2}$ contains a w_i-w_{i+1}-path, implying that $\bar{Y}_i \cup \bar{Y}_{i,i+1} \cup \bar{Y}_{i+1}$ contains an i-$(i+1)$-path.

We conclude that $\bar{\boldsymbol{Y}}$ is a feasible solution. Finally note that:

$$\begin{aligned} V(\bar{Y}_i) &= V(\bar{X}_{i+1}) \setminus \{i+1, u_{i+1}\} \cup \{i, w_i\} \\ &= T_{i+1} \triangle \{i+1\} \setminus \{i+1, u_{i+1}\} \cup \{i, w_i\} \\ &= T_{i+1} \setminus \{u_{i+1}\} \cup \{i\}, \end{aligned}$$

where in the last equality we have used that $w_i \in T_{i+1}$. This proves our claim and hence the lemma. ◀

Recall that $|T_i| = |T_j|$ for all $i, j \in W$ in a strong $\boldsymbol{T}$-solution. By repeatedly applying Lemma 13, we obtain an optimal $\boldsymbol{T}$-solution with $|T_i| = 1$ for all $i \in W$, which by Observation 12 is a hubbed solution. This completes the proof of Theorem 3.

4 The tree case

We consider the case where H is a bounded-degree tree. Since H is bipartite, we may restrict ourselves to integral solutions to (2). We first show that there is an optimal solution of a particular form, which we refer to as a *hubbed solution*.

▶ **Lemma 14.** *There exists an optimal solution $\boldsymbol{Y}$ to the tree* MVPN *problem, such that, for some choice $h_{ij} \in V$ for each $\{i, j\} \in E(H)$ (which we call* hub vertices*), Y_i is the edge set of a Steiner tree with terminals $\{i\} \cup \{h_{ij} : \{i, j\} \in E(H)\}$.*

Proof. We prove that we can transform an arbitrary solution $\boldsymbol{Y}$ into a feasible solution $\boldsymbol{Z}$ of the required form.

Choose an arbitrary terminal and consider H to be rooted at this node. Let $C(i)$ denote the set of children of terminal i in H. We construct $\boldsymbol{Z}$ as follows.

We initialize $Z_i = \emptyset$ for all leaf terminals i. Now suppose we have defined Z_j for all the children of a node i. Then define

$$Z'_i = \bigcup_{j \in C(i)} \{e \in Y_i \cup Y_j : e \in P_{ij}\} \setminus Z_j.$$

Now let Z_i be the connected component of (V, Z'_i) that contains i. By working up from the leaves of H, this clearly defines $\boldsymbol{Z}$.

Since $Z_i \subseteq \bigcup_{j \in \{i\} \cup C(i)} Y_j$ for all i, and $Z_i \cap Z_j = \emptyset$ for all $j \in C(i)$, $\boldsymbol{Z}$ costs no more in $\boldsymbol{Y}$.

To see that $\boldsymbol{Z}$ is indeed feasible and of the required form, note that for any terminal i and child $j \in C(i)$, by definition $Z'_i \cup Z_j$ must contain an i-j-path. If Z_j is empty, clearly Z_i must contain an i-j path. We set $h_{ij} = j$ and we are done. If not, then there exists a vertex h_{ij} in the single nonempty connected component of Z_j such that Z'_i contains a path from i to h_{ij}. But that path must be contained in the connected component of Z'_i that contains i, which is exactly Z_i, as required. ◀

With this structural lemma in place, Theorem 2 follows easily.

Proof. Theorem 2 We can solve the Steiner tree problem for a fixed number of terminals in polynomial time [4]. Therefore, for H a tree of bounded degree, finding an optimal solution reduces to finding the location of the hub vertices. We will show that we can do this efficiently with dynamic programming.

Suppose we root H at some terminal r. For each terminal i we let $\zeta(i,h)$ denote the minimum cost of the edge sets bought by all terminals in the subtree rooted at i, over all hubbed solutions such that the hub between i and its parent is located at h.

Now, define $\mathrm{MSt}(X)$ for $X \subseteq V$ as the minimum cost of a Steiner tree on terminal set X. We can calculate $\zeta(\cdot,\cdot)$ recursively as follows:

$$\zeta(i,h) = \min_{(h_j)_{j\in C(i)} \in V^{C(i)}} \mathrm{MSt}(\{i,h\} \cup \{h_j : j \in C(i)\}) + \sum_{j \in C(i)} \zeta(j,h_j).$$

In other words, we simply try all possible combinations of choices of $h_{j_1}, h_{j_2}, \ldots$ for $j_1, j_2, \ldots \in C(i)$. As the degree of H is bounded, the number of combinations is polynomially bounded. Since we can solve the Steiner tree instance in polynomial time as well, the result follows. ◀

As remarked in the introduction, a constant factor approximation can easily be obtained when H is an arbitrary tree, again from the structural lemma.

▶ **Theorem 15.** *The* MVPN *problem where H is a tree has a 2α-approximation, where α is the approximation ratio for Steiner tree.*

Proof. Root H at an arbitrary terminal. Again, let $C(i)$ denote the set of children of terminal i. Take any optimal solution $\boldsymbol{Y}$. Then define a new solution $Z_i := \bigcup_{j \in \{i\} \cup C(i)} Y_j$, which costs at most $2OPT$. Now for each terminal i, Z_i contains a Steiner tree on i and its children $C(i)$.

Let X_i be an α-approximate Steiner tree on i and its children $C(i)$. Then $\boldsymbol{X}$ is a feasible solution, and $c(\boldsymbol{X}) \leq \alpha \cdot c(\boldsymbol{Z}) \leq 2\alpha \cdot OPT$. ◀

At the time of writing the best known approximation for Steiner tree is $\ln 4 + \epsilon < 1.39$ [3], so the MVPN problem where H is a tree is approximable within 2.78.

5 Conclusion

Our results for H a tree can be extended easily to the capped hose model where the support (edges with $d_{ij} > 0$) forms a tree. If the support of $\boldsymbol{d}$ is a cycle, but $\boldsymbol{b}$ and $\boldsymbol{d}$ are otherwise arbitrary, the situation is unclear. There is a natural analog of the embedding algorithm. First, ensure that the components of $\boldsymbol{b}$ and $\boldsymbol{d}$ are all minimal, i.e., no component can be decreased without changing the uncertainty set. Then compute the cheapest embedding of the weighted version of the graph used in Section 3; edges $\{i, h_i\}$ get weight b_i, and edges $\{i, i+1\}$ get weight $d_{i,i+1}$. We leave it as an open question whether this algorithm is always optimal. More speculatively, we feel that our results suggest that embedding algorithms may play a deeper role in the subject than is currently apparent.

References

1. A. Altın, E. Amaldi, P. Belotti, and M. C. Pınar. Provisioning virtual private networks under traffic uncertainty. *Networks*, 49(1):100–115, 2007. doi:10.1002/net.v49:1.
2. Walid Ben-Ameur and Hervé Kerivin. New economical virtual private networks. *Commun. ACM*, 46(6):69–73, 2003. doi:10.1145/777313.777314.
3. Jarosław Byrka, Fabrizio Grandoni, Thomas Rothvoss, and Laura Sanità. Steiner tree approximation via iterative randomized rounding. *Journal of the ACM (JACM)*, 60(1):6, 2013. doi:10.1145/1806689.1806769.
4. Stuart E. Dreyfus and Robert A. Wagner. The Steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

5 N. G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, K. K. Ramakrishnan, and Jacobus E. van der Merwe. A flexible model for resource management in virtual private networks. In *Proc. SIGCOMM*, pages 95–108, 1999. doi:10.1145/316188.316209.

6 Friedrich Eisenbrand, Fabrizio Grandoni, Gianpaolo Oriolo, and Martin Skutella. New approaches for virtual private network design. *SIAM J. Comput.*, 37(3):706–721, 2007.

7 Friedrich Eisenbrand and Edda Happ. Provisioning a virtual private network under the presence of non-communicating groups. In *Proc. CIAC*, pages 105–114, 2006. doi:10.1007/11758471_13.

8 J. A. Fingerhut, S. Suri, and J. S. Turner. Designing least-cost nonblocking broadband networks. *J. Algorithms*, 24(2):287–309, August 1997. doi:10.1006/jagm.1997.0866.

9 S. Fiorini, G. Oriolo, L. Sanità, and D. O. Theis. The VPN problem with concave costs. *SIAM J. Discrete Math.*, 24(3):1080–1090, 2010.

10 Alexandre Fréchette, F. Bruce Shepherd, Marina K. Thottan, and Peter J. Winzer. Shortest path versus multi-hub routing in networks with uncertain demand. In *Proc. INFOCOM*, pages 710–718, 2013. doi:10.1109/INFCOM.2013.6566857.

11 Navin Goyal, Neil Olver, and F. Bruce Shepherd. The VPN Conjecture is true. *J. ACM*, 60(3):17:1–17:17, 2013. doi:10.1145/2487241.2487243.

12 Fabrizio Grandoni, Volker Kaibel, Gianpaolo Oriolo, and Martin Skutella. A short proof of the VPN tree routing conjecture on ring networks. *Oper. Res. Lett.*, 36(3):361–365, 2008. doi:10.1016/j.orl.2007.10.008.

13 Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. From uncertainty to nonlinearity: Solving virtual private network via single-sink buy-at-bulk. *Mathematics of Operations Research*, 36(2):185–204, 2011. doi:10.1287/moor.1110.0490.

14 Anupam Gupta, Amit Kumar, Martin Pál, and Tim Roughgarden. Approximation via cost sharing: Simpler and better approximation algorithms for network design. *J. ACM*, 54(3):11, 2007. doi:10.1145/1236457.1236458.

15 C. A. J. Hurkens, J. C. M. Keijsper, and L. Stougie. Virtual private network design: A proof of the tree routing conjecture on ring networks. *SIAM J. Discrete Math.*, 21(2):482–503, 2007.

16 Jennifer Iglesias, Rajmohan Rajaraman, R. Ravi, and Ravi Sundaram. Designing overlapping networks for publish-subscribe systems. In *Proc. APPROX*, pages 381–395, 2015. doi:10.4230/LIPIcs.APPROX-RANDOM.2015.381.

17 N. Olver and F. B. Shepherd. Approximability of robust network design. In *Proc. SODA*, pages 1097–1105, 2010. doi:10.1137/1.9781611973075.89.

18 Neil Olver. A note on hierarchical hubbing for a generalization of the VPN problem. *Oper. Res. Lett.*, 44(2):191–195, 2016. doi:10.1016/j.orl.2015.12.020.

Sampling Geometric Inhomogeneous Random Graphs in Linear Time*

Karl Bringmann[1], Ralph Keusch[2], and Johannes Lengler[3]

1 **Max-Planck-Institute for Informatics, Saarbrücken, Germany**
`kbringma@mpi-inf.mpg.de`
2 **Institute of Theoretical Computer Science, ETH Zürich, Switzerland**
`rkeusch@inf.ethz.ch`
3 **Institute of Theoretical Computer Science, ETH Zürich, Switzerland**
`lenglerj@inf.ethz.ch`

Abstract

Real-world networks, like social networks or the internet infrastructure, have structural properties such as large clustering coefficients that can best be described in terms of an underlying geometry. This is why the focus of the literature on theoretical models for real-world networks shifted from classic models without geometry, such as Chung-Lu random graphs, to modern geometry-based models, such as hyperbolic random graphs.

With this paper we contribute to the theoretical analysis of these modern, more realistic random graph models. Instead of studying directly hyperbolic random graphs, we introduce a generalization that we call *geometric inhomogeneous random graphs* (GIRGs). Since we ignore constant factors in the edge probabilities, GIRGs are technically simpler (specifically, we avoid hyperbolic cosines), while preserving the qualitative behaviour of hyperbolic random graphs, and we suggest to replace hyperbolic random graphs by this new model in future theoretical studies.

We prove the following fundamental structural and algorithmic results on GIRGs. (1) As our main contribution we provide a sampling algorithm that generates a random graph from our model in expected *linear* time, improving the best-known sampling algorithm for hyperbolic random graphs by a substantial factor $O(\sqrt{n})$. (2) We establish that GIRGs have clustering coefficients in $\Omega(1)$, (3) we prove that GIRGs have small separators, i.e., it suffices to delete a sublinear number of edges to break the giant component into two large pieces, and (4) we show how to compress GIRGs using an expected linear number of bits.

1998 ACM Subject Classification G.2.2 Graph Theory, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases real-world networks, random graph models, sampling algorithms, compression algorithms, hyperbolic random graphs

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.20

1 Introduction

Real-world networks, like social networks or the internet infrastructure, have structural properties that can best be described using *geometry*. For instance, in social networks two people are more likely to know each other if they live in the same region and share hobbies, both of which can be encoded as spatial information. This geometric structure may be responsible for some of the key properties of real-world networks, e.g., an underlying geometry

* A full version of this paper, including all proofs, is available at `http://arxiv.org/abs/1511.00576`.

naturally induces a large number of triangles, or large *clustering coefficient*: Two of one's friends are likely to live in one's region and have similar hobbies, so they are themselves similar and thus likely to know each other.

Classic mathematical models of real-world networks are *scale-free* (i.e., have a power-law degree distribution) and *small worlds* (i.e., most pairs of vertices have small graph-theoretic distance), thus reproducing these two key findings of large real-world networks. But since they have no underlying geometry their clustering coefficient is as small as $n^{-\Omega(1)}$; this holds in particular for preferential attachment graphs [3] and Chung-Lu random graphs [24, 25, 26] (and their variants [13, 41]). In order to close this gap between the empirically observed clustering coefficient and theoretical models, much of the recent work on models for real-world networks focussed on scale-free random graph models that are equipped with an underlying geometry, such as hyperbolic random graphs [11, 42], spatial preferred attachment [2], and many others [13, 14, 15, 35]. The basic properties – scale-freeness, small-world, and large clustering coefficient – have been rigorously established for most of these models. Beyond the basics, experiments suggest that these models have some very desirable properties.

In particular, hyperbolic random graphs are a promising model, as Boguñá et al. [11] computed a (heuristic) maximum likelihood fit of the internet graph into the hyperbolic random graph model and demonstrated its quality by showing that greedy routing in the underlying geometry of the fit finds near-optimal shortest paths. Further properties that have been studied on hyperbolic random graphs, mostly agreeing with empirical findings on real-world networks, are scale-freeness and clustering coefficient [33, 20], existence of a giant component [9], diameter [37, 32], average distance [1], separators and treewidth [6], spectral gap [38], bootstrap percolation [21], and clique number [7]. Algorithmic aspects include sampling algorithms [47], embedding algorithms [8], and compression schemes [45].

Our goal is to improve algorithmic and structural results on the promising model of hyperbolic random graphs. However, it turns out to be beneficial to work with a more general model, that we introduce with this paper: In a *geometric inhomogeneous random graph* (GIRG), every vertex v comes with a weight w_v (which we assume to follow a power law in this paper) and picks a uniformly random position x_v in the d-dimensional torus $\mathbb{T}^d$. Two vertices u, v then form an edge independently with probability p_{uv}, which is proportional to $\mathsf{w}_u\mathsf{w}_v$ and inversely proportional to some power of their distance $\|\mathsf{x}_u - \mathsf{x}_v\|$, see Section 2 for details. A major difference between hyperbolic random graphs and our generalisation is that we ignore constant factors in the edge probabilities p_{uv}. This allows to greatly simplify the edge probability expressions, thus reducing the technical overhead. GIRGs can be interpreted as a geometric variant of the classic Chung-Lu random graphs. Recently, with scale-free percolation a closely related model has been introduced [28] where the vertex set is given by the grid $\mathbb{Z}^d$. This model is similar with respect to component structure, clustering, and small-world properties [29, 34], but none of the algorithmic aspects studied in the present paper (sampling, compression, also separators) has been regarded thereon.

The basic connectivity properties of GIRGs follow from more general considerations in [17], where an even more general model of generic augmented Chung-Lu graphs is studied. In particular, with high probability[1] GIRGs have a giant component and polylogarithmic diameter, and a.a.s. doubly-logarithmic average distance within the giant. However, general studies such as [17] are limited to properties that do not depend on the specific underlying geometry. Recently, GIRGs turned out to be accesible for studying processes such as bootstrap percolation [39] and greedy routing [19].

[1] We say that an event holds *with high probability* (w.h.p.) if it holds with probability $1 - n^{-\omega(1)}$. If it holds with probability $1 - o(1)$, we say that it holds *asymptotically almost surely* (a.a.s.).

Our contribution. As our main result, we present a sampling algorithm that generates a random graph from our model in expected linear time. This improves the trivial sampling algorithm by a factor $O(n)$ and the best-known algorithm for hyperbolic random graphs by a factor $O(\sqrt{n})$ [47]. We also prove that the underlying geometry indeed causes GIRGs to have a clustering coefficient in $\Omega(1)$. Moreover, we show that GIRGs have small separators of expected size $n^{1-\Omega(1)}$; this is in agreement with empirical findings on real-world networks [5]. We then use the small separators to prove that GIRGs can be efficiently compressed (i.e., they have low entropy), specifically, we show how to store a GIRG using $O(n)$ bits in expectation. Finally, we show that hyperbolic random graphs are indeed a special case of GIRGs, so that all aforementioned results also hold for hyperbolic random graphs.

2 Model and Results

2.1 Definition of the Model

We prove algorithmic and structural results in a new random graph model which we call *geometric inhomogeneous random graphs*. In this model, each vertex v comes with a weight w_v and with a random position x_v in a geometric space, and the set of edges E is also random. We start by defining the by-now classical Chung-Lu model and then describe the changes that yield our variant with underlying geometry.

Chung-Lu random graph. For $n \in \mathbb{N}$ let $\mathsf{w} = (\mathsf{w}_1, \dots, \mathsf{w}_n)$ be a sequence of positive weights. We call $\mathsf{W} := \sum_{v=1}^{n} \mathsf{w}_v$ the *total weight*. The Chung-Lu random graph $G(n, \mathsf{w})$ has vertex set $V = [n] = \{1, \dots, n\}$, and two vertices $u \neq v$ are connected by an edge independently with probability $p_{uv} = \Theta\big(\min\big\{1, \frac{\mathsf{w}_u \mathsf{w}_v}{\mathsf{W}}\big\}\big)$ [24, 25]. Note that the term $\min\{1, .\}$ is necessary, as the product $\mathsf{w}_u \mathsf{w}_v$ may be larger than W. Classically, the Θ simply hides a factor 1, but by introducing the Θ the model also captures similar random graphs, like the Norros-Reittu model [41], while important properties stay asymptotically invariant.

Geometric inhomogeneous random graph (GIRG). Note that we obtain a circle by identifying the endpoints of the interval $[0, 1]$. Then the distance of $x, y \in [0, 1]$ along the circle is $|x - y|_C := \min\{|x - y|, 1 - |x - y|\}$. We fix a dimension $d \geq 1$ and use as our *ground space* the d-dimensional torus $\mathbb{T}^d = \mathbb{R}^d / \mathbb{Z}^d$, which can be described as the d-dimensional cube $[0, 1]^d$ where opposite boundaries are identified. As distance function we use the ∞-norm on $\mathbb{T}^d$, i.e., for $x, y \in \mathbb{T}^d$ we define $\|x - y\| := \max_{1 \leq i \leq d} |x_i - y_i|_C$.

As for Chung-Lu graphs, we consider the vertex set $V = [n]$ and a weight sequence w (in this paper we require the weights to follow a power law with exponent $\beta > 2$, see next paragraph). Additionally, for any vertex v we draw a point $\mathsf{x}_v \in \mathbb{T}^d$ uniformly and independently at random. Again we connect vertices $u \neq v$ independently with probability $p_{uv} = p_{uv}(r)$, which now depends not only on the weights $\mathsf{w}_u, \mathsf{w}_v$ but also on the positions $\mathsf{x}_u, \mathsf{x}_v$, more precisely, on the distance $r = \|\mathsf{x}_u - \mathsf{x}_v\|$. We require for some constant $\alpha > 1$ the following edge probability condition:

$$p_{uv} = \Theta\left(\min\left\{ \frac{1}{\|\mathsf{x}_u - \mathsf{x}_v\|^{\alpha d}} \cdot \left(\frac{\mathsf{w}_u \mathsf{w}_v}{\mathsf{W}}\right)^{\alpha}, 1 \right\} \right). \tag{EP1}$$

We also allow $\alpha = \infty$ and in this case require that

$$p_{uv} = \begin{cases} \Theta(1) & \text{if } \|\mathsf{x}_u - \mathsf{x}_v\| \leq O\big(\big(\frac{\mathsf{w}_u \mathsf{w}_v}{\mathsf{W}}\big)^{1/d}\big) \\ 0 & \text{if } \|\mathsf{x}_u - \mathsf{x}_v\| \geq \Omega\big(\big(\frac{\mathsf{w}_u \mathsf{w}_v}{\mathsf{W}}\big)^{1/d}\big), \end{cases} \tag{EP2}$$

where the constants hidden by O and Ω do not have to match, i.e., there can be an interval $[c_1(\frac{\mathsf{w}_u\mathsf{w}_v}{\mathsf{W}})^{1/d}, c_2(\frac{\mathsf{w}_u\mathsf{w}_v}{\mathsf{W}})^{1/d}]$ for $\|\mathsf{x}_u - \mathsf{x}_v\|$ where the behaviour of p_{uv} is arbitrary. This finishes the definition of GIRGs. The free parameters of the model are $\alpha \in (1, \infty]$, $d \in \mathbb{N}$, the concrete weights w with power-law exponent $\beta > 2$ and average weight W/n, the concrete function $f_{uv}(\mathsf{x}_u, \mathsf{x}_v)$ replacing the Θ in p_{uv}, and for $\alpha = \infty$ the constants hidden by O, Ω in the requirement for p_{uv}. We will typically hide the constants $\alpha, d, \beta, \mathsf{W}/n$ by O-notation.

Power-law weights. As is often done for Chung-Lu graphs, we assume throughout this paper that the weights follow a *power law*: the fraction of vertices with weight at least w is proportional to $w^{1-\beta}$ for some $2 < \beta < 3$ (the *power-law exponent* of w). More precisely, we assume that for some $\bar{w} = \bar{w}(n)$ with $n^{\omega(1/\log\log n)} \leq \bar{w} \leq n^{(1-\Omega(1))/(\beta-1)}$, the sequence w satisfies the following conditions:

(PL1) the minimum weight is constant, i.e., $\mathsf{w}_{\min} := \min\{\mathsf{w}_v \mid 1 \leq v \leq n\} = \Omega(1)$;

(PL2) for all $\eta > 0$ there exist constants $c_1, c_2 > 0$ such that

$$c_1 \frac{n}{w^{\beta-1+\eta}} \leq \#\{1 \leq v \leq n \mid \mathsf{w}_v \geq w\} \leq c_2 \frac{n}{w^{\beta-1-\eta}},$$

where the first inequality holds for all $\mathsf{w}_{\min} \leq w \leq \bar{w}$ and the second for all $w \geq \mathsf{w}_{\min}$. We remark that these are standard assumptions for power-law graphs with average degree $\Theta(1)$. In particular, (PL2) implies that the average weight W/n is $\Theta(1)$. An example is the widely used weight function $\mathsf{w}_v := \delta \cdot (n/v)^{1/(\beta-1)}$ with parameter $\delta = \Theta(1)$.

Discussion of the model. The choice of the ground space $\mathbb{T}^d$ is in the spirit of the classic random geometric graphs [44]. We prefer the torus to the hyper-cube for technical simplicity, as it yields symmetry. However, one could replace $\mathbb{T}^d$ by $[0,1]^d$ or other manifolds like the d-dimensional sphere; our results will still hold verbatim. Moreover, since in fixed dimension all L_p-norms on $\mathbb{T}^d$ are equivalent and since the edge probabilities p_{uv} have a constant factor slack, our choice of the L_∞-norm is without loss of generality (among all norms).

The model is already motivated since it generalizes the celebrated hyperbolic random graphs (see Theorem 7). Let us nevertheless discuss why our choice of edge probabilities is *natural*: The term $\min\{., 1\}$ is necessary, as in the Chung-Lu model, because p_{uv} is a probability. To obtain a geometric model, where adjacent vertices are likely to have small distance, p_{uv} should decrease with increasing distance $\|\mathsf{x}_u - \mathsf{x}_v\|$, and an inverse polynomial relation seems reasonable. The constraint $\alpha > 1$ is necessary to cancel the growth of the volume of the ball of radius r proportional to r^d, so that we expect most neighbors of a vertex to lie close to it. Finally, the factor $\left(\frac{\mathsf{w}_u\mathsf{w}_v}{\mathsf{W}}\right)^\alpha$ ensures that the marginal probability of vertices u, v with weights $\mathsf{w}_u, \mathsf{w}_v$ forming an edge is $\Pr[u \sim v] = \Theta\left(\min\left\{\frac{\mathsf{w}_u\mathsf{w}_v}{\mathsf{W}}, 1\right\}\right)$, as in the Chung-Lu model, and this probability does not change by more than a constant factor if we fix either x_u or x_v. This is why we see our model as a geometric variant of Chung-Lu random graphs. For a fixed vertex $u \in V$ we can sum up $\Pr[u \sim v \mid \mathsf{x}_u]$ over all vertices $v \in V \setminus \{u\}$, and it follows $\mathbb{E}[\deg(u)] = \Theta(\mathsf{w}_u)$. The main reason why GIRGs are also *technically easy* is that for any vertex u with fixed position x_u the incident edges $\{u, v\}$ are independent.

Finally, after rescaling the parameters ($\tilde{\mathsf{x}}_v := n^{1/d}\mathsf{x}_v$, $\tilde{\alpha} := d\alpha$, $\tau := 1 + (\beta-1)/\alpha$, see [46] for details), the GIRG model is closely related to scale-free percolation [28].

Sampling the weights. In the definition we assume that the weight sequence w is fixed. However, if we sample the weights independently according to an appropriate power-law distribution with minimum weight $\mathsf{w}_{\min}$ and density $f(w) \sim w^{-\beta}$, then the sampled weight sequence will follow a power law and fulfils (PL1) and (PL2) with probability $1 - n^{-\Omega(1)}$. Hence, a model with sampled weights is a.a.s. included in our model.

2.2 Structural Properties of GIRGs

As discussed in the introduction, reasonable random graph models for real-world networks should reproduce a power-law degree distribution and small graph-theoretical distances between nodes. For the GIRG model, these structural properties follow from a more general class of generic augmented Chung-Lu random graphs that have been studied in [17]. This framework has weaker assumptions on the underlying geometry than GIRGs. A short comparison reveals that GIRGs are a special case of this general class of random graph models. In the following we list the results of [17] transferred to GIRGs. As we are using power-law weights and $\mathbb{E}[\deg(v)] = \Theta(\mathsf{w}_v)$ holds for all $v \in V$, it is not surprising that the degree sequence follows a power-law.

▶ **Theorem 1** (Theorem 2.1 in [17]). *W.h.p. the degree sequence of a GIRG follows a power law with exponent β and average degree $\Theta(1)$.*

The next result determines basic connectivity properties. Note that for $\beta > 3$, they are not well-behaved, in particular since in this case even threshold hyperbolic random graphs do not possess a giant component of linear size [10]. This is one reason why we assume $2 < \beta < 3$ throughout the paper. For the following theorem, we require the additional assumption $\bar{w} = \omega(n^{1/2})$ in the limit case $\alpha = \infty$.

▶ **Theorem 2** (Theorems 2.2 and 2.3 in [17]). *W.h.p. the largest component of a GIRG has linear size and diameter $\log^{O(1)} n$, while all other components have size $\log^{O(1)} n$. Moreover, the average distance of vertices in the largest component is $(2 \pm o(1)) \frac{\log\log n}{|\log(\beta-2)|}$ in expectation and with probability $1 - o(1)$.*

We remark that most results of this paper crucially depend on an underlying geometry, and thus do *not* hold in the general model from [17].

2.3 Results

Sampling. Sampling algorithms that generate a random graph from a fixed distribution are known for Chung-Lu random graphs and others, running in expected linear time [4, 40]. As our main result, we present such an algorithm for GIRGs. This greatly improves the trivial $O(n^2)$ sampling algorithm (throwing a biased coin for each possible edge), as well as the best previous algorithm for threshold hyperbolic random graphs with expected time $O(n^{3/2})$ [47]. It allows to run experiments on much larger graphs than the ones with $\approx 10^4$ vertices in [11]. In addition to our model assumptions, here we assume that the Θ in our requirement on p_{uv} is sufficiently explicit, i.e., we can compute p_{uv} exactly and we know a constant $c > 0$ such that replacing Θ by c yields an upper bound on p_{uv}, see Section 3 for details.

▶ **Theorem 3** (Section 3). *Sampling a GIRG can be done in expected time $O(n)$.*

Clustering. In social networks, two friends of the same person are likely to also be friends with each other. This property of having many triangles is captured by the clustering coefficient, defined as the probability when choosing a random vertex v and two random neighbors $v_1 \neq v_2$ of v that v_1 and v_2 are adjacent (if v does not have two neighbors then its contribution to the clustering coefficient is 0). While Chung-Lu random graphs have a very small clustering coefficient of $n^{-\Omega(1)}$, it is easy to show that the clustering coefficient of GIRGs is $\Theta(1)$. This is consistent with empirical data of real-world networks [31] and the constant clustering coefficient of hyperbolic random graphs determined in [20, 33, 45].

▶ **Theorem 4.** *W.h.p. the clustering coefficient of a GIRG is* $\Theta(1)$.

Proof Outline. We show that the clustering coefficient is dominated by the contribution of constant-weight vertices v. Let $v \in V$ be a vertex of weight $\mathsf{w}_v = \Theta(1)$. Then, with at least constant probability, (i) $\deg(v) \geq 2$, and (ii) all neighbors of v are located in a ball of radius $cn^{-1/d}$ around x_v, for a sufficiently small constant $c > 0$. If the neighborhood of v has this property, then two random neighbors v_1, v_2 of v are connected with constant probability. Therefore, the expected contribution of v to the clustering coefficient is $\Omega(\frac{1}{n})$. As the number of such vertices v is $\Theta(n)$, it follows that the expected clustering coefficient is $\Theta(1)$. Proving the w.h.p.-statement requires additional arguments and the application of Azuma-type concentration inequalities with bad events. The detailed proof is included in the full version [18]. ◀

Stability. For real-world networks, a key property to analyze is their stability under attacks. It has been empirically observed that many real-world networks have small separators of size n^c, $c < 1$ [5]. In contrast, Chung-Lu random graphs are unrealistically stable, since any deletion of $o(n)$ nodes or edges reduces the size of the giant component by at most $o(n)$ [13]. We show that GIRGs agree with the empirical results much better. Specifically, if we cut the ground space $\mathbb{T}^d$ into two halves along one of the axes then we roughly split the giant component into two halves, but the number of edges passing this cut is quite small, namely $n^{1-\Omega(1)}$. Thus, GIRGs are prone to (quite strong) adversarial attacks, just as many real-world networks. Furthermore, their small separators are useful for many algorithms, e.g., the compression scheme of the next paragraph.

▶ **Theorem 5** (Section 4). *A.a.s. it suffices to delete* $O\left(n^{\max\{2-\alpha,3-\beta,1-1/d\}+o(1)}\right)$ *edges of a GIRG to split its giant component into two parts of linear size each.*

Since we assume $\alpha > 1$, $\beta > 2$, and $d = \Theta(1)$, the number of deleted edges is indeed $n^{1-\Omega(1)}$. Recently, Bläsius et al. [6] proved a better bound of $O(n^{(3-\beta)/2})$ for threshold hyperbolic random graphs which correspond to GIRGs with parameters $d = 1$ and $\alpha = \infty$.

Entropy. The internet graph has empirically been shown to be well compressible, using only 2-3 bits per edge [5, 12]. This is not the case for the Chung-Lu model, as its entropy is $\Theta(n \log n)$ [23]. We show that GIRGs have linear entropy, as is known for threshold hyperbolic random graphs [45].

▶ **Theorem 6** (Section 4). *We can store a GIRG using* $O(n)$ *bits in expectation. The resulting data structure allows to query the degree of any vertex and its* i*-th neighbor in time* $O(1)$. *The compression algorithm runs in time* $O(n)$.

Hyperbolic random graphs. We establish that hyperbolic random graphs are an example of one-dimensional GIRGs, and that the often studied special case of threshold hyperbolic graphs is obtained by our limit case $\alpha = \infty$. Specifically, we obtain hyperbolic random graphs from GIRGs by setting the dimension $d = 1$, the weights to a specific power law, and the Θ in the edge probability p_{uv} to a specific, complicated function.

▶ **Theorem 7.** *For every choice of parameters in the hyperbolic random graph model, there is a choice of parameters in the GIRG model such that the two resulting distributions of graphs coincide.*

In particular, all our results on GIRGs hold for hyperbolic random graphs, too. Moreover, as our proofs are much less technical than typical proofs for hyperbolic random graphs, we suggest to switch from hyperbolic random graphs to GIRGs in future studies. We prove Theorem 7 in the full version of this paper [18].

2.4 Preliminaries

We introduce a geometric ordering of the vertices, which we will use both for the sampling and for the compression algorithm. Consider the ground space $\mathbb{T}^d$, split it into 2^d equal cubes, and repeat this process with each created cube; we call the resulting cubes *cells*. Cells are cubes of the form $C = [x_1 2^{-\ell}, (x_1+1)2^{-\ell}) \times \ldots \times [x_d 2^{-\ell}, (x_d+1)2^{-\ell})$ with $\ell \geq 0$ and $0 \leq x_i < 2^\ell$. We represent cell C by the tuple $(\ell, x_1, \ldots, x_d)$. The volume of C is $\text{VOL}(C) = 2^{-\ell \cdot d}$. For $0 < x \leq 1$ we let $\lceil x \rceil_{2^d}$ be the smallest number larger or equal to x that is realized as the volume of a cell, or in other words x rounded up to a power of 2^d, $\lceil x \rceil_{2^d} = \min\{2^{-\ell \cdot d} \mid \ell \in \mathbb{N}_0 \colon 2^{-\ell \cdot d} \geq x\}$. Note that the cells of a fixed level ℓ partition the ground space. We obtain a *geometric ordering* of these cells by following the recursive construction of cells in a breadth-first-search manner. This yields the following lemma.

▶ **Lemma 8** (Geometric ordering). *There is an enumeration of the cells $C_1, \ldots, C_{2^{\ell d}}$ of level ℓ such that for every cell C of level $\ell' < \ell$ the cells of level ℓ contained in C form a consecutive block $C_i, \ldots, C_j$ in the enumeration.*

3 Sampling Algorithm

In this section we show that GIRGs can be sampled in expected time $O(n)$. The running time depends exponentially on the fixed dimension d. In addition to our model assumptions, in this section we require that (1) edge probabilities p_{uv} can be computed in constant time (given any vertices u, v and positions $\mathsf{x}_u, \mathsf{x}_v$) and (2) we know an explicit constant $c > 0$ such that if $\alpha < \infty$ we have

$$p_{uv} \leq \min\left\{c \frac{1}{\|\mathsf{x}_u - \mathsf{x}_v\|^{\alpha d}} \cdot \left(\frac{\mathsf{w}_u \mathsf{w}_v}{\mathsf{W}}\right)^\alpha, 1\right\}.$$

Note that existence of c follows from our model assumptions. In the remainder of this section we introduce building blocks of our algorithm (Section 3.1) and present our algorithm (Section 3.2) and its analysis (Section 3.3). Note that in the full version, we also show how the sampling algorithm can be adapted to the case $\alpha = \infty$.

3.1 Building Blocks

Data structures. We first build a basic data structure on a set of points P that allows to access the points in a given cell C (of volume at least ν) in constant time.

▶ **Lemma 9.** *Given a set of points P and $0 < \nu \leq 1$, in time $O(|P| + 1/\nu)$ we can construct a data structure $\mathcal{D}_\nu(P)$ supporting the following queries in time $O(1)$:*

- *given a cell C of volume at least ν, return $|C \cap P|$,*
- *given a cell C of volume at least ν and a number k, return the k-th point in $C \cap P$ (in a fixed ordering of $C \cap P$ depending only on P and ν).*

Proof. Let $\mu = \lceil \nu \rceil_{2^d} = 2^{-\ell \cdot d}$, so that $\nu \leq \mu \leq O(\nu)$. Following the recursive construction of cells, we can determine a geometric ordering of the cells of volume μ as in Lemma 8 in time $O(1/\mu) = O(1/\nu)$; say $C_1, \ldots, C_{1/\mu}$ are the cells of volume μ in the geometric ordering.

We store this ordering by storing a pointer from each cell $C_i = (\ell, x_1, \dots, x_d)$ to its successor $C_{i+1} = (\ell, x'_1, \dots, x'_d)$, which allows to scan the cells $C_1, \dots, C_{1/\mu}$ in linear time. For any point $x \in P$, using the floor function we can determine in time $O(1)$ the cell $(\ell, x_1, \dots, x_d)$ of volume μ that x belongs to (in our machine model we assume that the floor function can be computed in constant time). This allows to determine the numbers $|C_i \cap P|$ for all i in time $O(|P| + 1/\nu)$. We also compute each prefix sum $s_i := \sum_{j<i} |C_j \cap P|$ and store it at cell $C_i = (\ell, x_1, \dots, x_d)$. Using an array $A[.]$ of size $|P|$, we store (a pointer to) the k-th point in $C_i \cap P$ at position $A[s_i + k]$. This preprocessing can be performed in time $O(|P| + 1/\nu)$.

A given cell C of volume at least ν may consist of several cells of volume μ. By Lemma 8, these cells form a contiguous subsequence $C_i, C_{i+1}, \dots, C_{j-1}, C_j$ of $C_1, \dots, C_{1/\mu}$, so that the points $C \cap P$ form a contiguous subsequence of A. For constant access time, we store for each cell C of volume at least ν the indices s_C, e_C of the first and last point of $C \cap P$ in A. Then $|C \cap P| = e_C - s_C + 1$ and the k-th point in $C \cap P$ is stored at $A[s_C + k]$. Thus, both queries can be answered in constant time. Note that the ordering $A[.]$ of the points in $C \cap P$ is a mix of the geometric ordering of cells of volume μ and the given ordering of P within a cell of volume μ, in particular this ordering indeed only depends on P and ν. ◀

Next we construct a partitioning of $\mathbb{T}^d \times \mathbb{T}^d$ into products of cells $A_i \times B_i$. This partitioning allows to split the problem of sampling the edges of a GIRG into one problem for each $A_i \times B_i$, which is beneficial, since each product $A_i \times B_i$ has one of two easy types. For any $A, B \subseteq \mathbb{T}^d$ we denote the distance of A and B by $d(A, B) = \inf_{a \in A, b \in B} \|a - b\|$.

▶ **Lemma 10.** *Let $0 < \nu \leq 1$. In time $O(1/\nu)$ we can construct a set $\mathcal{P}_\nu = \{(A_1, B_1), \dots, (A_s, B_s)\}$ such that*
(1) *A_i, B_i are cells with $\text{VOL}(A_i) = \text{VOL}(B_i) \geq \nu$,*
(2) *for all i, either $d(A_i, B_i) = 0$ and $\text{VOL}(A_i) = \lceil \nu \rceil_{2^d}$ (type I) or $d(A_i, B_i) \geq \text{VOL}(A_i)^{1/d}$ (type II),*
(3) *the sets $A_i \times B_i$ partition $\mathbb{T}^d \times \mathbb{T}^d$,*
(4) *$s = O(1/\nu)$.*

Proof. Note that for cells A, B of equal volume we have $d(A, B) = 0$ if and only if either $A = B$ or (the boundaries of) A and B touch. For a cell C of level ℓ we let $\text{par}(C)$ be its *parent*, i.e., the unique cell of level $\ell - 1$ that C is contained in. Let $\mu = \lceil \nu \rceil_{2^d}$. We define $\mathcal{P}_\nu$ as follows. For any pair of cells (A, B) with $\text{VOL}(A) = \text{VOL}(B) \geq \nu$, we add (A, B) to $\mathcal{P}_\nu$ if either (i) $\text{VOL}(A) = \text{VOL}(B) = \mu$ and $d(A, B) = 0$, or (ii) $d(A, B) > 0$ and $d(\text{par}(A), \text{par}(B)) = 0$.

Property (1) follows by definition. Regarding property (2), the pairs (A, B) added in case (i) are clearly of type I. Observe that two cells A, B of equal volume that are not equal or touching have distance at least the sidelength of A, which is $\text{VOL}(A)^{1/d}$. Thus, in case (ii) the lower bound $d(A, B) > 0$ implies $d(A, B) \geq \text{VOL}(A)^{1/d}$, so that (A, B) is of type II.

For property (3), consider $(x, y) \in \mathbb{T}^d \times \mathbb{T}^d$ and let A, B be the cells of volume μ containing x, y. Let $A^{(0)} := A$ and $A^{(i)} := \text{par}(A^{(i-1)})$ for any $i \geq 1$, until $A^{(k)} = \mathbb{T}^d$. Similarly, define $B = B^{(0)} \subset \ldots \subset B^{(k)} = \mathbb{T}^d$ and note that $\text{VOL}(A^{(i)}) = \text{VOL}(B^{(i)})$. Observe that each set $A^{(i)} \times B^{(i)}$ contains (x, y). Moreover, any set $A' \times B'$, where A', B' are cells with $\text{VOL}(A') = \text{VOL}(B')$ and $(x, y) \in A' \times B'$, is of the form $A^{(i)} \times B^{(i)}$. Thus, to show that $\mathcal{P}_\nu$ partitions $\mathbb{T}^d \times \mathbb{T}^d$ we need to show that it contains exactly one of the pairs $(A^{(i)}, B^{(i)})$ (for any x, y). To show this, we use the monotonicity $d(A^{(i)}, B^{(i)}) \geq d(A^{(i+1)}, B^{(i+1)})$ and consider two cases. If $d(A, B) = 0$ then we add (A, B) to $\mathcal{P}_\nu$ in case (i), and we add no further $(A^{(i)}, B^{(i)})$, since $d(A^{(i)}, B^{(i)}) = 0$ for all i. If $d(A, B) > 0$ then since $d(A^{(k)}, B^{(k)}) = d(\mathbb{T}^d, \mathbb{T}^d) = 0$ there is a unique index $0 \leq i < k$ with $d(A^{(i)}, B^{(i)}) > 0$ and $d(A^{(i+1)}, B^{(i+1)}) = 0$. Then we add $(A^{(i)}, B^{(i)})$ in case (ii) and no further $(A^{(j)}, B^{(j)})$. This proves property (3).

Algorithm 1 Sampling algorithm for GIRGs in expected time $O(n)$

$E := \emptyset$
sample the positions x_v, $v \in V$, and determine the weight layers V_i
for all $1 \le i \le L$ **do** build data structure $\mathcal{D}_{\nu(i)}(\{\mathsf{x}_v \mid v \in V_i\})$ with $\nu(i) := \frac{w_i w_0}{\mathsf{W}}$
for all $1 \le i \le j \le L$ **do**
    construct partitioning $\mathcal{P}_{\nu(i,j)}$ with $\nu(i,j) := \frac{w_i w_j}{\mathsf{W}}$
    for all $(A, B) \in \mathcal{P}_{\nu(i,j)}$ of type I **do**
        for all $u \in V_i^A$ and $v \in V_j^B$ **do** with probability p_{uv} add edge $\{u, v\}$ to E
    for all $(A, B) \in \mathcal{P}_{\nu(i,j)}$ of type II **do**
        $\bar{p} := \min\left\{c \cdot \frac{1}{d(A,B)^{\alpha d}} \cdot \left(\frac{w_i w_j}{\mathsf{W}}\right)^{\alpha}, 1\right\}$
        $r := \mathrm{Geo}(\bar{p})$
        while $r \le |V_i^A| \cdot |V_j^B|$ **do**
            determine the r-th pair (u, v) in $V_i^A \times V_j^B$
            with probability $p_{uv}/\bar{p}$ add edge $\{u, v\}$ to E
            $r := r + \mathrm{Geo}(\bar{p})$
    if $i = j$ **then** remove all edges with $u > v$ sampled in this iteration

Property (4) follows from the running time bound of $O(1/\nu)$, which we show in the following. Note that we can enumerate all $1/\mu = O(1/\nu)$ cells of volume μ, and all of the at most $3^d = O(1)$ touching cells of the same volume, in time $O(1/\nu)$, proving the running time bound for case (i). Moreover, we can enumerate all $2^{\ell \cdot d}$ cells C in level ℓ, together with all of the at most $3^d = O(1)$ touching cells C' in the same level. Then we can enumerate all $2^d = O(1)$ cells A that have C as parent as well as all $O(1)$ cells B that have C' as parent. This enumerates (a superset of) all possibilities of case (ii). Summing the running time $O(2^{\ell \cdot d})$ over all levels ℓ with volume $2^{-\ell \cdot d} \ge \nu$ yields a total running time of $O(1/\nu)$. ◀

Weight layers. We set $w_0 := \mathsf{w}_{\min}$ and $w_i := 2w_{i-1}$ for $i \ge 1$. This splits the vertex set $V = [n]$ into *weight layers* $V_i := \{v \in V \mid w_{i-1} \le v < w_i\}$ for $1 \le i \le L$ with $L = O(\log n)$. We write V_i^C for the restriction of weight layer V_i to cell C, $V_i^C := \{v \in V_i \mid \mathsf{x}_v \in C\}$.

Geometric random variates. For $0 < p \le 1$ we write $\mathrm{Geo}(p)$ for a geometric random variable, taking value $i \ge 1$ with probability $p(1-p)^{i-1}$. $\mathrm{Geo}(p)$ can be sampled in constant time using the simple formula $\left\lceil \frac{\log(R)}{\log(1-p)} \right\rceil$, where R is chosen uniformly at random in $(0, 1)$, see [30]. To evaluate this formula exactly in time $O(1)$ we need to assume the RealRAM model of computation. However, also on a bounded precision machine like the WordRAM $\mathrm{Geo}(p)$ can be sampled in expected time $O(1)$ [16].

3.2 The Algorithm

Given the model parameters, our Algorithm 1 samples the edge set E of a GIRG. To this end, we first sample all vertex positions x_v uniformly at random in $\mathbb{T}^d$. Given weights $w_1, \dots, w_n$ we can determine the weight layers V_i in linear time (we may use counting sort or bucket sort since there are only $L = O(\log n)$ layers). Then we build the data structure from Lemma 9 for the points in V_i setting $\nu = \nu(i) = \frac{w_i w_0}{\mathsf{W}}$, i.e., we build $\mathcal{D}_{\nu(i)}(\{\mathsf{x}_v \mid v \in V_i\})$ for each i. In the following, for each pair of weight layers V_i, V_j we sample the edges between V_i and V_j. To this end, we construct the partitioning $\mathcal{P}_{\nu(i,j)}$ from Lemma 10 with $\nu(i,j) = \frac{w_i w_j}{\mathsf{W}}$. Since

$\mathcal{P}_{\nu(i,j)}$ partitions $\mathbb{T}^d \times \mathbb{T}^d$, every pair of vertices $u \in V_i, v \in V_j$ satisfies $\mathsf{x}_u \in A, \mathsf{x}_v \in B$ for exactly one $(A,B) \in \mathcal{P}_{\nu(i,j)}$. Thus, we can iterate over all $(A,B) \in \mathcal{P}_{\nu(i,j)}$ and sample the edges between V_i^A and V_j^B.

If (A,B) is of type I, then we simply iterate over all vertices $u \in V_i^A$ and $v \in V_j^B$ and add the edge $\{u,v\}$ with probability p_{uv}; this is the trivial sampling algorithm. Note that we can efficiently enumerate V_i^A and V_j^B using the data structure $\mathcal{D}_{\nu(i)}(\{\mathsf{x}_v \mid v \in V_i\})$ that we constructed above.

If (A,B) is of type II, then the distance $\|x-y\|$ of any two points $x \in A, y \in B$ satisfies $d(A,B) \le \|x-y\| \le d(A,B) + \textsc{vol}(A)^{1/d} + \textsc{vol}(B)^{1/d} \le 3d(A,B)$, by the definition of type II. Thus, $\bar{p} = \min\left\{c \cdot \frac{1}{d(A,B)^{\alpha d}} \cdot \left(\frac{w_i w_j}{\mathsf{W}}\right)^{\alpha}, 1\right\}$ is an upper bound on the edge probability p_{uv} for any $u \in V_i^A, v \in V_j^B$, and it is a good upper bound since $d(A,B)$ is within a constant factor of $\|\mathsf{x}_u - \mathsf{x}_v\|$ and w_i, w_j are within constant factors of $\mathsf{w}_u, \mathsf{w}_v$. Now we first sample the set of edges $\bar{E}$ between V_i^A and V_j^B that we would obtain if all edge probabilities were equal to $\bar{p}$, i.e., any $(u,v) \in V_i^A \times V_j^B$ is in $\bar{E}$ independently with probability $\bar{p}$. From this set $\bar{E}$, we can then generate the set of edges with respect to the true edge probabilities p_{uv} by throwing a coin for each $\{u,v\} \in \bar{E}$ and letting it survive with probability $p_{uv}/\bar{p}$. Then in total we choose a pair (u,v) as an edge in E with probability $\bar{p} \cdot (p_{uv}/\bar{p}) = p_{uv}$, proving that we sample from the correct distribution. Note that here we used $p_{uv} \le \bar{p}$. It is left to show how to sample the "approximate" edge set $\bar{E}$. First note that the data structure $\mathcal{D}_\nu(\{\mathsf{x}_v \mid v \in V_i\})$ defines an ordering on V_i^A, and we can determine the ℓ-th element in this ordering in constant time, similarly for V_j^B. Using the lexicographic ordering, we obtain an ordering on $V_i^A \times V_j^B$ for which we can again determine the ℓ-th element in constant time. In this ordering, the first pair $(u,v) \in V_i^A \times V_j^B$ that is in $\bar{E}$ is geometrically distributed, according to $\mathrm{Geo}(\bar{p})$. Since geometric random variates can be generated in constant time, we can efficiently generate $\bar{E}$, specifically in time $O(1+|\bar{E}|)$.

Finally, the case $i=j$ is special. With the algorithm described above, for any $u,v \in V_i$ we sample whether they form an edge twice, once for $\mathsf{x}_u \in A, \mathsf{x}_v \in B$ (for some $(A,B) \in \mathcal{P}_{\nu(i,j)}$) and once for $\mathsf{x}_v \in A', \mathsf{x}_u \in B'$ (for some $(A',B') \in \mathcal{P}_{\nu(i,j)}$). To fix this issue, in the case $i=j$ we only accept a sampled edge $(u,v) \in V_i^A \times V_j^B$ if $u < v$; then only one way of sampling edge $\{u,v\}$ remains. This changes the expected running time only by a constant factor.

3.3 Analysis

Correctness of our algorithm follows immediately from the above explanations. In the following we show that Algorithm 1 runs in expected linear time. This is clear for lines 1-2. For line 3, since building the data structure from Lemma 9 takes time $O(|P| + 1/\nu)$, it takes total time $\sum_{i=1}^{L} O\big(|V_i| + \mathsf{W}/(w_i w_0)\big)$. Clearly, the first summand $|V_i|$ sums up to n. Using $w_0 = \mathsf{w}_{\min} = \Omega(1)$, $\mathsf{W} = O(n)$, and that w_i grows exponentially with i, implying $\sum_i 1/w_i = O(1)$, also the second summand sums up to $O(n)$. For line 5, all invocations in total take time $O\big(\sum_{i,j} \mathsf{W}/(w_i w_j)\big)$, which is $O(n)$, since again $\mathsf{W} = O(n)$ and $\sum_i 1/w_i = O(1)$. We claim that for any weight layers V_i, V_j the expected running time we spend on any $(A,B) \in \mathcal{P}_{\nu(i,j)}$ is $O(1 + \mathbb{E}[|E_{i,j}^{A,B}|])$, where $E_{i,j}^{A,B}$ is the set of edges in $V_i^A \times V_j^B$. Summing up the first summand $O(1)$ over all $(A,B) \in \mathcal{P}_{\nu(i,j)}$ sums up to $1/\nu(i,j) = \mathsf{W}/(w_i w_j)$. As we have seen above, this sums up to $O(n)$ over all i,j. Summing up the second summand $O(\mathbb{E}[|E_{i,j}^{A,B}|])$ over all $(A,B) \in \mathcal{P}_{\nu(i,j)}$ and weight layers V_i, V_j yields the total expected number of edges $O(\mathbb{E}[|E|])$, which is $O(n)$, since the average weight $\mathsf{W}/n = O(1)$ and thus the expected average degree is constant.

It is left to prove the claim that for any weight layers V_i, V_j the expected time spent on $(A,B) \in \mathcal{P}_{\nu(i,j)}$ is $O(1 + \mathbb{E}[|E_{i,j}^{A,B}|])$. If (A,B) is of type I, then any pair of vertices $(u,v) \in$

$V_i^A \times V_j^B$ has probability $\Theta(1)$ to form an edge: Since the volume of A and B is $w_i w_j/\mathsf{W}$, their diameter is $(w_i w_j/\mathsf{W})^{1/d}$ and we obtain $\|\mathsf{x}_u - \mathsf{x}_v\| \leq (w_i w_j/\mathsf{W})^{1/d} = O((\mathsf{w}_u \mathsf{w}_v/\mathsf{W})^{1/d})$, which yields $p_{uv} = \Theta\big(\min\big\{\big(\frac{\mathsf{w}_u \mathsf{w}_v}{\|\mathsf{x}_u - \mathsf{x}_v\|^d \mathsf{W}}\big)^\alpha, 1\big\}\big) = \Theta(1)$. As we spend time $O(1)$ for any $(u,v) \in V_i^A \times V_j^B$, we stay in the desired running time bound $O(\mathbb{E}[|E_{i,j}^{A,B}|])$.

If (A,B) is of type II, we first sample edges $\bar{E}$ with respect to the larger edge probability $\bar{p}$, and then for each edge $e \in \bar{E}$ sample whether it belongs to E. This takes total time $O(1+|\bar{E}|)$. Note that any edge $e \in \bar{E}$ has constant probability $p_{uv}/\bar{p} = \Theta(1)$ to survive: It follows from $\mathsf{w}_u = \Theta(w_i), \mathsf{w}_v = \Theta(w_j)$, and $\|\mathsf{x}_u - \mathsf{x}_v\| = \Theta(d(A,B))$ that $p_{uv} = \Theta(\bar{p})$. Hence, we obtain $\mathbb{E}[|\bar{E}|] = O(\mathbb{E}[|E_{i,j}^{A,B}|])$, and the running time $O(1+|\bar{E}|)$ is in expectation bounded by $O(1+\mathbb{E}[|E_{i,j}^{A,B}|])$. This finishes the proof of the claim.

4 Stability of the Giant, Entropy, and Compression Algorithm

In this section we prove Theorems 5 and 6. More precisely, we show that w.h.p. the graph (and its giant) has separators of sublinear size, and we make use of these small separators to devise a compression algorithm that can store the graph using a linear number of bits in expectation. Note that the compression maintains only the graph up to isomorphism, not the underlying geometry. The main idea is to enumerate the vertices in an ordering that reflects the geometry, and then storing for each vertex i the differences $i-j$ for all neighbors j of i. We start with a technical lemma that gives the number of edges intersecting an axis-parallel, regular grid. (For $\gamma > 0$ with $1/\gamma \in \mathbb{N}$, the axis-parallel, regular grid with side length γ is the union of all $d-1$-dimensional hyperplanes that are orthogonal to an axis and that are in distance $k\gamma$ from the origin for a $k \in \mathbb{Z}$.) Both the existence of small separators and the efficiency of the compression algorithm follow easily from that formula. For detailed proofs we refer to the full version [18].

▶ **Lemma 11.** *Let $\eta > 0$. Let $1 \leq \mu \leq n^{1/d}$ be an integer, and consider an axis-parallel, regular grid with side length $1/\mu$ on $\mathbb{T}^d$. Then in expectation the grid intersects at most $O(n \cdot (n/\mu^d)^{2-\beta+\eta} + (n^{2-\alpha}\mu^{d(\alpha-1)} + n^{1-1/d}\mu)(1+\log(n/\mu^d)))$ edges.*

Proof Outline. For $u, v \in V$, let ρ_{uv} be the probability that the edge uv exists and cuts the grid. Let $r_{\max} := 1/2$ be the diameter of $\mathbb{T}^d$. We write

$$\rho_{uv} = \int_0^{r_{\max}} \Pr[\|\mathsf{x}_u - \mathsf{x}_v\| = r] \cdot p_{uv}(r) \cdot \Pr[\mathsf{x}_u, \mathsf{x}_v \text{ in different cells of } \mu\text{-grid}]\, dr. \tag{1}$$

Observe that u and v have distance r with probability density $\Pr[\|\mathsf{x}_u - \mathsf{x}_v\| = r] = O(r^{d-1})$. For $p_{uv}(r)$ we plug in the bound from (EP1) or (EP2), respectively. Finally, using symmetry of $\mathbb{T}^d$ we can show that the last probability in (1) is bounded by $O(\min\{\mu r, 1\})$. The remainder of the proof is a straight-forward, yet technical calculation of the integral. ◀

Compression algorithm. We remark that our Theorem 6 does not directly follow from the general compression scheme on graphs with small separators in [5], since our graphs only have small separators in expectation, in particular, small subgraphs of size $O(\sqrt{\log n})$ can form expanders and thus not have small separators. However, our algorithm loosely follows their algorithm as well as the practical compression scheme of [12], see also [22].

We first enumerate the vertices as follows. Recall the definition of cells from Section 2.4, and consider all cells of level $\ell_0 := \lfloor \log n/d \rfloor$. Note that the boundaries of these cells induce a grid as in Lemma 11. Since each such cell has volume $\Theta(1/n)$, the expected number of vertices in each cell is constant. We fix a geometric ordering of these cells as in Lemma 8,

Algorithm 2 Finding the s-th neighbor of vertex i

```
b := SELECT(i, B_V)                    ▷ starting position of vertex i
k := RANK(b, B_E)                      ▷ number of edges and vertices before b
b_1 := SELECT(k + s, B_E)              ▷ starting position of s-th edge of vertex i
b_2 := SELECT(k + s + 1, B_E)          ▷ bit after ending position of s-th edge of vertex i
return B[b_1 : b_2 − 1]                ▷ block that stores s-th edge of vertex i
```

and we enumerate the vertices in the order of the cells, breaking ties (between vertices in the same cell) arbitrarily. From now on we assume that the vertices are enumerated in this way, i.e., we identify $V = [n]$, where $i \in [n]$ refers to the vertex with index i.

Having enumerated the vertices, for each vertex $i \in [n]$ we store a block of $1 + \deg(i)$ sub-blocks. The first sub-block consists of a single dummy bit (to avoid empty sequences arising from isolated vertices). In the other $\deg(i)$ sub-blocks we store the differences $i - j$ using $\log_2 |i - j| + O(1)$ bits, where j runs through all neighbors of i. We assume that the information for all vertices is stored in a successive block B in the memory. Moreover, we create two more blocks B_V and B_E of the same length. Both B_V and B_E have a one-bit whenever the corresponding bit in B is the first bit of the block of a vertex, and B_E has also a one-bit whenever the corresponding bit in B is the first bit of an edge (i.e., the first bit encoding a difference $i - j$). All other bits in B_V and B_E are zero.

It is clear that with the data above the graph is determined. To handle queries efficiently, we replace B_V and B_E each with a rank/select data structure. This data structure allows to handle in constant time queries of the form "Rank(b)", which returns the number of one-bits up to position b, and "Select(i)", which returns the position of the i-th one-bit [36, 27, 43]. Given $i, s \in \mathbb{N}$, we can find the index of the s-th neighbor of i in constant time by Algorithm 2. Note that we can also compute $\deg(i)$ in constant time as $\textsc{Rank}(b_{i+1}, B_E) - \textsc{Rank}(b_i, B_E) - 1$, where $b_i = \textsc{Select}(i, B_V)$ and $b_{i+1} = \textsc{Select}(i + 1, B_V)$ are the starting positions of vertex i and $i + 1$, respectively. In particular, it is possible for Algorithm 2 to first check whether $s \leq \deg(i)$.

We need to show that the data structure needs $O(n)$ bits in expectation. There are n dummy bits, so we must show that we require $O(n)$ bits to store all differences $i - j$, where ij runs through all edges of the graph. We need $\log_2 |i - j| + O(1)$ bits for each edge, and the $O(1)$ terms sum up to $O(|E|)$, which is $O(n)$ in expectation. Thus, it remains to prove the following.

▶ **Lemma 12.** *If V is enumerated geometrically, then $\mathbb{E}[\sum_{ij \in E} \log(|i - j|)] = O(n)$.*

Proof outline. The geometric ordering puts all the vertices that are in the same cell of a $2^{-\ell}$-grid in a consecutive block, for all $1 \leq \ell \leq \ell_0$. Therefore, if $e = ij$ does not intersect the $2^{-\ell}$-grid then $|i - j| \leq \#\{\text{vertices in the } 2^{-\ell}\text{-cell of } e\} \approx n2^{-\ell}$. Hence, if $E_{\ell+1}$ is the number of edges intersecting the $2^{-\ell}$-grid, but not the $2^{-\ell-1}$-grid, then $\mathbb{E}[\sum_{ij \in E} \log(|i - j|)] \approx \sum_{\ell=0}^{\ell_0} \mathbb{E}[E_{\ell+1}] \log(n2^{-\ell})$, and we show that the latter term is $O(n)$ using Lemma 11. ◀

Acknowledgements. We thank Hafsteinn Einarsson, Tobias Friedrich, and Anton Krohmer for helpful discussions.

References

1 M. A. Abdullah, M. Bode, and N. Fountoulakis. Typical distances in a geometric model for complex networks. *Preprint available at arXiv:1506.07811*, 2015.

2 W. Aiello, A. Bonato, C. Cooper, J. Janssen, and P. Prałat. A spatial web graph model with local influence regions. *Internet Mathematics*, 5(1-2):175–196, 2008.

3 A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

4 V. Batagelj and U. Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

5 D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *Proceedings of the 14 annual ACM-SIAM Symposium on Discrete algorithms (SODA)*, pages 679–688. SIAM, 2003.

6 T. Bläsius, T. Friedrich, and A. Krohmer. Hyperbolic random graphs: Separators and treewidth. In *European Symposium on Algorithms (ESA)*, pages 15:1–15:16, 2016.

7 T. Bläsius, T. Friedrich, and A. Krohmer. Cliques in hyperbolic random graphs. *Algorithmica*, 2017.

8 T. Bläsius, T. Friedrich, A. Krohmer, and S. Laue. Efficient embedding of scale-free graphs in the hyperbolic plane. In *European Symposium on Algorithms (ESA)*, pages 16:1–16:18, 2016.

9 M. Bode, N. Fountoulakis, and T. Müller. On the giant component of random hyperbolic graphs. In *7th European Conference on Combinatorics, Graph Theory and Applications (EUROCOMB)*, pages 425–429. Springer, 2013.

10 M. Bode, N. Fountoulakis, and T. Müller. On the geometrisation of the Chung-Lu model and its component structure. *Preprint*, 2014.

11 M. Boguñá, F. Papadopoulos, and D. Krioukov. Sustaining the Internet with hyperbolic mapping. *Nature Communications*, 1(6), September 2010.

12 P. Boldi and S. Vigna. The WebGraph framework I : compression techniques. In *13th International Conference on World Wide Web (WWW)*, pages 595–602, 2004.

13 B. Bollobás, S. Janson, and O. Riordan. The phase transition in inhomogeneous random graphs. *Random Structures & Algorithms*, 31(1):3–122, 2007.

14 A. Bonato, J. Janssen, and P. Prałat. A geometric model for on-line social networks. In *1st International Workshop on Modeling Social Media (WOSM)*, 2010.

15 M. Bradonjić, A. Hagberg, and A. G. Percus. The structure of geographical threshold graphs. *Internet Mathematics*, 5(1-2):113–139, 2008.

16 K. Bringmann and T. Friedrich. Exact and efficient generation of geometric random variates and random graphs. In *40th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 267–278, 2013.

17 K. Bringmann, R. Keusch, and J. Lengler. Average distance in a general class of scale-free networks with underlying geometry. *Preprint avaiable at arxiv:1602.05712*, 2016.

18 K. Bringmann, R. Keusch, and J. Lengler. Sampling geometric inhomogeneous random graphs in linear time. *Preprint available at arXiv:1511.00576*, 2016.

19 K. Bringmann, R. Keusch, J. Lengler, Y. Maus, and A. Molla. Greedy routing and the algorithmic small-world phenomenon. *Preprint available at arXiv:1612.05539*, 2017.

20 E. Candellero and N. Fountoulakis. Clustering and the hyperbolic geometry of complex networks. In *11th International Workshop on Algorithms and Models for the Web Graph (WAW)*, pages 1–12, 2014.

21 E. Candellero and N. Fountoulakis. Bootstrap percolation and the geometry of complex networks. *Stochastic Processes and their Applications*, 126:234–264, 2016.

22 F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *15th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 219–228, 2009.

23 F. Chierichetti, R. Kumar, S. Lattanzi, A. Panconesi, and P. Raghavan. Models for the Compressible Web. In *50th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 331–340, 2009.

24 F. Chung and L. Lu. The average distances in random graphs with given expected degrees. *Proceedings of the National Academy of Sciences (PNAS)*, 99(25):15879–15882, 2002.

25 F. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *Annals of Combinatorics*, 6(2):125–145, 2002.

26 F. Chung and L. Lu. The average distance in a random graph with given expected degrees. *Internet Mathematics*, 1(1):91–113, 2004.

27 D. R. Clark and I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7 annual ACM-SIAM Symposium on Discrete algorithms (SODA)*, pages 383–391. SIAM, 1996.

28 M. Deijfen, R. van der Hofstad, and G. Hooghiemstra. Scale-free percolation. *Annales de l'Institut Henri Poincaré, Probabilités et Statistiques*, 49(3):817–838, 2013.

29 P. Deprez, R. S. Hazra, and M. V. Wüthrich. Inhomogeneous long-range percolation for real-life network modeling. *Risks*, 3(1), 2015.

30 L. Devroye. *Nonuniform random variate generation*. Springer, New York, 1986.

31 S. N. Dorogovtsev and J. F. F. Mendes. Evolution of networks. *Advances in Physics*, 51(4):1079–1187, 2002.

32 T. Friedrich and A. Krohmer. On the diameter of hyperbolic random graphs. In *42nd International Colloquium on Automata, Languages, and Programming (ICALP)*, Lecture Notes in Computer Science, 2015.

33 L. Gugelmann, K. Panagiotou, and U. Peter. Random hyperbolic graphs: degree sequence and clustering. In *39th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 573–585, 2012.

34 M. Heydenreich, T. Hulshof, and J. Jorritsma. Structures in supercritical scale-free percolation. *Preprint available at arXiv:1604.08180*, 2016.

35 E. Jacob and P. Mörters. A spatial preferential attachment model with local clustering. In *Algorithms and Models for the Web Graph*, pages 14–25. Springer, 2013.

36 G. Jacobson. Space-efficient static trees and graphs. In *30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

37 M. Kiwi and D. Mitsche. A bound for the diameter of random hyperbolic graphs. In *2015 Proceedings of the Twelfth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 26–39. SIAM, 2015.

38 M. Kiwi and D. Mitsche. Spectral gap of random hyperbolic graphs and related parameters. *Preprint available at arXiv:1606.02240*, 2016.

39 C. Koch and J. Lengler. Bootstrap percolation on geometric inhomogeneous random graphs. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 147:1–147:15, 2016.

40 J. C. Miller and A. Hagberg. Efficient generation of networks with given expected degrees. In *8th International Conference on Algorithms and Models for the Web Graph (WAW)*, pages 115–126, 2011.

41 I. Norros and H. Reittu. On a conditionally Poissonian graph process. *Advances in Applied Probability*, 38(1):59–75, 2006.

42 F. Papadopoulos, D. Krioukov, M. Boguñá, and A. Vahdat. Greedy forwarding in dynamic scale-free networks embedded in hyperbolic metric spaces. In *INFOCOM 2010. 29th IEEE International Conference on Computer Communications*, pages 1–9, March 2010.

43 M. Pătraşcu. Succincter. In *49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.

44 M. Penrose. *Random geometric graphs*, volume 5. Oxford University Press Oxford, 2003.

45 U. Peter. *Random Graph Models for Complex Systems*. PhD thesis, ETH Zurich, 2014.

46 R. van der Hofstad and J. Komjathy. Explosion and distances in scale-free percolation. *arXiv preprint arXiv:1706.02597*, 2017.

47 M. Von Looz, C. L. Staudt, H. Meyerhenke, and R. Prutkin. Fast generation of dynamic complex networks with underlying hyperbolic geometry. *Preprint available at arXiv:1501.03545*, 2015.

Cache Oblivious Algorithms for Computing the Triplet Distance Between Trees*†

Gerth Stølting Brodal[1] and Konstantinos Mampentzidis[2]

1 **Department of Computer Science, Aarhus University, Aarhus, Denmark**
gerth@cs.au.dk
2 **Department of Computer Science, Aarhus University, Aarhus, Denmark**
kmampent@cs.au.dk

Abstract

We study the problem of computing the triplet distance between two rooted unordered trees with n labeled leafs. Introduced by Dobson 1975, the triplet distance is the number of leaf triples that induce different topologies in the two trees. The current theoretically best algorithm is an $O(n \log n)$ time algorithm by Brodal *et al.* (SODA 2013). Recently Jansson *et al.* proposed a new algorithm that, while slower in theory, requiring $O(n \log^3 n)$ time, in practice it outperforms the theoretically faster $O(n \log n)$ algorithm. Both algorithms do not scale to external memory.

We present two cache oblivious algorithms that combine the best of both worlds. The first algorithm is for the case when the two input trees are binary trees and the second a generalized algorithm for two input trees of arbitrary degree. Analyzed in the RAM model, both algorithms require $O(n \log n)$ time, and in the cache oblivious model $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os. Their relative simplicity and the fact that they scale to external memory makes them achieve the best practical performance. We note that these are the first algorithms that scale to external memory, both in theory and practice, for this problem.

1998 ACM Subject Classification G.2.2 Trees, G.2.1 Combinatorial Algorithms

Keywords and phrases Phylogenetic tree, tree comparison, triplet distance, cache oblivious algorithm

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.21

1 Introduction

Background. Trees are data structures that are often used to represent relationships. For example in the field of Biology, a tree can be used to represent evolutionary relationships, with the leafs corresponding to species that exist today, and internal nodes to ancestor species that existed in the past. For a fixed set of n species, different data or construction methods (e.g. Q* [2], neighbor joining [13]) can lead to trees that look structurally different. An interesting question that arises then is, given two trees T_1 and T_2 over n species, how different are they? An answer to this question could potentially be used to determine whether the difference is statistically significant or not, which in turn could help with evolutionary inferences. Several ways of comparing two trees have been proposed in the past, with different types of trees (e.g. rooted versus unrooted, binary versus arbitrary degree) having different distance measures (e.g. Robinson-Foulds distance [12], triplet distance [6], quartet distance [7]). In this paper we focus on the triplet distance computation, which is defined for rooted trees.

* Research supported by the Danish National Research Foundation, grant DNRF84, Center for Massive Data Algorithmics (MADALGO).
† An extended version of the paper is available on arXiv [4].

(a) $xy|z$ **(b)** $xz|y$ **(c)** $yz|x$ **(d)** xyz

Figure 1 Triplet topologies.

Problem Definition. For a given rooted unordered tree T where each leaf has a unique label, a *triplet* is defined by a set of three leaf labels x, y and z and their induced topology in T. The four possible topologies are illustrated in Figure 1. For two such trees T_1 and T_2 that are built on n identical leaf labels, the *triplet distance* $D(T_1, T_2)$ is the number of triplets that are different in T_1 and T_2. Let $S(T_1, T_2)$ be the number of *shared* triplets in the two trees, i.e. leaf triples with identical topologies in the two trees. We have the relationship that $D(T_1, T_2) + S(T_1, T_2) = \binom{n}{3}$.

Results. All related work can be found in [5, 1, 14, 3, 15, 9, 10, 11]. Previous and new results are shown in the table below. For the cache oblivious model [8], the papers [5, 1, 14, 3, 10, 11] do not provide an analysis, so here we provide an upper bound.

Year	Reference	Time	IOs	Space	Non-Binary Trees
1996	Critchlow *et al.* [5]	$O(n^2)$	$O(n^2)$	$O(n^2)$	no
2011	Bansal *et al.* [1]	$O(n^2)$	$O(n^2)$	$O(n^2)$	yes
2013	Brodal *et al.* [14]	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n)$	no
2013	Brodal *et al.* [3]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	yes
2015	Jansson *et al.* [10, 11]	$O(n \log^3 n)$	$O(n \log^3 n)$	$O(n \log n)$	yes
2017	new	$O(n \log n)$	$O(\frac{n}{B} \log_2 \frac{n}{M})$	$O(n)$	yes

The common main bottleneck with all previous approaches is that the data structures used rely intensively on $\Omega(n \log n)$ random memory accesses. This means that all algorithms are penalized by cache performance and thus do not scale to external memory. We address this limitation by proposing new algorithms for computing the triplet distance on binary and non-binary trees, that match the previous best $O(n \log n)$ time and $O(n)$ space bounds in the RAM model, but for the first time also scale to external memory. More specifically, in the cache oblivious model, the total number of I/Os required is $O(\frac{n}{B} \log_2 \frac{n}{M})$. The basic idea is to essentially replace the dependency of random access to data structures by scanning contracted versions of the input trees. A careful implementation of the algorithms is shown to achieve the best practical performance, thus essentially documenting that the theoretical results carry over to practice.

2 Previous Approaches

A naive algorithm would enumerate over all $\binom{n}{3}$ sets of 3 labels and find for each set whether the induced topologies in T_1 and T_2 differ or not, giving an $O(n^3)$ algorithm. This naive approach does not exploit the fact that the triplets are not completely independent. For example the triplets $xy|z$ and $yx|u$ share the leafs x and y and the fact that the lowest common ancestor of x and y is at a lower depth than the lowest common ancestor of z with either x or y and the lowest common ancestor of u with either x or y. Dependencies like this can be exploited to count the number of shared triplets faster.

Critchlow *et al.* [5] exploit the depth of the leafs' ancestors to achieve the first improvement over the naive approach. Bansal *et al.* [1] exploit the shared leafs between subtrees and reduce the problem to computing the intersection size (number of shared leafs) of all pairs of subtrees, one from T_1 and one from T_2, which can be solved with dynamic programming.

The $O(n^2)$ Algorithm for Binary Trees in [14]. The algorithm for binary trees in [14] is the basis for all subsequent improvements [14, 3, 10], including ours as well, so we will describe it in more detail here. The dependency that was exploited is the same as in [1], but the procedure for counting the shared triplets is completely different. More specifically, each triplet in T_1 and T_2, defined by the leafs i, j and k, is implicitly *anchored* in the lowest common ancestor of i, j and k. For a node u in T_1 and v in T_2, let $s(u)$ and $s(v)$ be the set of triplets that are anchored in u and v respectively. For the number of shared triplets $S(T_1, T_2)$ we then have that

$$S(T_1, T_2) = \sum_{u \in T_1} \sum_{v \in T_2} |s(u) \cap s(v)| \ .$$

For the algorithm to be $O(n^2)$ the value $|s(u) \cap s(v)|$ must be computed in $O(1)$ time. This is achieved by a leaf colouring procedure as follows: Fix a node u in T_1 and color the leafs in the left subtree of u *red*, the leafs in the right subtree of u *blue*, let every other leaf have no color and then transfer this coloring to the leafs in T_2, i.e. identically labelled leafs get the same color. To compute $|s(u) \cap s(v)|$ we do as follows: let l and r be the left and right children of v, and let $w_{\texttt{red}}$ and $w_{\texttt{blue}}$ be the number of red and blue leafs in a subtree rooted at a node w in T_2. We then have that

$$|s(u) \cap s(v)| = \binom{l_{\texttt{red}}}{2} r_{\texttt{blue}} + \binom{l_{\texttt{blue}}}{2} r_{\texttt{red}} + \binom{r_{\texttt{red}}}{2} l_{\texttt{blue}} + \binom{r_{\texttt{blue}}}{2} l_{\texttt{red}} \ . \tag{1}$$

Subquadratic Algorithms. To reduce the time, Brodal *et al.* [14] applied the *smaller half trick*, which specifies a depth first order to visit the nodes u of T_1, so that each leaf in T_1 changes color at most $O(\log n)$ times. To count shared triplets efficiently without scanning T_2 completely for each node u in T_1, the tree T_2 is stored in a data structure denoted a *hierarchical decomposition tree* (*HDT*). This HDT maintains for the current visited node u in T_1, according to (1) the sum $\sum_{v \in T_2} |s(u) \cap s(v)|$, so that each color change in T_1 can be updated efficiently in T_2. In [14] the HDT is a binary tree of height $O(\log n)$ and every update can be done in a leaf to root path traversal in the HDT, which in total gives $O(n \log^2 n)$ time. In [3] the HDT is generalized to also handle non-binary trees, each query operates the same, and now due to a contraction scheme of the HDT the total time is reduced to $O(n \log n)$. Finally, in [10] as an HDT the so called *heavy-light tree decomposition* is used. Note that the only difference in all $O(n \operatorname{polylog} n)$ results that are available until now is the type of HDT used.

In terms of external memory efficiency, every $O(n \operatorname{polylog} n)$ algorithm performs $\Theta(n \log n)$ updates to an HDT data structure, which means that for sufficiently large input trees every algorithm requires $\Omega(n \log n)$ I/Os.

3 The New Algorithm for Binary Trees

Overview. We will use the $O(n^2)$ algorithm described in Section 2 as a basis. The main difference lies in the order that we visit the nodes of T_1 and how we process T_2 when we count. We propose a new order of visiting the nodes of T_1, which we find by applying a

hierarchical decomposition on T_1. Every component in this decomposition corresponds to a connected part of T_1 and a contracted version of T_2. In simple terms, if Λ is the set of leafs in a component of T_1, the contracted version of T_2 is a binary tree on Λ that preserves the topologies induced by Λ in T_2 and has size $O(|\Lambda|)$. To count shared triplets, every component of T_1 has a representative node u that we use to scan the corresponding contracted version of T_2 in order to find $\sum_{v \in T_2} |s(u) \cap s(v)|$. Unlike previous algorithms, we do not store T_2 in a data structure. We process T_2 by contracting and counting, both of which can be done by scanning. At the same time, even though we apply a hierarchical decomposition on T_1, the only reason why we do so, is so we can find the order in which to visit the nodes of T_1. This means that we do not need to store T_1 in a data structure either. Thus, we completely remove the need of data structures (and thereby random memory accesses) and scanning becomes the basic primitive in the algorithm. To make our algorithm I/O efficient, all that remains to be done is to use a proper layout to store the trees in memory, so that every time we scan a tree of size s we spend $O(s/B)$ I/Os.

Preprocessing. As a preprocessing step, first we make T_1 *left heavy*, by swapping children so that for every node u in T_1 the left subtree is larger than the right subtree, by a depth first traversal. Second, we change the leaf labels of T_1, which can also be done by a depth first traversal of T_1, so that the leafs are numbered 1 to n from left to right. This step takes $O(n)$ time in the RAM model. The second step is done to simplify the process of transferring the leaf colors between T_1 and T_2. The coloring of a subtree in T_1 will correspond to assigning the same color to a contiguous range of leaf labels. Determining the color of a leaf in T_2 will then require one `if-statement` to find in what range (red or blue) its label belongs to.

Centroid Decomposition. For a given rooted binary tree T we let $|T|$ denote the number of nodes in T (internal nodes and leafs). For a node u in T we let l and r be the left and right children of u, and p the parent. Removing u from T partitions T into three (possibly empty) *connected components* T_l, T_r and T_p containing l, r and p, respectively. A *centroid* is a node u in T such that $\max\{|T_l|, |T_r|, |T_p|\} \leq |T|/2$. A centroid always exists and can be found by starting from the root of T and iteratively visiting the child with a largest subtree, eventually we will reach a centroid. Finding the size of every subtree and identifying u takes $O(|T|)$ time in the RAM model. By recursively finding centroids in each of the three components, we will in the end get a ternary tree of centroids, which is called the *centroid decomposition* of T, denoted $CD(T)$. We can generate a level of $CD(T)$ in $O(|T|)$ time, given the decomposition of T into components by the previous level. Since we have to generate at most $1 + \log_2(|T|)$ levels, the total time required to build $CD(T)$ is $O(|T| \log |T|)$, hence we get Lemma 1.

▶ **Lemma 1.** *For any rooted binary tree T with n leafs, building $CD(T)$ takes $O(n \log n)$ time in the RAM model.*

A component in a centroid decomposition $CD(T)$, might have many edges crossing its boundaries (connecting nodes inside and outside the component). The below *modified centroid decomposition*, denoted $MCD(T)$, generates components with at most two edges crossing the boundary, one going towards the root and one down to exactly one subtree.

Modified Centroid Decomposition. An $MCD(T)$ is built recursively as follows: If a component C has no edge from below, we select the centroid c of C as a splitting node as described above. Otherwise, let (x, y) be the edge that crosses the boundary from below,

where x is in C and let c be centroid of C. As a splitting node choose the lowest common ancestor of x and c. By induction every component has at most one edge from below and one edge from above. A useful property of $MCD(T)$ is captured by the following lemma:

▶ **Lemma 2.** *For any rooted binary tree T with n leafs, we have that $h(MCD(T)) \leq 2+2\log_2 n$, where $h(MCD(T))$ denotes the height of $MCD(T)$.*

Since each level of $MCD(T)$ can be constructed in $O(n)$ time, we have

▶ **Theorem 3.** *For any rooted binary tree T with n leafs, building $MCD(T)$ takes $O(n \log n)$ time in the RAM model.*

To return to our original problem, we visit the nodes of T_1, given by the depth first traversal of the ternary tree $MCD(T_1)$, where the children of every node u in $MCD(T_1)$ are visited from left to right. For every such node u we process T_2 in two phases, the *contraction* phase and the *counting* phase.

Contraction. Let $L(T_2)$ denote the set of leafs in T_2 and $\Lambda \subseteq L(T_2)$. In the contraction phase, T_2 is compressed into a binary tree of size $O(|\Lambda|)$ whose leaf set is Λ. The contraction is done in a way so that all the topologies induced by Λ in T_2 are preserved in the compressed binary tree. This is achieved by the following three sequential steps: prune all leafs of T_2 that are not in Λ, repeatedly prune all internal nodes of T_2 with no children and repeatedly contract unary internal nodes, i.e. nodes having exactly one child.

Let u be a node of $MCD(T_1)$ and C_u the corresponding component of T_1. For every such node u we have a contracted version of T_2, from now on referred to as $T_2(u)$, where $L(T_2(u)) = L(C_u)$. The goal is to augment $T_2(u)$ with counters (see counting phase below), so that we can find $\sum_{v \in T_2} |s(u) \cap s(v)|$ by scanning $T_2(u)$. One can imagine $MCD(T_1)$ as being a tree where each node u is augmented with $T_2(u)$. To generate all contractions of T_2 for level i of $MCD(T_1)$, which correspond to a set of disjoint connected components in T_1, we can reuse the contractions of T_2 at level $i-1$ in $MCD(T_1)$. This means that we have to spend $O(n)$ time to generate the contractions of level i, so to generate all contractions of T_2 we need $O(n \log n)$ time. Note that by explicitly storing all contractions, we will also need to use $O(n \log n)$ space. For our problem, we traverse $MCD(T_1)$ in a depth first manner, so we only have to store a stack of contractions corresponding to the stack of nodes of $MCD(T_1)$ that we have to remember during our traversal. Since the components at every second level of $MCD(T_1)$ have at most half the size of the components two levels above, Lemma 4 states that the size of this stack is always $O(n)$.

▶ **Lemma 4.** *Let T_1 and T_2 be two rooted binary trees with n leafs and $u_1, u_2, ..., u_k$ a root to leaf path of $MCD(T_1)$. For the corresponding contracted versions $T_2(u_1), T_2(u_2), ..., T_2(u_k)$ we have that $\sum_{i=1}^{k} |T_2(u_i)| = O(n)$.*

Counting. In the counting phase, we find $\sum_{v \in T_2} |s(u) \cap s(v)|$ by scanning $T_2(u)$ instead of T_2. This makes the total time of the algorithm in the RAM model $O(n \log n)$. We consider the following two cases:

- *C_u has no edges from below.*
  In this case C_u corresponds to a complete subtree of T_1. We act exactly like in the $O(n^2)$ algorithm (Section 2) but now instead of scanning T_2 we scan $T_2(u)$.
- *C_u has one edge from below.*
  In this case C_u does not correspond to a complete subtree of T_1, since the edge from below C_u, will point to a subtree X_u, that is located outside of C_u (for an illustration of

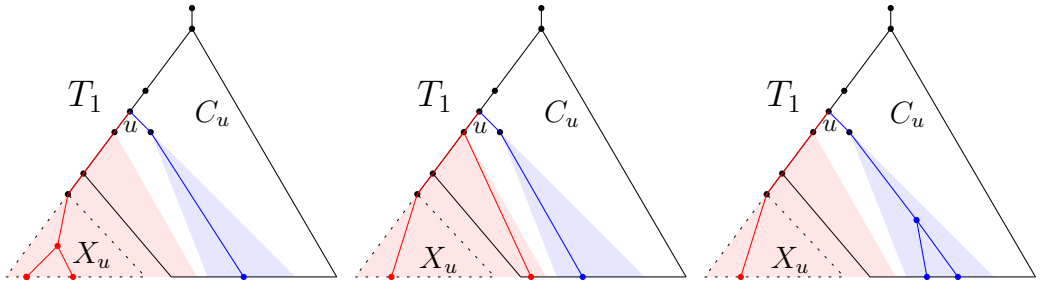


Figure 2 $MCD(T_1)$: Triplets that can be anchored in u with the leafs not being in the component C_u.

this case see Figure 2). Note that because T_1 is left heavy, X_u is always rooted in a node on the left most path from u. The leafs in X_u are important because they can be used to form triplets that are anchored in u. Acting in the same manner as in the previous case is not sufficient because we need to count the triplets involving X_u as well.
To address this problem, every edge (p_v, v) in $T_2(u)$ between a node v and its parent p_v, is augmented with some counters about the leafs from X_u that were contracted away in T_2. For every such edge (p_v, v), let $s_1, s_2, ..., s_k$ be the contracted subtrees rooted on the edge. Every such subtree contains either leafs with no color or leafs from X_u that have the color red (the color can not be blue because T_1 is left heavy). For every node v in $T_2(u)$ the counters that we have are the following:

- $v_{\texttt{red}}$: total number of red leafs in the subtree of v (including those coming from X_u).
- $v_{\texttt{blue}}$: total number of blue leafs in the subtree of v.
- v_{ts}: total number of red leafs in $s_1, s_2, ..., s_k$.
- v_{ps}: total number of pairs of red leafs in $s_1, s_2, ..., s_k$ such that each pair comes from the same contracted subtree, i.e. $\sum_{i=1}^{k} \binom{r_i}{2}$ where r_i is the number of red leafs in s_i.

The number of shared triplets that are anchored in a non-contracted node v of $T_2(v)$ can be found like in the $O(n^2)$ algorithm using the counters $v_{\texttt{red}}$ and $v_{\texttt{blue}}$ in (1). As for the number of shared triplets that are anchored in a contracted node on edge (p_v, v), this value is exactly $\binom{v.\texttt{blue}}{2} \cdot v_{ts} + v_{\texttt{blue}} \cdot v_{ps}$.

Scaling to External Memory. If we store T_1 in an array of size $2n-1$ by using a preorder layout, i.e. if a node v is stored in position p, the left child of v is stored in position $p+1$ and if x is the size of the left subtree of v the right child of v is stored in position $p+x+1$, we can make T_1 left heavy in two depth first traversals using $O(n/B)$ I/Os. The preprocessing step that changes the labels of the leafs in T_1 and T_2 can be done in $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os with a cache oblivious sorting routine, e.g. using merge sort. By scanning the left most path that starts from the root of a component C_u, we can find the splitting node of C_u in $O(|C_u|/B)$ I/Os, so in total the number of I/Os spent processing T_1 becomes $O(\frac{n}{B} \log_2 \frac{n}{M})$.

We use the proof of Lemma 4 (see [4]) to initialize an array that can fit the contractions that we need to remember while traversing $MCD(T_1)$. This array is used as a stack that we use to push and pop the contractions of T_2. Each contraction of T_2 is stored in memory using a post order layout, i.e. if a node v is stored in position p and y is the size of the right subtree of v, the left child of v is stored in position $v-y-1$ and the right child of v is stored in position $v-1$. By using a stack, counting and contracting $T_2(u)$ requires $O(|T_2(u)|/B)$ I/Os, so the total number of I/Os spent processing T_2 becomes $O(\frac{n}{B} \log_2 \frac{n}{M})$ as well.

Overall, our algorithm requires $O(\frac{n}{B} \log_2 \frac{n}{M})$ I/Os in the cache oblivious model.

4 The New Algorithm for General Trees

In this algorithm we anchor the triplets of T_1 and T_2 in edges. Let t be a triplet with leafs i, j and k that is either a resolved triplet $ij|k$ or an unresolved triplet ijk, where i is to the left of j and for the triplet ijk, k is also to the right of j. Let w be the lowest common ancestor of i and j and (w, c) the edge from w to the child c whose subtree contains j. We anchor t in edge (w, c). Define $s'(w, c)$ to be the number of triplets anchored in edge (w, c).

Preprocessing. In the preprocessing step of the algorithm, we start by transforming T_1 into a binary tree, denoted $b(T_1)$. Let w be a node of T_1 that has exactly k children, where $k > 2$. The k edges that connect w to its children in T_1 are replaced in $b(T_1)$ by a so called *orange binary tree*. The root of this binary tree is w and the leafs are the k children of w in T_1. Every internal node (except the root) and edge is colored orange, hence the given name. We assume that node w and its k children in T_1, in $b(T_1)$ have the color *black*. This binary tree is built in a way so that every orange node is on the left most path that starts from w, and its left most leaf stores the heaviest child of w in T_1, thus making $b(T_1)$ left heavy. The order in which the other children of w in T_1 are stored in the remaining leafs does not matter, however for the notation below to be mathematically correct, we assume that after constructing $b(T_1)$, the left to right order of the children of w in T_1 is implicitly updated, so that it matches the left to right order in which they appear in the leafs of the orange binary tree below w in $b(T_1)$.

Let u be a node in $b(T_1)$ and c its right child. By construction, c must be a black node. If u is orange, then let $u_{\texttt{root}}$ be the root of the orange binary tree that u is part of. If u is black, then let $u_{\texttt{root}} = u$. Again by construction, $u_{\texttt{root}}$ must be the parent of c in T_1. For the edge (u, c) in $b(T_1)$, we define $s''(u, c)$ to be the set of triplets that are anchored in edge (u_{root}, c) of T_1. Note that for an edge (u', c') in $b(T_1)$ connecting u' with its left child c' we have $s''(u', c') = 0$.

For the number of shared triplets we then have:

$$S(T_1, T_2) = \sum_{(u,c)\in b(T_1)} \sum_{(v,c')\in T_2} |s''(u, c) \cap s'(v, c')| \ .$$

We can capture all triplets in T_1 by coloring $b(T_1)$ instead of T_1. For the nodes u and c where c is the right child of u, the leafs of $b(T_1)$ are colored according to edge (u, c) as follows: the leafs in the left subtree of u are colored red, the leafs in the right right subtree of u are colored blue. If u is an orange node, then the black leafs in the remaining subtrees of the orange binary tree that u is part of are colored green. All other leafs of $b(T_1)$ maintain their color black.

The reason behind transforming T_1 into the binary tree $b(T_1)$, is because now we can use exactly the same core ideas described in Section 3. The tree $b(T_1)$ is a binary tree, so we apply the same preprocessing step, except we do not make it left heavy because by construction it already is. However, we change the labels of the leafs in $b(T_1)$ and T_2, so that the leafs in $b(T_1)$ are numbered 1 to n from left to right.

Modified Centroid Decomposition. After the preprocessing step, we build $MCD(b(T_1))$ as described in Section 3. Then we traverse the nodes of $b(T_1)$, given by a depth first traversal of $MCD(b(T_1))$, where we visit the children of every node u in $MCD(b(T_1))$ from left to right.

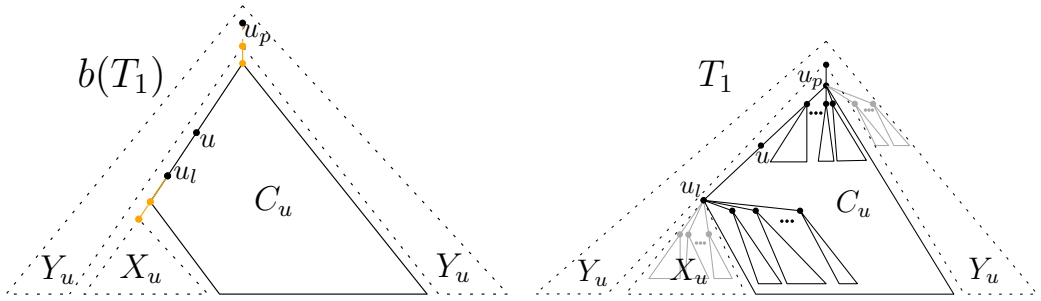


Figure 3 How a component in $b(T_1)$ translates to a component in T_1.

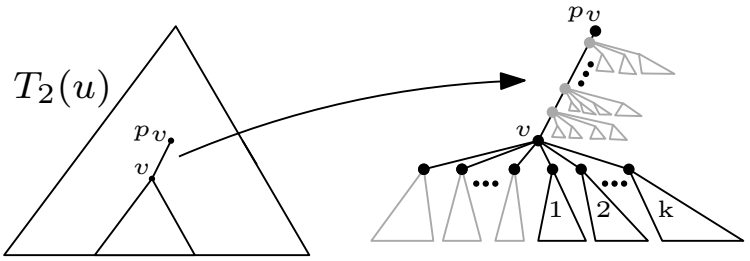


Figure 4 $T_2(u)$: Contracted children subtrees rooted on node v and contracted subtrees rooted on contracted nodes (gray color) in edge (p_v, v).

Like in the binary algorithm, while traversing $MCD(b(T_1))$ we process T_2 in two phases, the contraction phase and the counting phase. The only difference after this point in the algorithm for general trees, is the counters that we have to maintain in the contracted versions of T_2, but otherwise, the same analysis from Section 3 holds.

Contraction. The contraction of T_2 with respect to a set of leafs $\Lambda \subseteq L(T_2)$, happens in the exact same way as described in Section 3, i.e. we start by pruning all leafs of T_2 that are not in Λ, then we prune all internal nodes of T_2 with no children, and finally, we contract the nodes that have exactly one child.

Let u be a node of $MCD(b(T_1))$ and C_u the corresponding component of $b(T_1)$. For every such node u we have a contracted version of T_2, denoted $T_2(u)$, where $L(T_2(u)) = L(C_u)$. Like in the binary algorithm, the goal is to augment $T_2(u)$ with counters, so that we can find $\sum_{(v,c')\in T_2} |s''(u,c) \cap s'(v,c')|$ by scanning $T_2(u)$ instead of T_2.

Because of the location where the triplets are anchored, in $T_2(u)$ every leaf that was contracted away, must have a color and be stored in some way. The color of each leaf depends on the type of the component that we have in $b(T_1)$ and the splitting node that is used for that component. For example, in Figure 3 the contracted leafs from X_u will have the red color because like in the binary algorithm $b(T_1)$ is left heavy. The contracted leafs from the children subtrees of u_p in T_1 can either have the color green or black. If u in $b(T_1)$ happens to be orange and part of the orange binary tree that u_p is the root of, then the color must be green, otherwise black. Finally, every leaf that is not in the subtree defined by u_p, and thus is in Y_u, must have the color black. The way we store this information is described in the counting phase below.

Counting. In Figure 4 we illustrate how a node v in $T_2(u)$ can look like. The contracted subtrees are illustrated with the dark gray color. Every such subtree contains some number

of red, green and black leafs. The counters that we need to maintain should be so that if v has k children in $T_2(u)$, then we can count all shared triplets that are anchored in every child edge (including those of the contracted children subtrees) of v in $O(k)$ time. At the same time, in $O(1)$ time we should be able to count all shared triplets that are anchored in every child edge of every contracted node that lies on edge (p_v, v). In this way, the counting phase will require $O(|T_2(u)|)$ time, hence we will get the same bounds like in the binary algorithm.

In v we have the following counters:

- $v_{\texttt{i}}$: number of leafs with color i (including the contracted leafs) in the subtree of v, where $i \in \{\texttt{red, blue, green}\}$.
- $\overline{v}_{\texttt{black}}$: number of black leafs (including the contracted leafs) not in the subtree of v.

We divide the rest of the counters into two categories. The first category corresponds to the leafs in the contracted children subtrees of v and each counter will be stored in a variable of the form $v_{A.x}$. The second category corresponds to the leafs in the contracted subtrees in edge (p_v, v) and each counter will be stored in a variable of the form $v_{B.x}$.

For the first category A we have the following counters:

- $v_{A.\texttt{i}}$: total number of leafs with color i in the contracted children subtrees of v, where $i \in \{\texttt{red, green, black}\}$.
- $v_{A.\texttt{red,green}}$: total number of pairs of leafs where one is red, the other is green and one leaf comes from one contracted child subtree of v and the other leaf comes from a different contracted child subtree of v.

While scanning the k children edges of v from left to right, for the child c' that is the m^{th} child of v, we also maintain the following:

- $a_{\texttt{i}}$: total number of leafs with color i from the first $m-1$ children subtrees, including the contracted children subtrees, where $i \in \{\texttt{red, blue, green}\}$.
- $p_{\texttt{i,j}}$: total number of pairs of leafs from the first $m-1$ children subtrees, including the contracted children subtrees, where one has color i, the other has color j and they both come from different subtrees (one might be contracted and the other non-contracted). We have that $(i,j) \in \{(\texttt{red}, \texttt{green}), (\texttt{red}, \texttt{blue}), (\texttt{blue}, \texttt{green})\}$.
- $t_{\texttt{red,blue,green}}$: total number of leaf triples from the first $m-1$ children subtrees, including the contracted children subtrees, where one is red, one is blue and one is green, and all three leafs come from different subtrees (some might be contracted, some might be non-contracted).

Every variable is initialized and updated in the following order:

- $(a_{\texttt{red}}, a_{\texttt{blue}}, a_{\texttt{green}}) = (v_{A.\texttt{red}}, 0, v_{A.\texttt{green}})$
- $p_{\texttt{red,green}} = v_{A.\texttt{red,green}}$
- $p_{\texttt{red,blue}} = p_{\texttt{blue,green}} = t_{\texttt{red,blue,green}} = 0$
- $a_i = a_i + c'_i$, where $i \in \{\texttt{red, blue, green}\}$.
- $p_{i,j} = p_{i,j} + a_i \cdot c'_j + a_j \cdot c'_i$, where $(i,j) \in \{(\texttt{red}, \texttt{green}), (\texttt{red}, \texttt{blue}), (\texttt{blue}, \texttt{green})\}$
- $t_{\texttt{red,blue,green}} = t_{\texttt{red,blue,green}} + p_{\texttt{red,green}} \cdot c'_{\texttt{blue}} + p_{\texttt{red,blue}} \cdot c'_{\texttt{green}} + p_{\texttt{blue,green}} \cdot c'_{\texttt{red}}$

After finishing scanning the k children edges of v, we can compute the shared triplets that are anchored in every child edge of v (including the children edges pointing to contracted subtrees) as follows: for the total number of shared resolved triplets, denoted $\texttt{tot}_{\texttt{A.res}}$, we have that $\texttt{tot}_{\texttt{A.res}} = p_{\texttt{red,blue}} \cdot \overline{v}_{\texttt{black}}$ and for the total number of shared unresolved triplets, denoted $\texttt{tot}_{\texttt{A.unres}}$, we have that $\texttt{tot}_{\texttt{unres}} = t_{\texttt{red,blue,green}}$.

The second category B of counters will help us count triplets involving leafs (contracted and non-contracted) from the subtree of v and leafs from the contracted subtrees rooted on edge (p_v, v). We maintain the following:

- $v_{B.i}$: total number of leafs with color i in all contracted subtrees rooted on edge (p_v, v), where $i \in \{\texttt{red, green, black}\}$.
- $v_{B.\texttt{red,green}}$: total number of pairs of leafs where one is red and the other is green such that one leaf comes from a contracted child subtree of a contracted node v' and the other leaf comes from a different contracted child subtree of the same contracted node v'.

- $v_{B.\texttt{red,black}}$: total number of pairs of leafs where one is red and the other is black such that the red leaf comes from a contracted child subtree of a contracted node v' and the black leaf comes from a contracted child subtree of a contracted node v''. For v' and v'' we have that v'' is closer to v_p than v'.

For the total number of shared unresolved triplets, denoted $\texttt{tot}_{B.\texttt{unres}}$, that are anchored in the children edges of every contracted node that exists in edge (v_p, v), we have that $\texttt{tot}_{B.\texttt{unres}} = v_{\texttt{blue}} \cdot v_{B.\texttt{red,green}}$. For the total number of shared resolved triplets, denoted $\texttt{tot}_{B.\texttt{res}}$, that are anchored in the children edges of every contracted node that exists in edge (v_p, v), we have that $\texttt{tot}_{B.\texttt{res}} = v_{\texttt{blue}} \cdot v_{B.\texttt{red,black}} + v_{\texttt{blue}} \cdot v_{B.\texttt{red}} \cdot (\overline{v}_{\texttt{black}} - v_{B.\texttt{black}})$.

5 Experiments

The implementation of both algorithms was made using the C++ programming language. The source code can be found in `https://github.com/kmampent/CacheTD`.

The Setup. The experiments were performed on a machine with 8GB RAM, Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz, 32K L1 cache, 256K L2 cache and 6144K L3 cache. The operating system was Ubuntu 16.04.2 LTS. The compiler used was g++ 5.4 and cmake 3.5.1.

Generating Random Trees. We use two different models for generating input trees. The first model is called the *random model*. A tree T with n leafs in this model is generated as follows:

- Create a binary tree T' with n leafs as follows: start with a binary tree T' with two leafs. Iteratively pick a leaf l uniformly at random. Make l an internal node by appending a left child node and a right child node to l, thus increasing the number of leafs in T' by exactly 1.
- With probability p contract every internal node u of T', i.e make the children of u be the children of u's parent and remove u.

The second model is called the *skewed* model. In this model, we can control more directly the shape of the input trees. A tree T with n leafs in this model is generated as follows:

- Create a binary tree T' with n leafs as follows: let $0 \leq \alpha \leq 1$ be a parameter, u some internal node in T, l and r the left and right children of u, and $T(u)$, $T(l)$ and $T(r)$ the subtrees rooted on u, l and r respectively. Create T' so that for every internal node u we have $\frac{|T(l)|}{|T(u)|} \approx \alpha$, i.e. $|T_l| = \max(1, \min(\lfloor a \cdot n \rfloor, n-1))$ and $|T_r| = 1 - |T_l|$, where $|T_l|$ and $|T_r|$ are the number of leafs in $T(l)$ and $T(r)$ respectively.
- With probability p contract every internal node u of T' like in the random model.

In both models, after creating T, we shuffle the leaf labels by using `std::shuffle`[1] together with `std::default_random_engine`[2].

Implementations Tested. Let p_1 and p_2 denote the contraction probability of T_1 and T_2 respectively. When $p_1 = p_2 = 0$, the trees T_1 and T_2 are binary trees, so in our experiments we use the algorithm from Section 3. In all other cases, the algorithm from Section 4 is used. Note that the algorithm from Section 4 can handle binary trees just fine, however there is an

[1] `http://www.cplusplus.com/reference/algorithm/shuffle/`
[2] `http://www.cplusplus.com/reference/random/default_random_engine/`

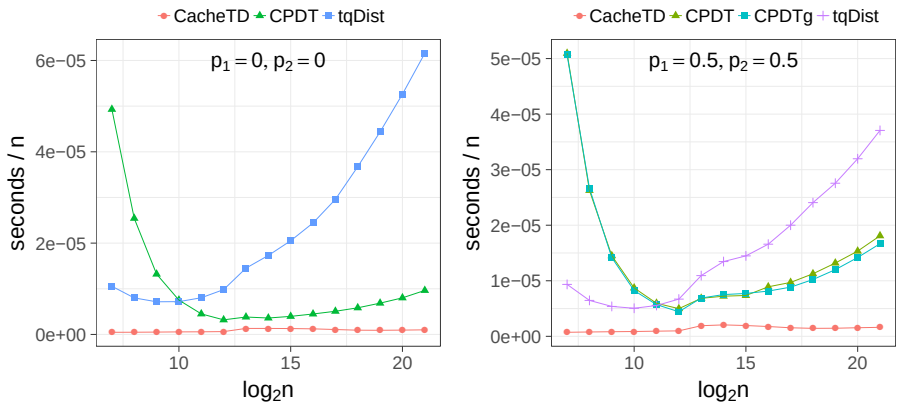


Figure 5 Time performance in the random model.

extra overhead (factor 1.8 slower) compared to the algorithm from Section 3 that comes due to the additional counters that we have to maintain for the contractions of T_2.

We compared our implementation with previous implementations of [10] and [14, 3] available at `http://sunflower.kuicr.kyoto-u.ac.jp/~jj/Software/` and `http://users-cs.au.dk/cstorm/software/tqdist/` respectively. The implementation of the $O(n \log^3 n)$ algorithm in [10] has two versions, one that uses `unordered_map`[3], which we refer to as `CPDT`, and another that uses sparsehash[4], which we refer to as `CPDTg`. For binary input trees the hash maps are not used, thus `CPDT` and `CPDTg` are the same. The tqdist library [15], which we will refer to as `tqDist`, has an implementation of the binary $O(n \log^2 n)$ algorithm from [14] and the general $O(n \log n)$ algorithm from [3]. If the two input trees are binary the $O(n \log^2 n)$ algorithm is used. We will refer to our new algorithm as `CacheTD`.

Statistics. We measured the execution time of the algorithms with the `clock_gettime` function in C++. Due to the different parser implementations, we do not consider the time taken to parse the input trees. We used the PAPI library[5] for statistics related to L1, L2 and L3 cache accesses and misses. Finally, we count the space of the algorithms by considering the *Maximum resident set size* returned by `/usr/bin/time -v`.

Results. The experiments are divided into two parts. In the first part, we look at how the algorithms perform when the memory requirements do not exceed the available main memory (8G RAM). In the second part, we look at how they perform when the memory requirements exceed the available main memory (by limiting the available RAM to the operating system to be 1GB), thus forcing the operating system to use the swap space, which in turn can for large enough input trees yield the very expensive disk I/Os.

RAM experiments. For the random model, in Figure 5 we illustrate a time comparison of all implementations for trees of up to 2^{21} leafs ($\sim$ 2 million) with varying contraction probabilities. Every data point is the average of 10 runs. In all cases `CacheTD` achieves the best time performance. The space and L1/L2/L3 cache performance of `CacheTD` is the best

[3] `http://en.cppreference.com/w/cpp/container/unordered_map`
[4] `https://github.com/sparsehash/sparsehash`
[5] `http://icl.utk.edu/papi/`

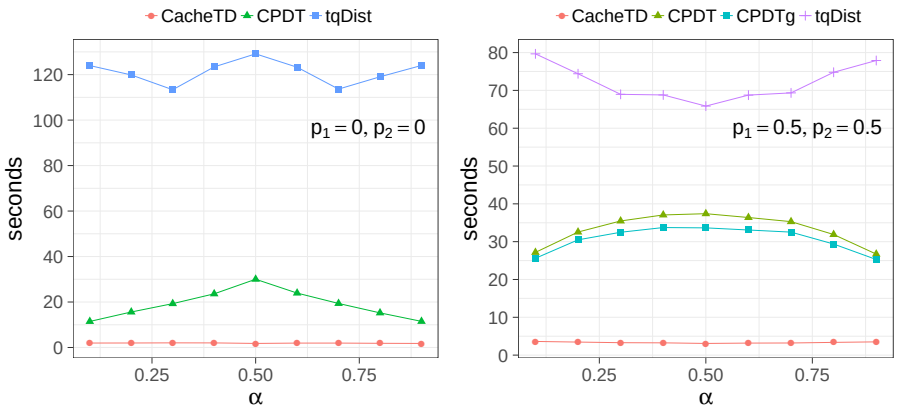


Figure 6 How the alpha parameter affects running time ($n = 2^{21}$).

Table 1 Time performance when limiting the available RAM to be 1GB. For both tables we have $\alpha = 0.5$. For the left table we have $p_1 = p_2 = 0$ and for the right table $p_1 = p_2 = 0.5$.

n	CPDT/CPDTg	tqDist	CacheTD
2^{17}	0m:01s	0m:08s	0m:01s
2^{18}	0m:02s	3m:10s	0m:01s
2^{19}	0m:05s	2h:16m	0m:01s
2^{20}	0m:34s	-	0m:01s
2^{21}	7h:09m	-	0m:03s
2^{22}	-	-	0m:35s
2^{23}	-	-	10m:09s
2^{24}	-	-	43m:52s

n	CPDT	CPDTg	tqDist	CacheTD
2^{17}	0m:01s	0m:01s	0m:03s	0m:01s
2^{18}	0m:03s	0m:03s	1m:18s	0m:01s
2^{19}	0m:10s	0m:07s	19m:02s	0m:01s
2^{20}	1h:58m	6h:32m	>10h	0m:02s
2^{21}	-	-	-	0m:56s
2^{22}	-	-	-	4m:11s
2^{23}	-	-	-	24m:44s
2^{24}	-	-	-	2h:13m

as well (see [4]). For the skewed model, in Figure 6 we plot the alpha parameter against the execution time of the algorithms, when $n = 2^{21}$. The alpha parameter has the least effect on `CacheTD`, with the maximum running time being only a factor of 1.1 larger than the minimum. From Section 2, `CPDT` and `CPDTg` use the heavy light decomposition for T_2. When α approaches 0 or 1, the number of heavy paths that will be updated because of a leaf color change decreases, thus the total number of operations of the algorithm decreases as well (see [4]).

I/O experiments. The results are included in Table 1. Each cell contains the execution time (including the waiting time due to disk I/Os). For this experiment we used the `time` function of Ubuntu and thus also considered the time taken to parse the input trees. Each cell contains the result of 1 run and for input trees with 2^{23} and 2^{24} leafs we used the 128 bit implementation of the new algorithms to avoid numeric overflows. The exact running times vary from run to run, but the general outcome is the same: unlike `CacheTD`, the performance of `CPDT`, `CPDTg` and `tqDist` deteriorates significantly the moment they start performing disk I/Os. More elaborate I/O experiments can be found in the arXiv version of the paper [4].

References

1 M.S. Bansal, J. Dong, and D. Fernández-Baca. Comparing and aggregating partially resolved trees. *Theoretical Computer Science*, 412(48):6634–6652, 2011.

2 V. Berry and O. Gascuel. Inferring evolutionary trees with strong combinatorial evidence. *Theoretical Computer Science*, 240(2):271–298, 2000. doi:10.1016/S0304-3975(99)00235-2.

3 G.S. Brodal, R. Fagerberg, C.N.S. Pedersen, T. Mailund, and A. Sand. Efficient algorithms for computing the triplet and quartet distance between trees of arbitrary degree. *24th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1814–1832, 2013. doi:10.1137/1.9781611973105.130.

4 G.S Brodal and K. Mampentzidis. Cache oblivious algorithms for computing the triplet distance between trees. *Computing Research Repository*, abs/1706.10284, 2017.

5 D.E. Critchlow, D.K. Pearl, and C.L. Qian. The Triples Distance for Rooted Bifurcating Phylogenetic Trees. *Systematic Biology*, 45(3):323, 1996. doi:10.1093/sysbio/45.3.323.

6 A.J. Dobson. Comparing the shapes of trees. *Combinatorial Mathematics III. Lecture Notes in Mathematics*, pages 95–100, 1975. doi:10.1007/BFb0069548.

7 G.F. Estabrook, F.R. McMorris, and C.A. Meacham. Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. *Systematic Zoology*, 34(2):193–200, 1985. doi:10.2307/2413326.

8 M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–297, 1999. doi:10.1109/SFFCS.1999.814600.

9 M.K. Holt, J. Johansen, and G.S. Brodal. On the scalability of computing triplet and quartet distances. *16th Workshop on Algorithm Engineering and Experiments*, pages 9–19, 2014. doi:10.1137/1.9781611973198.2.

10 J. Jansson and R. Rajaby. A more practical algorithm for the rooted triplet distance. *International Conference on Algorithms for Computational Biology*, pages 109–125, 2015. doi:10.1007/978-3-319-21233-3_9.

11 J. Jansson and R. Rajaby. A More Practical Algorithm for the Rooted Triplet Distance. *Journal of Computational Biology*, 24(2):106–126, 2017. doi:10.1089/cmb.2016.0185.

12 D.F. Robinson and L.R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1):131–147, 1981. doi:http://dx.doi.org/10.1016/0025-5564(81)90043-2.

13 N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406, 1987. doi:10.1093/oxfordjournals.molbev.a040454.

14 A. Sand, G.S. Brodal, R. Fagerberg, C.N.S. Pedersen, and T. Mailund. A practical $O(n \log^2 n)$ time algorithm for computing the triplet distance on binary trees. *BMC Bioinformatics*, 14(2):S18, 2013. doi:10.1186/1471-2105-14-S2-S18.

15 A. Sand, M.K. Holt, J. Johansen, G.S. Brodal, T. Mailund, and C.N.S. Pedersen. tqdist: A library for computing the quartet and triplet distances between binary or general trees. *Bioinformatics*, 30(14):2079, 2014. doi:10.1093/bioinformatics/btu157.

Online Algorithms for Maximum Cardinality Matching with Edge Arrivals*

Niv Buchbinder[1], Danny Segev[2], and Yevgeny Tkach[3]

1 **Department of Statistics and Operations Research, School of Mathematical Sciences, Tel Aviv University, Israel**
niv.buchbinder@gmail.com
2 **Department of Statistics, University of Haifa, Haifa 31905, Israel**
segevd@stat.haifa.ac.il
3 **Department of Statistics and Operations Research, School of Mathematical Sciences, Tel Aviv University, Israel**
ivgitk@gmail.com

Abstract

In the adversarial edge arrival model for maximum cardinality matching, edges of an unknown graph are revealed one-by-one in arbitrary order, and should be irrevocably accepted or rejected. Here, the goal of an online algorithm is to maximize the number of accepted edges while maintaining a feasible matching at any point in time. For this model, the standard greedy heuristic is $1/2$-competitive, and on the other hand, no algorithm that outperforms this ratio is currently known, even for very simple graphs.

We present a clean *Min-Index* framework for devising a family of randomized algorithms, and provide a number of positive and negative results in this context. Among these results, we present a $5/9$-competitive algorithm when the underlying graph is a forest, and prove that this ratio is best possible within the Min-Index framework. In addition, we prove a new general upper bound of $\frac{2}{3+1/\phi^2} \approx 0.5914$ on the competitiveness of any algorithm in the edge arrival model. Interestingly, this bound holds even for an easier model in which vertices (along with their adjacent edges) arrive online, and when the underlying graph is a tree of maximum degree at most 3.

1998 ACM Subject Classification F.1.2 Online computation; G.2.2 Graph algorithms.

Keywords and phrases Maximum matching, online algorithms, competitive analysis, primal-dual method.

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.22

1 Introduction

Graph matchings are cornerstone problems in combinatorial optimization, that have extensively been studied by the discrete mathematics, computer science, and operations research communities. In the most fundamental setting, given an undirected graph $G = (V, E)$, our objective is to identify a maximum cardinality matching, namely, a subset of edges $M \subseteq E$ without any vertices in common. Motivated by emerging applications in online advertising, numerous generalizations of this classic problem have been investigated in the last two decades from the perspective of both offline and online settings.

* The research of Niv Buchbinder is supported by ISF grant 1585/15 and US-Israel BSF grant 2014414. The research of Danny Segev is supported by ISF grant 148/16.

In online computation, the seminal work of Karp, Vazirani, and Vazirani [14] studies an online model of maximum cardinality matching in which the underlying graph is bipartite. Specifically, the vertices on one side of the partition are known in advance, whereas those on the other side arrive one-by-one in online fashion. Upon the arrival of a vertex, all its adjacent edges are revealed simultaneously; the algorithm is then required to irrevocably decide how to match the newly arrived vertex. For this setting, Karp et al. designed a randomized $(1 - 1/e)$-competitive algorithm, and showed that the latter factor is best possible. Due to the breadth and depth of subsequent research on this one-sided arrival model, it is beyond the scope of this paper to provide a comprehensive literature review. For this purpose, we refer the reader to a number of selected papers on this topic [12, 3, 18, 1, 6, 7], as well as to additional work on non-adversarial settings, in which the input sequence is randomly generated [10, 5, 9, 15, 13, 16], and finally, to the excellent survey of Mehta [17].

Additional online models, most of which are somewhat more difficult in terms of the achievable competitive ratio, have been proposed in recent years. Wang and Wong [20] introduced a *vertex arrival* model, where vertices from either side of the partition arrive online. Here, whenever a vertex arrives, all edges connecting this vertex to previously arrived vertices are revealed simultaneously. Wang and Wong demonstrated that this model is strictly harder than the one-sided vertex arrival model of Karp et al. [14] by proving an upper bound of $0.6252 < 1 - 1/e$. In addition, they presented a fractional matching algorithm with a competitive ratio of 0.526. An even harder setting is the *edge arrival* model, where edges are revealed one-by-one in arbitrary order, and should be irrevocably accepted or rejected. For this model, the simple greedy heuristic, that deterministically adds an arriving edge to the current matching whenever possible, is $1/2$-competitive. Consequently, the main open question is whether we can attain competitiveness strictly better than $1/2$.

Existing results for relaxed models. As addressing the above question in the edge arrival model seems particularly challenging, recent efforts have mainly concentrated on various relaxations. One such relaxation allows for preemption, where the algorithm is allowed to discard previously accepted edges. For this model, Epstein, Levin, Segev, and Weimann [8] established an upper bound of $\frac{1}{1+\ln 2} \approx 0.591$ on the competitiveness of any algorithm, even on bipartite graphs. To our knowledge, this is the best known upper bound for the edge arrival model without preemption as well. Chiplunkar, Tirodkar, and Vishwanathan [4] designed a $15/28 \approx 0.535$-competitive algorithm for a special case of the vertex arrival model on a tree graph. Very recently, Tirodkar and Vishwanathan [19] devised a $33/64 \approx 0.515$-competitive algorithm for trees in the edge arrival model. As mentioned earlier, these results are heavily based on preemptions. Guruganesh and Singla [11] studied a different type of relaxation, in which edges arrive according to a uniformly-picked random permutation, rather than in an arbitrary adversarial order. Under this assumption, they were able to design a $(1/2 + \delta)$-competitive algorithm, for some absolute constant $\delta > 0$. Nevertheless, for the adversarial edge arrival model, no algorithm that outperforms the basic greedy heuristic is known at present time, even for seemingly-simple network topologies, such as trees or bounded-degree graphs.

The Min-Index framework. In this paper, we consider the adversarial edge arrival model for maximum cardinality matching. Here, a randomized algorithm can be thought of as a procedure that maintains at any given time, explicitly or implicitly, a distribution over matchings. These are updated whenever a new edge arrives, subject to the respective online constraints on the allowable updates. However, since maintaining a general distribution of this

Algorithm 1 Min-Index$(k, p_1, \ldots, p_k)$

Initialization: $M_i \leftarrow \varnothing$, for every $i = 1, \ldots, k$.
When edge e arrives:
If e cannot be added to any of the matchings $M_1, \ldots, M_k$, reject this edge. Otherwise, update $M_i \leftarrow M_i \cup \{e\}$, where i is the minimal index for which $M_i \cup \{e\}$ is a feasible matching.
Return M_i with probability p_i.

nature may be a difficult task, we propose a simple family of randomized algorithms, which is referred to as the *Min-Index framework*. Our generic algorithm maintains a pre-determined distribution over k matchings, $M_1, \ldots, M_k$, and associates each matching M_i with a fixed probability p_i, such that $\sum_{i=1}^{k} p_i = 1$. Whenever a new edge arrives, it is greedily accepted to the first matching (index-wise) for which this augmentation is possible; when no such matching exists, the current edge is rejected. As a result, we obtain a clean framework that directly leads to a randomized online matching algorithm, whose formal statement is given in Algorithm 1.

1.1 Our results

Our main contribution is to prove tight upper and lower bounds of $5/9 \approx 0.555$ for the Min-Index framework on forest graphs, as stated in the following theorem.

▶ **Theorem 1.** *For a forest graph, the generic Min-Index algorithm instantiated with $k = 3$ and $(p_1, p_2, p_3) = (5/9, 3/9, 1/9)$ is $5/9$-competitive for the edge arrival model. Moreover, any instantiation of Min-Index, with any number of matchings and respective probabilities, is at most $5/9$-competitive for forest graphs.*

This result improves on that of Chiplunkar et al. [4], who obtained (in a vertex arrival model) a competitive ratio of $15/28 \approx 0.535$ on forests, as well as on the results of Tirodkar and Vishwanathan [19], who obtained (for the edge arrival model) a $33/64 \approx 0.515$-competitive algorithm. In fact, as mentioned earlier, both of these bounds are in an easier model, allowing the online algorithm to preempt edges.

As a warmup, we also show that for graphs of maximum degree 2 (i.e., union of paths and cycles) the Min-Index algorithm with $k = 2$ matchings, picked with probabilities $(p_1, p_2) = (2/3, 1/3)$, is $2/3$-competitive. This result is shown to be best possible for any algorithm in the edge arrival model.

For graphs of maximum degree d, we prove that any instantiation of Min-Index is at most $\frac{1}{2}(1 + \frac{1}{2^d - 1})$-competitive, even on bipartite graphs. In spite our best efforts, we could not match this bound. However, inspired by the general idea behind this framework, we design a *fractional* $\frac{1}{2}(1 + \frac{1}{2^d - 1})$-competitive algorithm on graphs of maximum degree d. In other words, our online procedure computes a fractional matching whose objective value with respect to the standard LP-relaxation of maximum cardinality matching (see Figure 1) is within factor $\frac{1}{2}(1 + \frac{1}{2^d - 1})$ of optimal.

Our final contribution is to establish a general upper bound for any online algorithm.

▶ **Theorem 2.** *The competitive ratio of any fractional (or randomized) online algorithm for maximum matching in the vertex arrival model even for subcubic trees is at most $\frac{2}{3 + 1/\phi^2} \approx 0.5914$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.*

Interestingly, this result holds even in the vertex arrival model, when the underlying graph is a tree of maximum degree 3. On the one hand, this bound still leaves some marginal

$$\begin{aligned}
&\text{(P) maximize} && \sum_{e\in E} y_e && && \text{(D) minimize} && \sum_{v\in V} x_v && \\
&\text{subject to:} && \sum_{e\in\delta(v)} y_e \le 1 && \forall v\in V && \text{subject to:} && x_u + x_v \ge 1 && \forall e\in E \\
& && y_e \ge 0 && \forall e\in E && && x_v \ge 0 && \forall v\in V
\end{aligned}$$

Figure 1 The primal matching problem (P) and its dual the vertex cover problem (D).

room for improvements in comparison to our $5/9 \approx 0.555$-competitive algorithm on forests, and our fractional $4/7 \approx 0.571$ algorithm for subcubic graphs (note that $\frac{1}{2}(1+\frac{1}{2^d-1}) = 4/7$ for $d=3$). On the other hand, $\frac{2}{3+1/\phi^2} \approx 0.5914$ improves on the currently best-known upper bound of 0.6252 for the vertex arrival model, due to Wang and Wong [20]. We note in passing that, for the more restrictive edge arrival model (even with preemption), a slightly better upper bound of $\frac{1}{1+\ln 2} \approx 0.5906$ was proven by Epstein et al. [8]. However, their bound holds for high-degree bipartite graphs, while our bound holds even for trees of maximum degree 3.

Organization. All algorithms are given in Section 2, with corresponding upper bounds on the Min-Index framework in Section 3. Our general upper bound is established in Section 4.

1.2 Techniques

The main technical ingredient in proving lower bounds on the competitiveness of our algorithms is based on a primal-dual approach. Specifically, we make use of the standard fractional matching LP and its dual, the fractional vertex cover LP, both stated in Figure 1. To analyze the performance of our algorithms on various classes of graphs, we construct in each setting a feasible fractional vertex cover, that will eventually allow us to bound the expected cardinality of the resulting matching with respect to the optimal vertex cover, and in turn, with respect to the optimal matching via weak duality. In some cases, this construction is performed in an offline fashion, requires complete knowledge of the final input graph, and hence can be viewed as employing a dual-fitting approach. In other cases, our construction is fully online, and therefore also yields a monotonically increasing fractional vertex cover. Such solutions can be rounded online with no loss in optimality on bipartite graphs [20].

To prove upper bounds for the Min-Index algorithm, we construct adversarial sequences that allow us to derive linear inequalities on the achievable competitive ratio in terms of the choice probabilities of the different matchings. These inequalities naturally induce a linear program whose optimal solution provides an upper bound on the best-possible competitiveness. An approach in this spirit has recently been employed by Azar, Cohen, and Roytman [2]. For our general upper bound of $\frac{2}{3+1/\phi^2}$, the adversarial sequences we consider are parameterized according to the phase number m upon which they terminate. As a result, in order to derive an upper bound on the competitiveness of any fractional online algorithm, we are required to solve a corresponding linear program, parameterized by m as well. Rather than solving this LP numerically, we obtain an explicit closed-form solution for its optimum, thereby proposing an analytical proof for the desired upper bound, as the number of phases m tends to infinity.

2 Algorithms

In this section, we establish lower bounds on the competitive ratio of the generic Min-Index framework. Specifically, in Section 2.1, we show that for graphs of maximum degree 2, an appropriate instantiation of the Min-Index algorithm is $2/3$-competitive. In Section 2.2, we prove the first part of Theorem 1, arguing that the right instantiation of the Min-Index algorithm is $5/9$-competitive on forest graphs. Finally, in Section 2.3, we design a fractional $\frac{1}{2}(1+\frac{1}{2^d-1})$-competitive algorithm for graphs of maximum degree at most d.

2.1 A $2/3$-competitive algorithm for graphs of maximum degree 2

As a warm-up, we demonstrate some of our ideas by analyzing how the Min-Index algorithm performs on graphs of maximum degree 2. Such graphs can be viewed as a union of vertex-disjoint cycles and paths, whose edges are revealed to the algorithm one-by-one. Due to space limitations, the proof of Theorem 3 below is omitted. We remark that this proof also shows that a competitive ratio of $3/2$ can be attained for the online fractional vertex cover problem in graphs of maximum degree 2.

▶ **Theorem 3.** *On graphs on maximum degree 2, the Min-Index algorithm with $k = 2$ matchings, picked with probabilities $(p_1, p_2) = (2/3, 1/3)$, is $2/3$-competitive.*

2.2 A $5/9$-competitive algorithm for forest graphs

In this section, we design a randomized $5/9$-competitive algorithm when the underlying graph is a forest. Specifically, as stated in the next theorem, this competitive ratio is attained by our Min-Index algorithm.

▶ **Theorem 4.** *On forests, the Min-Index algorithm with $k = 3$ matchings, picked with probabilities $(p_1, p_2, p_3) = (5/9, 3/9, 1/9)$, is $5/9$-competitive.*

Proof. Clearly, the algorithm returns a feasible matching. Thus, it remains to analyze the expected cardinality of its output matching, given by $p_1 \cdot |M_1| + p_2 \cdot |M_2| + p_3 \cdot |M_3| = \frac{5}{9} \cdot |M_1| + \frac{3}{9} \cdot |M_2| + \frac{1}{9} \cdot |M_3|$. To this end, we use once again a primal-dual approach, by constructing a feasible fractional vertex cover to the dual LP (D), shown in Figure 1. We then prove that the expected cardinality of the matching produced by the algorithm is at least $5/9$ times the value of this fractional vertex cover.

In the (omitted) proof of Theorem 3, we construct a dual solution step-by-step, resulting in an algorithm with a similar competitive ratio for the online fractional vertex cover problem. On the other hand, in this case the (dual) fractional vertex cover is constructed retrospectively, once the input sequence has ended. As the final graph is guaranteed to be a forest, we separately define a feasible vertex cover for each tree of the forest. To this end, consider such a tree, T. We first root T at an arbitrarily-picked vertex r, and orient the edges from the root down toward the leaves, such that each vertex other than r has one ingoing edge. With respect to this orientation, for each edge $e = (u, v)$ that was oriented $u \to v$, we update the fractional vertex cover according to the 4 possible decisions of our algorithm:

1. When $e = u \to v$ is accepted to M_1: $x_u \leftarrow x_u + 3/5$ and $x_v \leftarrow x_v + 2/5$.
2. When $e = u \to v$ is accepted to M_2 (after being rejected from M_1): $x_u \leftarrow x_u + 2/5$ and $x_v \leftarrow x_v + 1/5$.
3. When $e = u \to v$ is accepted to M_3 (after being rejected from both M_1 and M_2): $x_u \leftarrow x_u + 1/5$ and x_v is not updated.
4. When $e = u \to v$ is rejected from M_1, M_2, and M_3: The values x_u and x_v are not updated.

First, we claim that the expected cardinality of the matching produced by our algorithm is precisely $\frac{5}{9}$ times the value of the fractional vertex cover solution we have just constructed. This claim is straightforward, as whenever an edge e is accepted to one of the matchings M_i, corresponding to cases 1-3 above, the expected cardinality of the matching increases by p_i. On the other hand, it is easy to verify that, by our construction, the increase in the fractional vertex cover solution is exactly $\frac{9}{5}p_i$. In case 4, when an edge e is not accepted to any of the matchings, the vertex cover solution remains unchanged. Thus, it remains to prove that the fractional vertex cover is indeed feasible. To this end, we show that $x_u + x_v \geq 1$ for every edge $e = u \to v$ by inspecting the 4 possible decisions of our algorithm.

Case 1: The edge $e = u \to v$ is accepted to M_1. In this case, due to the updates $x_u \leftarrow x_u + 3/5$ and $x_v \leftarrow x_v + 2/5$, we clearly have $x_u + x_v \geq 1$.

Case 2: The edge $e = u \to v$ is accepted to M_2. Since e was rejected from M_1, there must be an edge $e' \in M_1$ that is adjacent to e. By construction, the vertex cover update due to the edge e increases $x_u + x_v$ by $\frac{3}{5}$, whereas that of e' contributes at least $\frac{2}{5}$ to $x_u + x_v$, meaning that e is fractionally covered.

Case 3: The edge $e = u \to v$ is accepted to M_3. Since e was rejected from both M_1 and M_2, there must be a pair of edges, $e_1 \in M_1$ and $e_2 \in M_2$, that are both adjacent to e. Due to the update rule in this case, the edge e caused $x_u + x_v$ to increase by $\frac{1}{5}$, and it remains to show that the combined contribution of e_1 and e_2 to $x_u + x_v$ is at least $\frac{4}{5}$. Note that the orientation of T guarantees that u has at most one ingoing edge. Therefore, at most one of e_1 and e_2 is of the form $w \to u$, and we are left with considering the following cases:

- When $e_1 = w \to u$: Here, e_2 is necessarily of the form $v \to z$ or $u \to z$. It follows that e_1 contributes $\frac{2}{5}$ to x_u whereas e_2 contributes $\frac{2}{5}$ to either x_v or x_u.
- When $e_2 = w \to u$: Then, e_1 is of the form $v \to z$ or $u \to z$. In this case, e_2 contributes $\frac{1}{5}$ to x_u, and e_1 contributes $\frac{3}{5}$ to either x_v or x_u.
- When both e_1 and e_2 are of the form $v \to z$ or $u \to z$: The respective contributions of e_1 and e_2 to $x_u + x_v$ are $\frac{3}{5}$ and $\frac{2}{5}$.

Case 4: The edge $e = u \to v$ is rejected from M_1, M_2, and M_3. Since e is rejected from M_1, M_2, and M_3, this edge must be adjacent to some $e_1 \in M_1$, $e_2 \in M_2$, and $e_3 \in M_3$. We prove that the total contribution of these three edges to $x_u + x_v$ is at least 1. Similar to the argument used in case 3, at most one edge out of e_1, e_2, and e_3 is of the form $w \to u$, and we therefore consider the following cases:

- When $e_1 = w \to u$: The contribution of e_1 to x_u is $\frac{2}{5}$. The contribution of e_2 to x_v or x_u is $\frac{2}{5}$, and the contribution of e_3 to x_v or x_u is $\frac{1}{5}$. Hence, $x_u + x_v \geq 1$.
- When $e_2 = w \to u$: The contribution of e_2 to x_u is $\frac{1}{5}$. The contribution of e_1 to x_v or x_u is $\frac{3}{5}$, and the contribution of e_3 to x_v or x_u is $\frac{1}{5}$. Once again, $x_u + x_v \geq 1$.
- When $e_3 = w \to u$: Even though e_3 does not contribute to x_u, the respective contributions of e_1 and e_2 to $x_v + x_u$ are $\frac{3}{5}$ and $\frac{2}{5}$, implying that $x_u + x_v \geq 1$.
- When e_1, e_2, and e_3 are all of the form $v \to z$ or $u \to z$: The contributions of e_1, e_2, and e_3 to $x_v + x_u$ are $\frac{3}{5}$, $\frac{2}{5}$, and $\frac{1}{5}$, respectively, and we have $x_u + x_v = \frac{6}{5} > 1$. ◀

2.3 Fractional $\frac{1}{2}(1 + \frac{1}{2^d-1})$-competitiveness for maximum degree d

In this section, we design a $\frac{1}{2}(1 + \frac{1}{2^d-1})$-competitive algorithm for fractional matching and vertex cover in graphs with maximum degree d, assuming that the value d is known to the

Algorithm 2 Fractional matching and vertex cover for graphs of maximum degree d:

Initialize $y \leftarrow 0$ and $x \leftarrow 0$.
When edge $e = (u, v)$ arrives:
Let $0 \leq i \leq d-1$ be the maximal integer for which $\psi_i \leq 1 - \max\{\sum_{e' \in \delta(u)} y_{e'}, \sum_{e' \in \delta(v)} y_{e'}\}$.
Primal update: Set $y_e = \psi_i = \frac{2^i}{2^d - 1}$.
Dual update: Set $x_u \leftarrow x_u + \frac{2^i}{2^d}$ and $x_v \leftarrow x_v + \frac{2^i}{2^d}$.

algorithm in advance. A fractional algorithm should irrevocably assign each arriving edge e a fraction y_e, subject to the constraint that the total sum of fractions assigned to edges emanating from each vertex v can be at most 1, i.e., $\sum_{e \in \delta(v)} y_e \leq 1$. Although the algorithm proposed here deviates from our general Min-Index framework, it has the same flavor. As shown is Algorithm 2, each new edge is assigned a certain fraction, out of d possible values, which is (greedily) chosen as the largest possible such value. In order to simplify subsequent notation, for $i = 0, 1, \ldots, d-1$, let $\psi_i = \frac{2^i}{2^d - 1}$, noting that $\sum_{i=0}^{d-1} \psi_i = 1$.

At first, it is not clear that our algorithm is well-defined, i.e., that upon the arrival of (u, v) an integer $0 \leq i \leq d-1$ satisfying $\psi_i \leq 1 - \max\{\sum_{e' \in \delta(u)} y_{e'}, \sum_{e' \in \delta(v)} y_{e'}\}$ necessarily exists. The next claim proves this property, which is useful for our analysis later on.

▶ **Lemma 5.** *For every vertex u, as long as fewer than d edges adjacent to u have arrived, we have $1 - \sum_{e \in \delta(u)} y_e \geq \frac{1}{2^d - 1}$.*

Proof. Let us focus on a particular point in time, such that at most $d-1$ edges adjacent to u have arrived thus far. For $0 \leq i \leq d-1$, let a_i be the number of edges $e \in \delta(u)$ for which we currently have $y_e = \psi_i$. With this notation, $\sum_{e \in \delta(u)} y_e = \sum_{i=0}^{d-1} a_i \psi_i \leq 1$ and in addition $\sum_{i=0}^{d-1} a_i \leq d-1$.

Given $a_0, \ldots, a_{d-1}$, we define a corresponding sequence $b_0, \ldots, b_{d-1}$ through the following iterative procedure. Initially, $b_i = a_i$ for every i. Then, while there exists an index i with $b_i \geq 2$, we decrease b_i by 2 and increase b_{i+1} by 1. Since $\psi_{i+1} = 2\psi_i$, this operation keeps the sum $\sum_{i=0}^{d-1} b_i \psi_i$ unchanged (and always equal to $\sum_{i=0}^{d-1} a_i \psi_i$) and strictly decreases $\sum_{i=0}^{d-1} b_i$. It is worth noting that we could never have $b_{d-1} \geq 2$, or otherwise $\sum_{i=0}^{d-1} a_i \psi_i = \sum_{i=0}^{d-1} b_i \psi_i \geq 2\psi_{d-1} = 2 \cdot \frac{2^{d-1}}{2^d - 1} > 1$. At the end of this procedure, each of $b_0, \ldots, b_{d-1}$ takes a binary value, and moreover, $\sum_{i=0}^{d-1} b_i \leq \sum_{i=0}^{d-1} a_i \leq d-1$. The desired claim now follows by observing that

$$\sum_{e \in \delta(u)} y_e = \sum_{i=0}^{d-1} a_i \psi_i = \sum_{i=0}^{d-1} b_i \psi_i \leq \sum_{i=1}^{d-1} \psi_i = \sum_{i=1}^{d-1} \frac{2^i}{2^d - 1} = 1 - \frac{1}{2^d - 1} \ ,$$

where the above inequality holds since the binary vector with $\sum_{i=0}^{d-1} b_i \leq d-1$ that maximizes $\sum_{i=0}^{d-1} b_i \psi_i = \frac{1}{2^d - 1} \cdot \sum_{i=0}^{d-1} b_i \cdot 2^i$ is clearly $b_0 = 0$ and $b_1 = \cdots = b_{d-1} = 1$. ◀

▶ **Theorem 6.** *On graphs of maximum degree d, Algorithm 2 is $\frac{1}{2}(1 + \frac{1}{2^d - 1})$-competitive for online fractional matching and fractional vertex cover.*

Proof. First, the fractional matching y returned by the algorithm is feasible, as our choice of ψ_i in each step guarantees that the matching constraints are satisfied. In addition, our dual update rule ensures that the total contribution of the edge (u, v) to the fractional vertex cover value is $\Delta x_u + \Delta x_v = \frac{2^i}{2^{d-1}} = \frac{2^d - 1}{2^{d-1}} \cdot y_e = [\frac{1}{2}(1 + \frac{1}{2^d - 1})]^{-1} \cdot y_e$. Thus, the final fractional matching value is exactly $\frac{1}{2}(1 + \frac{1}{2^d - 1})$ times the fractional vertex cover produced by the algorithm. It remains to prove that the latter is indeed feasible.

For this purpose, let $e = (u, v)$ be an edge that has just arrived. We prove that, after its dual update step, this edge is fractionally covered. For any vertex u, let $y_u = \sum_{e' \in \delta(u)} y_{e'}$ be

the total fractions assigned to edges adjacent to u just before the arrival of the edge e. Since the input graph is guaranteed to be of degree at most d, by Lemma 5, we necessarily assigned y_e with one of the values $\psi_0, \ldots, \psi_{d-1}$. If $y_e = \psi_{d-1}$, then $\Delta x_u = \Delta x_v = \frac{1}{2}$, and we have $x_u + x_v + \Delta x_u + \Delta x_v \geq 1$. In the opposite case, where $y_e = \psi_i$ for some $0 \leq i \leq d-2$, since the edge e could not be assigned with the value ψ_{i+1}, we must have $\max\{y_u, y_v\} > 1 - \frac{2^{i+1}}{2^d-1}$. As $\psi_0, \ldots, \psi_{d-1}$ are all integer multiples of $\frac{1}{2^d-1}$, it follows that y_u and y_v are such multiples as well, meaning that the latter inequality implies $\max\{y_u, y_v\} \geq 1 - \frac{2^{i+1}}{2^d-1} + \frac{1}{2^d-1}$. In addition, our primal and dual update rules ensure that $x_u = \frac{2^d-1}{2^d} \cdot y_u$ and hence, $\max\{x_u, x_v\} = \frac{2^d-1}{2^d} \cdot \max\{y_u, y_v\} \geq \frac{2^d-1}{2^d} \cdot (1 - \frac{2^{i+1}-1}{2^d-1})$. By these observations, after the current update we have

$$x_u + x_v + \Delta x_u + \Delta x_v \geq \max\{x_u, x_v\} + \Delta x_u + \Delta x_v \geq \frac{2^d-1}{2^d} \cdot \left(1 - \frac{2^{i+1}-1}{2^d-1}\right) + \frac{2^{i+1}}{2^d} = 1 \,. \blacktriangleleft$$

3 Upper Bounds for our Framework

In this section, we prove upper bounds on the competitive ratio of the Min-Index algorithm, as stated in the following theorem.

▶ **Theorem 7.** *For any number of matchings $k \geq 1$ and probabilities $p_1, \ldots, p_k$, the Min-Index algorithm is:*

1. *At most $2/3$-competitive on graphs of maximum degree at most 2.*
2. *At most $5/9$-competitive on forest graphs.*
3. *At most $\frac{1}{2}(1 + \frac{1}{2^d-1})$-competitive for bipartite graphs of maximum degree at most d.*

3.1 A $2/3$ upper bound for graphs of maximum degree at most 2

Proof of Theorem 7, Part (1). Consider any instantiation of the Min-Index algorithm that makes use of k matchings with probabilities $p_1, \ldots, p_k$. We define two simple adversarial sequences of edge arrivals.

Sequence 1: A single edge $e = (u, v)$ arrives.

Sequence 2: First, an edge $e = (u, v)$ arrives. Then, two additional edges $e_1 = (u, z)$ and $e_2 = (v, w)$ arrive (in any order).

Clearly, both sequences form graphs of maximum degree at most 2. Let c be the competitive ratio of the algorithm. In Sequence 1, the optimal matching consists of the single edge e, whereas Min-Index adds e to M_1 and obtains a matching with expected cardinality p_1, meaning that $c \leq p_1$. In Sequence 2, the optimal matching consists of e_1 and e_2. However, Min-Index adds e to M_1, and subsequently adds e_1 and e_2 to M_2. As a result, a matching with expected cardinality $p_1 + 2p_2$ is obtained, and therefore $c \leq \frac{p_1}{2} + p_2$. To derive an upper bound on the competitive ratio c, it remains to solve the following linear program, where p_1 and p_2 are treated as probabilities (i.e., required to satisfy $p_1 + p_2 \leq 1$ and $p_1, p_2 \geq 0$):

$$\begin{aligned} \text{maximize} \quad & c \\ \text{subject to} \quad & p_1 \geq c \\ & \frac{p_1}{2} + p_2 \geq c \\ & p_1 + p_2 \leq 1 \\ & p_1, p_2 \geq 0 \end{aligned}$$

The optimal solution to this LP is $p_1 = \frac{2}{3}$, $p_2 = \frac{1}{3}$, and $c = \frac{2}{3}$, concluding our proof. ◀

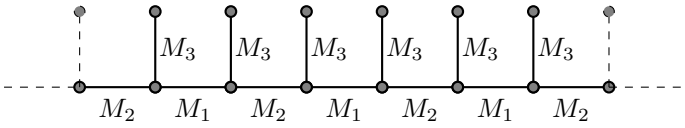


Figure 2 An example for Sequence 2.

▶ Remark. For graphs of maximum degree at most 2, a similar proof actually shows that $2/3$ is the best competitive ratio achievable by any algorithm (not necessarily in our framework), even when the algorithm is allowed to produce a fractional matching.

3.2 A $5/9$ upper bound for forests

Proof of Theorem 7, Part (2). The proof follows the same lines as that of Part (1), with more involved adversarial sequences. Consider an algorithm in our Min-Index framework that makes use of k matchings with probabilities $p_1, \ldots, p_k$. Letting $N = \lceil 1/\epsilon \rceil$, we define the following 3 adversarial sequences of edge arrivals, each forming a tree graph.

Sequence 1: A single edge $e = (u, v)$ arrives.

Sequence 2: In this sequence, a path of length $2N + 1$ is constructed, one edge after the other. By first presenting edges located at even positions of the path, and then those in odd positions, we ensure that the algorithm picks N edges in M_1 and $N + 1$ edges in M_2. Once the entire path is constructed, we add next to each internal vertex u an additional edge (u, v) connecting it to a distinct vertex v. In total, there are $2N$ such edges. The final tree is depicted in Figure 2.

Sequence 3: Figure 3 describes our last sequence. Here, the overall tree is comprised of N copies of a basic gadget, that consists of a 9-edge path with 3 additional edges emanating from middle vertices. Two edges are "going up" from the 5-th and 6-th vertex on the path (marked with bold lines) and another edge is "going down" from the 6-th vertex (marked with dashed lines). The edge arrival sequence proceeds as follows: First, all edges marked with M_1 over all copies arrive, then those marked with M_2, then M_3, and finally M_4. Clearly, Min-Index accepts each edge according to its marked matching, since every edge in M_i is adjacent upon arrival to edges that have already been accepted to $M_1, \ldots, M_{i-1}$. Note that all edges marked with bold lines are accepted to M_4, except for a single edge in the first copy.

Let c be the competitive of the algorithm. To obtain bounds on c in terms of the probabilities $p_1, \ldots, p_4$, for each arrival sequence we compare between the cardinality of the optimal matching and the expected cardinality of the matching produced by the algorithm:

- In Sequence 1, the optimal matching consists of the single edge e, whereas Min-Index adds e to M_1 and obtains a matching with expected cardinality p_1, meaning that $c \leq p_1$.
- In Sequence 2, the optimal matching is composed of all $2N$ edges in M_3, whereas the expected cardinality of the matching returned by Min-Index is $N \cdot p_1 + (N+1) \cdot p_2 + 2N \cdot p_3$. Thus, $c \leq \frac{1}{2}p_1 + \frac{1}{2}p_2 + p_3 + \frac{1}{2N}p_2 \leq \frac{1}{2}p_1 + \frac{1}{2}p_2 + p_3 + \epsilon$, where the last inequality holds since $N = \lceil 1/\epsilon \rceil$.
- In Sequence 3, the optimal matching consists of $6N$ edges, by picking from each gadget the two edges marked in bold and the 1-st, 3-rd, 7-th, and 9-th edges on the path. It is easy to verify that this matching is indeed optimal, as its cardinality is equal to the vertex cover created by picking the 2-nd, 4-th, 5-th, 6-th, 7-th, and 9-th vertices on each path. On the other hand, copies $2, \ldots, N$ of the gadget have 3 edges in M_1, 4 edges in M_2, 3

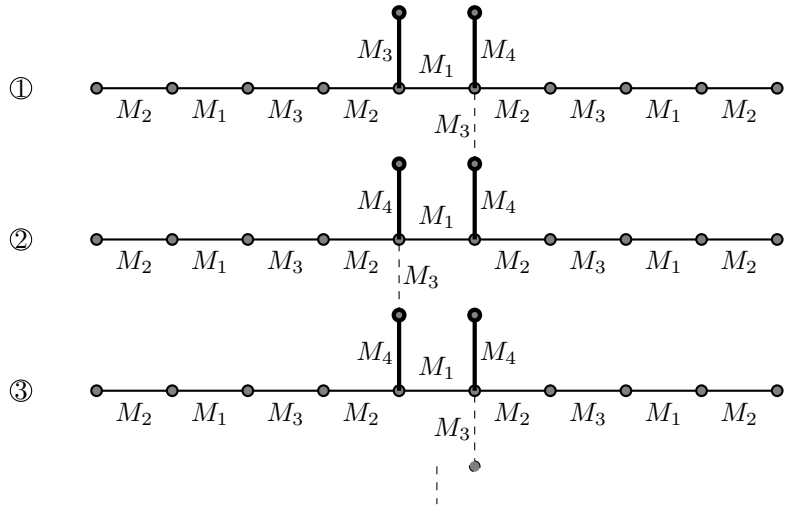


Figure 3 An example for Sequence 3.

edges in M_3, and 2 edges of M_4 each. The first copy has one more edge in M_3 and one less edge in M_4. Thus, the expected cardinality of the matching returned by Min-Index is $N \cdot (3p_1 + 4p_2 + 3p_3 + 2p_4) + p_3 - p_4$. Therefore, $c \leq \frac{1}{2}p_1 + \frac{2}{3}p_2 + \frac{1}{2}p_3 + \frac{1}{3}p_4 + \frac{1}{6N}(p_3 - p_4) \leq \frac{1}{2}p_1 + \frac{2}{3}p_2 + \frac{1}{2}p_3 + \frac{1}{3}p_4 + \epsilon$, where the last inequality holds since $N = \lceil 1/\epsilon \rceil$.

To obtain an upper bound on the competitive ratio c, we now solve the following linear program, where $p_1, \ldots, p_4$ are treated as probabilities:

$$\begin{aligned} \mathrm{LP}(\epsilon) = \text{maximize} \quad & c \\ \text{subject to} \quad & p_1 \geq c \\ & \frac{1}{2}p_1 + \frac{1}{2}p_2 + p_3 + \epsilon \geq c \\ & \frac{1}{2}p_1 + \frac{2}{3}p_2 + \frac{1}{2}p_3 + \frac{1}{3}p_4 + \epsilon \geq c \\ & p_1 + p_2 + p_3 + p_4 \leq 1 \\ & p_1, p_2, p_3, p_4 \geq 0 \end{aligned}$$

It is easy to verify that the optimal solution to $\mathrm{LP}(\epsilon)$ has $c \leq \frac{5}{9} + \epsilon$. To see this, note that $\mathrm{LP}(0) \leq \mathrm{LP}(\epsilon) \leq \mathrm{LP}(0) + \epsilon$ and that the optimal solution to $\mathrm{LP}(0)$ is given by $p_1 = \frac{5}{9}$, $p_2 = \frac{3}{9}$, $p_3 = \frac{1}{9}$, $p_4 = 0$, and $c = \frac{5}{9}$. We conclude that $\frac{5}{9}$ is the best competitive ratio achievable through the Min-Index framework, even for trees with maximum degree 4. ◀

3.3 A $\frac{1}{2}(1 + \frac{1}{2^d - 1})$ upper bound for bipartite graphs

Proof of Theorem 7, Part (3). Consider an algorithm in our Min-Index framework that makes use of k matchings with probabilities $p_1, \ldots, p_k$. Given an integer parameter d, we define the following d adversarial sequences of edge arrivals, each forming a bipartite graph of maximum degree at most d:

Sequence 1: A single edge $e = (u, v)$ arrives.

Sequences $\ell = 2, \ldots, d$: Let G be an $(\ell - 1)$-regular bipartite graph, with n vertices on each side. It is well-known, as an immediate corollary of Hall's Marriage Theorem, that the edge set of such graphs can be partitioned into $\ell - 1$ perfect matchings. In the

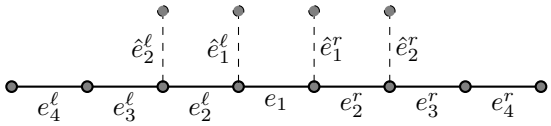


Figure 4 A sequence terminated at round $n = 4$.

sequence of edge arrivals, these matchings are presented one after the other, with an arbitrary order for the edges within each matching. Clearly, Min-Index accepts the first matching into M_1, the second into M_2, and so on. Next, we create a new edge emanating from each of the $2n$ vertices into a new distinct vertex. As these edges are disjoint and cannot be accepted to any of the matchings $M_1, \ldots, M_{\ell-1}$, they are all accepted into M_ℓ.

Let c be the competitive of the algorithm. To obtain bounds on c in terms of the probabilities $p_1, \ldots, p_k$, for each arrival sequence we compare between the cardinality of the optimal matching and the expected cardinality of the matching produced by the algorithm:

- In Sequence 1, the optimal matching consists of the single edge e, whereas Min-Index adds e to M_1 and obtains a matching with expected cardinality p_1, meaning that $c \leq p_1$.
- In Sequence $2 \leq \ell \leq d$, the optimal matching consists of all $2n$ edges in M_ℓ, whereas the expected cardinality of the matching returned by Min-Index is $n \cdot \sum_{t=1}^{\ell-1} p_t + 2n \cdot p_\ell$. Therefore, $c \leq \frac{1}{2} \cdot \sum_{t=1}^{\ell-1} p_t + p_\ell$.

Multiplying both sides of the upper bound due to Sequence ℓ by $\frac{1}{2^{d-\ell}}$, and summing the resulting inequalities over all $1 \leq \ell \leq d$, we get $\sum_{\ell=1}^{d} p_\ell \geq c \cdot \sum_{\ell=1}^{d} \frac{1}{2^{d-\ell}} = c \cdot (2 - \frac{1}{2^{d-1}})$. Since $\sum_{\ell=1}^{d} p_\ell \leq 1$, it follows that the competitive ratio satisfies $c \leq (2 - \frac{1}{2^{d-1}})^{-1} = \frac{1}{2}(1 + \frac{1}{2^{d}-1})$. ◀

4 Upper Bound for any Algorithm

In this section, we present our general upper bound, formally stated in Theorem 2. In particular, we prove that the competitive ratio of any fractional (or randomized) online algorithm for maximum matching is at most $\frac{2}{3+1/\phi^2} \approx 0.5914$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. In fact, this result holds even in the vertex arrival model, when the underlying graph is a tree of maximum degree 3.

We first note that any randomized algorithm induces a marginal expected value of y_e for accepting each edge e. As these marginal values must satisfy the packing constraints of the standard matching linear program (P), shown in Figure 1, they induce a valid fractional algorithm. Therefore, proving an upper bound for fractional online algorithms suffices.

Arrival sequence. To understand the upcoming construction, we advise the reader to consult Figure 4. Consider an adversarial sequence consisting of $2n - 1$ edges (and $2n$ vertices) that eventually forms a path as follows. In the first round, an edge $e_1 = (v_1^\ell, v_1^r)$ arrives. Then, for $i \geq 2$, the i-th round introduces two edges of the form $e_i^\ell = (v_i^\ell, v_{i-1}^\ell)$ and $e_i^r = (v_{i-1}^r, v_i^r)$, that augment the path on both sides. The adversary may terminate the sequence once round n ends. Terminating the sequence for any $n \geq 3$ is done by introducing $2(n-2)$ additional "leaf edges", adjacent to the inner vertices $v_1^\ell, \ldots, v_{n-2}^\ell$ and $v_1^r, \ldots, v_{n-2}^r$. The leaf edges adjacent to v_i^ℓ and v_i^r are denoted by $\hat{e}_i^\ell$ and $\hat{e}_i^r$, respectively.

Upper bound as a linear program. Consider any fractional algorithm. For an edge e, let y_e be the fraction given to this edge. In addition, for $i \geq 2$, the sum of fractions given to the edges e_i^ℓ and e_i^r is denoted by $y_i = y_{e_i^\ell} + y_{e_i^r}$; it is convenient to denote $y_1 = y_{e_1}$ as well.

First observe that, since the algorithm is required to meet the matching constraints $\sum_{e\in\delta(v)} y_e \leq 1$ at any point in time, as soon as e_2^ℓ and e_2^r arrive we must have $y_{e_1} + y_{e_2^\ell} \leq 1$ and $y_{e_1} + y_{e_2^r} \leq 1$. By adding up these inequalities, it follows that

$$2y_1 + y_2 \leq 2 \ . \tag{1}$$

Based on precisely the same logic, for $i \geq 2$, once e_{i+1}^ℓ and e_{i+1}^r arrive we would get $y_{e_i^\ell} + y_{e_{i+1}^\ell} \leq 1$ and $y_{e_i^r} + y_{e_{i+1}^r} \leq 1$, implying in turn that

$$y_i + y_{i+1} \leq 2 \ . \tag{2}$$

Now let c be the competitive ratio of the algorithm. After rounds 1 and 2, the optimal matchings are of cardinality 1 and 2, respectively, and therefore

$$c \leq y_1 \quad \text{and} \quad c \leq \frac{1}{2}\left(y_1 + y_2\right) \ . \tag{3}$$

In addition, if the adversarial sequence ends at round $n \geq 3$, the matching constraints due to the inner vertices $v_1^\ell, \ldots, v_{n-2}^\ell$ and $v_1^r, \ldots, v_{n-2}^r$ lead to the aggregate inequality

$$2(n-2) \geq \sum_{i=1}^{n-2} \left(\sum_{e\in\delta(v_i^\ell)} y_e + \sum_{e\in\delta(v_i^r)} y_e \right) = y_{n-1} + 2\cdot\sum_{i=1}^{n-2} y_i + \sum_{i=1}^{n-2} \left(y_{\hat{e}_i^\ell} + y_{\hat{e}_i^r} \right) \ .$$

Consequently, it follows that the total fractions assigned by the algorithm to all edges is

$$\sum_{i=1}^{n} y_i + \sum_{i=1}^{n-2} \left(y_{\hat{e}_i^\ell} + y_{\hat{e}_i^r} \right) \leq \sum_{i=1}^{n} y_i + 2(n-2) - y_{n-1} - 2\cdot\sum_{i=1}^{n-2} y_i = 2(n-2) + y_n - \sum_{i=1}^{n-2} y_i \ .$$

However, the optimal matching consists of $2(n-1)$ edges: $e_n^\ell, e_n^r, \hat{e}_1^\ell, \ldots, \hat{e}_{n-2}^\ell, \hat{e}_1^r, \ldots, \hat{e}_{n-2}^r$. Thus, we get the following upper bound on the competitive ratio c:

$$c \leq \frac{1}{2(n-1)} \cdot \left(2(n-2) + y_n - \sum_{i=1}^{n-2} y_i \right) \qquad \forall n \geq 3 \tag{4}$$

To summarize, the competitive ratio is upper bounded by the supremum value of c that satisfies Inequalities (1), (2), (3), and (4), noting that the latter actually provides a separate inequality for each $n \geq 3$. Therefore, any finite subset of these inequalities provides a concrete upper bound on the value c. For every $m \geq 4$, let c_m be the bound attained by the following (finite) LP, consisting of a subset of the constraints that are equivalent to truncating the input sequence after m rounds:

$$\begin{aligned}
c_m = \text{maximize} \quad & c \\
\text{subject to} \quad & c \leq y_1 && (5)\\
& c \leq \frac{1}{2}\left(y_1 + y_2\right) && (6)\\
& c \leq \frac{1}{2(n-1)} \cdot \left(2(n-2) + y_n - \sum_{i=1}^{n-2} y_i \right) \qquad \forall n = 3, \ldots, m && (7)\\
& y_{m-1} + y_m \leq 2 && (8)
\end{aligned}$$

▶ **Lemma 8.** *$c_m = \frac{2F_{m+1}-2}{3F_{m+1}+F_{m-1}-4}$, where F_m is the m-th Fibonacci number.*

Due to space limitations, we omit the proof. As the competitive ratio of any algorithm is at most c_m for any $m \geq 4$, and $\lim_{m\to\infty} \frac{F_{m-1}}{F_{m+1}} = \frac{1}{\phi^2}$, we conclude the proof by observing that, $\lim_{m\to\infty} c_m = \lim_{m\to\infty} \frac{2F_{m+1}-2}{3F_{m+1}+F_{m-1}-4} = \frac{2}{3+1/\phi^2} \approx 0.591372$.

References

1. Gagan Aggarwal, Gagan Goel, Chinmay Karande, and Aranyak Mehta. Online vertex-weighted bipartite matching and single-bid budgeted allocations. In *Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1253–1264, 2011.
2. Yossi Azar, Ilan Reuven Cohen, and Alan Roytman. Online lower bounds via duality. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1038–1050, 2017.
3. Niv Buchbinder, Kamal Jain, and Joseph Naor. Online primal-dual algorithms for maximizing ad-auctions revenue. In *Proceedings of the 15th Annual European Symposium on Algorithms*, pages 253–264, 2007.
4. Ashish Chiplunkar, Sumedh Tirodkar, and Sundar Vishwanathan. On randomized algorithms for matching in the online preemptive model. In *Proceedings of the 23rd Annual European Symposium on Algorithms*, pages 325–336, 2015.
5. Nikhil R. Devanur and Thomas P. Hayes. The adwords problem: online keyword matching with budgeted bidders under random permutations. In *Proceedings 10th ACM Conference on Electronic Commerce*, pages 71–78, 2009.
6. Nikhil R. Devanur and Kamal Jain. Online matching with concave returns. In *Proceedings of the 44th ACM Symposium on Theory of Computing*, pages 137–144, 2012.
7. Nikhil R. Devanur, Kamal Jain, and Robert D. Kleinberg. Randomized primal-dual analysis of RANKING for online bipartite matching. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 101–107, 2013.
8. Leah Epstein, Asaf Levin, Danny Segev, and Oren Weimann. Improved bounds for online preemptive matching. In *Proceedings of the 30th International Symposium on Theoretical Aspects of Computer Science*, pages 389–399, 2013.
9. Jon Feldman, Aranyak Mehta, Vahab S. Mirrokni, and S. Muthukrishnan. Online stochastic matching: Beating $1-1/e$. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 117–126, 2009.
10. Gagan Goel and Aranyak Mehta. Online budgeted matching in random input models with applications to adwords. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 982–991, 2008.
11. Guru Prashanth Guruganesh and Sahil Singla. Online matroid intersection: Beating half for random arrival. Preprint, arXiv:1512.06271, 2015.
12. Bala Kalyanasundaram and Kirk Pruhs. An optimal deterministic algorithm for online b-matching. *Theoretical Computer Science*, 233(1-2):319–325, 2000.
13. Chinmay Karande, Aranyak Mehta, and Pushkar Tripathi. Online bipartite matching with unknown distributions. In *Proceedings of the 43rd ACM Symposium on Theory of Computing*, pages 587–596, 2011.
14. Richard M. Karp, Umesh V. Vazirani, and Vijay V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 352–358, 1990.
15. Mohammad Mahdian and Qiqi Yan. Online bipartite matching with random arrivals: an approach based on strongly factor-revealing lps. In *Proceedings of the 43rd ACM Symposium on Theory of Computing*, pages 597–606, 2011.
16. Vahideh H. Manshadi, Shayan Oveis Gharan, and Amin Saberi. Online stochastic matching: Online actions based on offline statistics. *Mathematics of Operations Research*, 37(4):559–573, 2012.
17. Aranyak Mehta. Online matching and ad allocation. *Foundations and Trends in Theoretical Computer Science*, 8(4):265–368, 2013.
18. Aranyak Mehta, Amin Saberi, Umesh V. Vazirani, and Vijay V. Vazirani. Adwords and generalized online matching. *Journal of the ACM*, 54(5):22, 2007.

19 Sumedh Tirodkar and Sundar Vishwanathan. Maximum matching on trees in the online preemptive and the incremental dynamic graph models. Preprint, arXiv:1612.05419, 2016.

20 Yajun Wang and Sam Chiu-wai Wong. Two-sided online bipartite matching and vertex cover: Beating the greedy algorithm. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming*, pages 1070–1081, 2015.

Computing Optimal Homotopies over a Spiked Plane with Polygonal Boundary*

Benjamin Burton[1], Erin Chambers[2], Marc van Kreveld[3], Wouter Meulemans[4], Tim Ophelders[5], and Bettina Speckmann[6]

1 **School of Mathematics and Physics, University of Queensland, Brisbane, Australia**
bab@maths.uq.edu.au
2 **Dept. of Computer Science, Saint Louis University, Saint Louis, MO, USA**
echambe5@slu.edu
3 **Dept. of Information and Computing Sciences, Utrecht University, The Netherlands**
m.j.vankreveld@uu.nl
4 **Dept. of Mathematics and Computer Science, TU Eindhoven, The Netherlands**
w.meulemans@tue.nl
5 **Dept. of Mathematics and Computer Science, TU Eindhoven, The Netherlands**
t.a.e.ophelders@tue.nl
6 **Dept. of Mathematics and Computer Science, TU Eindhoven, The Netherlands**
b.speckmann@tue.nl

Abstract

Computing optimal deformations between two curves is a fundamental question with various applications, and has recently received much attention in both computational topology and in mathematics in the form of homotopies of disks and annular regions. In this paper, we examine this problem in a geometric setting, where we consider the boundary of a polygonal domain with *spikes*, point obstacles that can be crossed at an additive cost. We aim to continuously morph from one part of the boundary to another, necessarily passing over all spikes, such that the most expensive intermediate curve is minimized, where the cost of a curve is its geometric length plus the cost of any spikes it crosses.

We first investigate the general setting where each spike may have a different cost. For the number of inflection points in an intermediate curve, we present a lower bound that is linear in the number of spikes, even if the domain is convex and the two boundaries for which we seek a morph share an endpoint. We describe a 2-approximation algorithm for the general case, and an optimal algorithm for the case that the two boundaries for which we seek a morph share both endpoints, thereby representing the entire boundary of the domain.

We then consider the setting where all spikes have the same unit cost and we describe a polynomial-time exact algorithm. The algorithm combines structural properties of homotopies arising from the geometry with methodology for computing Fréchet distances.

1998 ACM Subject Classification I.3.5 Computational Geometry and Object Modeling

Keywords and phrases Fréchet distance, polygonal domain, homotopy, geodesic, obstacle

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.23

* This research was initiated at Dagstuhl seminar 17072, "Applications of Topology to the Analysis of 1-Dimensional Objects". E. Chambers is supported by NSF CCF-1614562 and CCF-1054779; W. Meulemans by NLeSC grant 027.015.G02; and T. Ophelders and B. Speckmann by NWO grant 639.023.208.

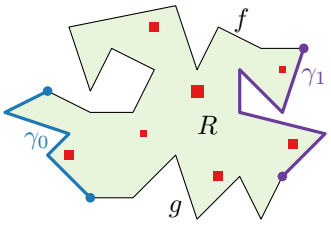


Figure 1 Polygonal domain R bounded by polylines $f \cup g \cup \gamma_0 \cup \gamma_1$. K in red squares.

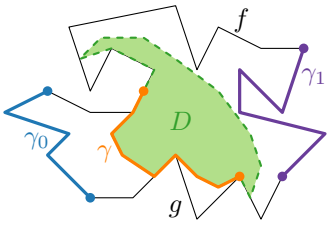


Figure 2 Disk D is convex relative to R. γ is backwards- but not forwards-convex.

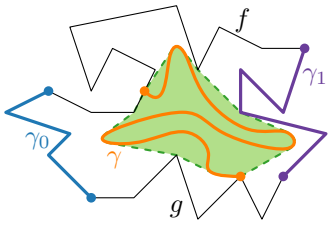


Figure 3 The convex hull $\mathrm{gh}_R(\gamma)$ of a curve $\gamma \in \Gamma$.

1 Introduction

Computing optimal deformations between two input curves is a fundamental building block in many application areas, such as robotics, motion planning, geographic information systems, and graphics. Such deformations or morphs have several salient properties, for example, the maximum necessary length of intermediate curves. The properties of an optimal morph between two curves also serve as a natural measure of similarity between these curves. Examples of such measures include the Fréchet distance and its variants.

In this paper we consider the following scenario: our input is a polygonal domain with point obstacles, similar to [6]. We aim to continuously morph from one part of the boundary to another. The intermediate curves (or leashes) are allowed to pass over the point obstacles, albeit for a fixed cost. We hence refer to the point obstacles as *spikes* to reinforce the intuition that encountering them is costly, but that they do not form impassable barriers. Our goal is to minimize the cost of the most expensive leash during the morph, where the cost of a leash is defined as its length plus the cost of the spikes it encounters. We consider both variable-cost spikes and unit-cost spikes, and describe several structural results as well as algorithms. In the following we first introduce some necessary definitions which allow us to state our our results more precisely.

Definitions and problem statement. Let $f, g, \gamma_0, \gamma_1 \colon [0,1] \to \mathbb{R}^2$ be four interior-disjoint simple polylines (possibly of length 0) in the plane, with $f(0) = \gamma_0(0)$, $f(1) = \gamma_1(0)$, $g(0) = \gamma_0(1)$, and $g(1) = \gamma_1(1)$, whose union bounds a polygonal domain R of n vertices. Let $K \subset R$ be a finite set of k point-obstacles which we call *spikes* (see Figure 1). The *cost* of a spike is given by a function $w \colon K \to \mathbb{R}^{\geq 0}$. We assume that spikes lie in general position in the sense that no three spikes lie on the same geodesic in R. Standard perturbation techniques (as also described below) can lift this assumption.

Let Γ be the family of simple curves in R from f to g; that is, all curves $\gamma \colon [0,1] \to R$ with $\gamma(0) \in \mathrm{Im}(f)$ and $\gamma(1) \in \mathrm{Im}(g)$. A *homotopy* on R from γ_0 to γ_1 is a continuous map $h \colon [0,1] \times [0,1] \to R$ with $h(0,\cdot) = \gamma_0$, $h(1,\cdot) = \gamma_1$. All homotopies we consider have $h(t,\cdot) \in \Gamma$ for all $t \in [0,1]$. With slight abuse of terminology, we refer to these simply as homotopies. We call a homotopy *monotone* if it is injective after infinitesimal perturbation. We refer to each curve $\gamma_t \colon p \mapsto h(t,p)$ as the *leash* of h at time t, and define the *cost* of a leash as its length plus the total cost of spikes in K (with multiplicity) it encounters. The *cost* of a homotopy is the cost of its maximum-cost leash. We are interested in the minimum-cost homotopy on R from γ_0 to γ_1 and refer to the cost of this homotopy as the *homotopy height* from γ_0 to γ_1 on (R, K, w).

For two curves γ and γ' from a monotone homotopy, denote the region between them by $R(\gamma, \gamma')$; that is, $R(\gamma, \gamma')$ is the (possibly degenerate) topological disk bounded by γ, γ', the arc of f between $\gamma(0)$ and $\gamma'(0)$, and the arc of g between $\gamma(1)$ and $\gamma'(1)$. For a simple curve $\gamma \in \Gamma$, define its *swept region* as $R(\gamma_0, \gamma)$ and symmetrically define its *unswept region* as $R(\gamma, \gamma_1)$. We call a region $D \subseteq R$ *convex relative to* R if for any two points in D, the shortest path in R connecting those points also lies in D [20]; for this definition we do not not charge the additional cost of any spikes. A curve $\gamma \in \Gamma$ is *forwards-convex* if it lies on the boundary of a region D convex relative to R, and D is contained in the swept region of γ. Symmetrically, γ is *backwards-convex* if it lies on the boundary of a region D convex relative to R, and D is contained in the unswept region of γ (see Figure 2). A curve may be both forwards- and backwards-convex; a shortest path is always both.

Region R is convex relative to itself, and the intersection of any two convex sets relative to R is also convex relative to R. For $\gamma \in \Gamma$ define its convex hull (also known as geodesic hull) $\mathrm{gh}_R(\gamma)$ relative to R as the unique minimal region containing γ which is convex relative to R (see Figure 3).

Consider a homotopy h that contains a leash γ_t that crosses several spikes. Infinitesimal perturbation of the leash at the spikes ensures that γ_t no longer crosses a spike, but is then forced into some homotopy class. In particular, this has two implications: (1) as the perturbation tends to zero, this has no effect on the length of the leash and thus, strictly speaking, the optimal homotopy is an infimum rather than a minimum if the given leash is the maximum-cost one in the optimal homotopy; (2) we can decompose a single leash crossing a number of spikes into a homotopy crossing each spike separately, essentially holding the leash fixed—hence an optimal homotopy exists that crosses spikes one at the time.

Results and organization. We consider various settings of a spiked plane with polygonal boundary. In Section 2, we investigate the general setting where each spike may have a different cost. First, we consider the number of inflection points that the leash may need in an optimal homotopy and present a lower bound that is linear in the number of spikes, even if R is convex and only f has positive length. We then present a 2-approximation algorithm for the general case and an optimal algorithm for computing the homotopy height for the case that f and g have length 0 (i.e., γ_1 and γ_2 together form the boundary of R).

In Section 3, we consider the setting where all spikes have the same unit cost. Here we present our main result: an algorithm to compute the exact homotopy height in polynomial time. The algorithm combines structural properties of homotopies arising from the geometry with methodology for computing Fréchet distances. To the best of our knowledge, these are the first polynomial-time algorithms to compute the exact homotopy height in any setting.

Related work. Chambers *et al.* [5] recently proved that there always is a minimum-cost homotopy between the boundaries of an annular surface that is an isotopy and monotone, that is, the intermediate leashes never move "backwards". This proof readily transfers to our setting and is indeed supporting our results as described in Section 3.

Homotopy height was introduced independently in the computational geometry community [7] and in the combinatorics community [4]. On triangulated surfaces, the best known algorithm gives an $O(\log n)$ approximation, where n is the complexity of the surface [18]. More recently, it has also been studied in more general settings, where instead of having point obstacles, obstacles are modeled by assigning a non-Euclidean metric to R [10, 11, 12].

Fréchet distance is a well studied metric. Also known as the dog-leash distance, the goal is to minimize the length of the longest leash connecting a man walking along one curve and a dog along the other, where the man and dog walk monotonically along the curves.

The classic algorithm computes this metric in $O(n^2 \log n)$ time [1], with many variants and approximation algorithms having been studied since. The geodesic and homotopic Fréchet distance are particularly related to our setting. The former is a variant where the leashes must stay inside a polygonal boundary and remain geodesic, but no obstacles are present inside the polygon [13]. For the latter, point or polygonal obstacles are given which no leash may cross: this case can be solved in polynomial time [6].

Optimal morphs have also been studied in a variety of other settings. From the topology end, minimum-area homotopies can be computed for planar or surface embedded curves in polynomial time [9, 17, 19]. Minimum-area homologies are a closely related similarity measure on curves that can be computed very quickly using linear algebra packages [8, 14], but do not yield intuitive deformations or morphings in the same way as homotopy.

From the geometry and graph theory communities, much work has been done on computing morphs between inputs; indeed, it is well known that any two drawings of the same planar graph can be morphed to each other. Optimal morphs between such graphs are still being studied, including work that bounds the complexity of the morph [2, 3]. However, none these morphs bound the length of any of the "leashes" tracing the paths of vertices during the morph. Morphs based on geodesic width [16] force all intermediate curves to not cross the input curves (which are part of the boundary polygon in our setting); however, none of these have been considered in the presence of obstacles. Dynamic time warping and related concepts [15] also consider ways to match and morph curves, but again do not extend to more general settings.

2 Variable-cost spikes

In this section, we consider the setting where each spike may have a different cost. As we prove, the variable costs have a profound effect on the leash complexity. Nonetheless, we obtain a general approximation algorithm as well as an optimal algorithm for a special case.

Leash complexity. We define the *complexity* of a leash as the number of inflection points that are not caused by the boundary of R. In particular, the leash complexity needed for an optimal homotopy is defined by the number of spikes that cause an inflection point. Unfortunately, in the general case, the complexity may be linear, even when R is convex. Correspondingly, we do not have a polynomial-time algorithm even when R is convex.

▶ **Lemma 1.** *In the worst case, the leash complexity for an optimal homotopy for* (R, K, w) *is* $\Omega(|K|)$*, even if* γ_0*,* γ_1 *and* g *have length* 0 *and* f *is convex.*

Proof. For ease of exposition, we first argue the general case, using the construction illustrated in Figure 4(a). We have n spikes, $K = k_1, \ldots, k_n$, lined up in the middle, at vertical distance 1 from each other and f and g; k_1 is the highest- and k_n the lowest-positioned spike. Moreover, the odd-numbered spikes have a closest point on the first half of f and the even-numbered spikes have a closest point on the second half of f; these closest points are at distance 0.75. This implies that the optimal leash crossing k_i has cost $n - i + 1.75 + w(k_i)$. By setting $w(k_i) = c + i$, all these leashes have the same cost, for suitable constant $c > 0$, depending on the longest leash necessary that does not cross a spike. Now, the homotopy height of this instance is $1.75 + w(k_n) = 1.75 + c + n$. This requires the leashes to cross the spikes in the order of their closest points along f, that is, odd-numbered before even-numbered spikes (see Figure 4(b)). When crossing spike k_1, the leash has crossed all other odd-numbered spikes, but none of the even-numbered spikes. Thus, the leash has a linear number of inflection points, if we perturb all odd-numbered spikes slightly.

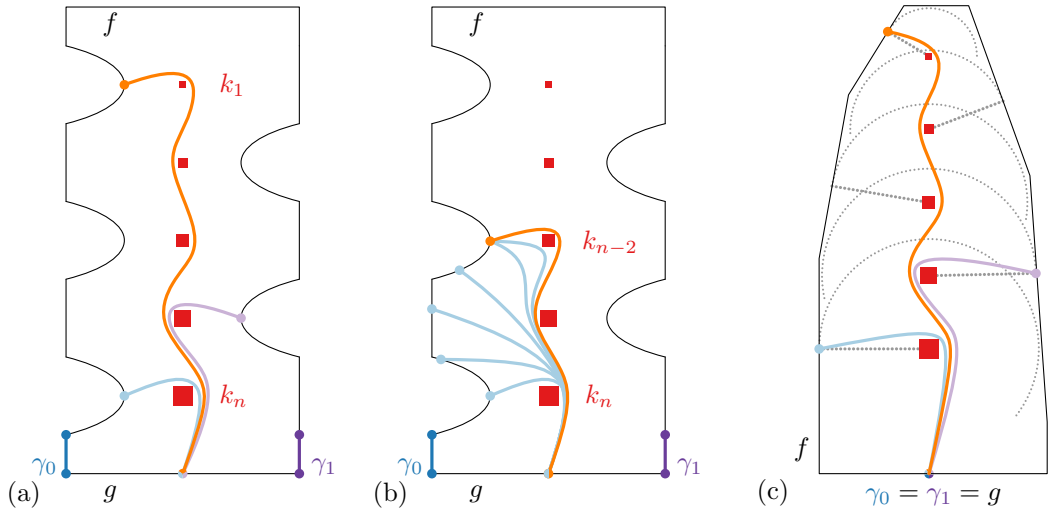


Figure 4 (a) A leash may need linear complexity when considering variable-cost spikes. (b) Part of the optimal homotopy, after crossing k_n up to crossing k_{n-2}. (c) This even holds in the convex case, with one boundary path and the two initial leashes having length 0. The closest point and corresponding distance circles are indicated for each spike.

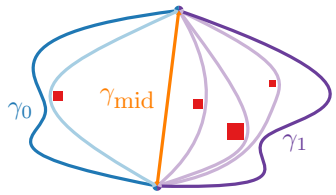


Figure 5 When f and g have length 0, we can apply a simple greedy strategy to shrink both γ_0 and γ_1 onto γ_{mid}.

A similar construction can be made when we require that f is convex and γ_0, γ_1 and g have length 0. This is illustrated in Figure 4(c). The same principle applies: we position spikes such that their closest point is alternatingly on the left half and right half of f. By setting the weights appropriately, we can again ensure that the optimal homotopy must cross the spikes in some order along f and force a linear number of inflection points. ◀

Algorithms. If f and g collapse onto a point, we can compute the homotopy height in polynomial time with a greedy algorithm. Interestingly, this contrasts the potential complexity of the problem if γ_0 and γ_1 and even g collapse onto a point, as suggested by the lower bound in Lemma 1.

▶ **Lemma 2.** *We can compute in polynomial time the homotopy height of (R, K, w), if f and g have length 0.*

Proof. Consider the geodesic leash γ_{mid} between $f(0)$ and $g(0)$, ignoring any spikes; see Figure 5. As described below, we greedily shrink γ_0 until we reach γ_{mid}. We first compute the geodesic leash γ_t between $f(0)$ and $g(0)$ in the same homotopy class as γ_0. By definition, γ_t cannot be longer than γ_0. Then, we cross the minimal-cost spike $k \in K \cap \gamma_t$, resulting in a cost $\|\gamma_t\| + w(k)$. We repeat the process from γ_t, until we reach γ_{mid}. Analogously,

we shrink γ_1 to γ_{mid}. The maximum of $\|\gamma_0\|$, $\|\gamma_1\|$ and all intermediate $\|\gamma_t\| + w(k)$ is the homotopy height of (R, K, w).

As all intermediate leashes from γ_0 to γ_{mid} are backwards-convex, these leashes grow only shorter. In particular, this implies that we cannot make a leash on some spike k shorter, by first crossing other spikes that are not on the leash but in the unswept area. In other words, we cannot improve the cost by crossing a spike k on a geodesic by first crossing spikes that are not on the geodesic. Hence, the greedy choice is optimal. ◀

The above proof readily implies that the longest leash in the optimal homotopy is determined by the initial and final leash, and thus results in the following lemma. Note that we are interested here only in the geometric length, excluding any spikes. We capture this in a separate lemma as it supports the unit-cost case, detailed in the next section.

▶ **Lemma 3.** *If $l \in \Gamma$ is backwards-convex and $r \in \Gamma$ is forwards-convex, with $l(0) = r(0)$, $l(1) = r(1)$, such that l and r together bound a region D convex relative to R, then there is a monotone homotopy from l to r consisting of only backwards- and forwards-convex leashes, and whose longest leash has length* $\max\{\|l\|, \|r\|\}$.

For the general case, there is also a simple 2-approximation achievable, by using the algorithm for the geodesic Fréchet distance.

▶ **Lemma 4.** *We can compute in $O(|R|^2 \log^2 |R|)$ time a 2-approximation of the homotopy height of (R, K, w).*

Proof. The algorithm computes the geodesic Fréchet distance [13] in R, that is, ignoring the spikes. Consider the optimal geodesic Fréchet matching μ. We may extend μ into a homotopy h by infinitesimal perturbation to cross only one spike at once and by shortening γ_0 and γ_1 to the geodesics between $f(0)$ and $g(0)$ and between $f(1)$ and $g(1)$ respectively. We prove that h is a 2-approximation of the minimal-cost homotopy h^* including the spikes.

Let c be the longest leash in h and let $w_{\max}$ denote the maximal cost of a spike. The cost of h is bounded by $c + w_{\max}$. Either c is defined by γ_0 or γ_1 (which must be in any homotopy) or c is defined by a leash in μ: in either case, c provides a lower bound on the maximal leash length in h'. Moreover, h' must also cross the maximal-cost spike. Hence, the cost of h' is at least $\max\{c, w_{\max}\}$. We now have that $c + w_{\max} \leq 2 \cdot \max\{c, w_{\max}\}$, thus proving that h is a 2-approximation of h^*. ◀

3 Unit-cost spikes

In this section, we give an algorithm to compute the homotopy height in the case where all spikes have cost 1. We start by proving properties on the homotopy classes and lengths of curves in Γ. These properties allow us to construct for any homotopy, a homotopy of similar cost with a regular structure. Finally, we show how to decide the existence of such a regular homotopy cheaper than a given cost in polynomial time.

Shortcutting curves. Consider a curve $\gamma \in \Gamma$ and let $D = \text{gh}(\gamma)$ be its convex hull. Let l and r respectively be the backwards- and forwards-convex arcs of ∂D between the endpoints of γ. Consider an arc φ of $\gamma \setminus l$ or $\gamma \setminus r$. Let $\bar{\varphi}$ be the corresponding arc of l or r between the endpoints of φ. Then we refer to the disk bounded by $\varphi \cup \bar{\varphi}$ as a *pocket* of γ, and refer to $\bar{\varphi}$ as its *lid*, see Figure 6. Each lid is a shortest path in R, and the pockets of γ partition D.

▶ **Lemma 5.** *Let γ and $\gamma' \in \Gamma$ be two non-crossing simple curves. Each arc ψ of $\gamma' \cap \text{gh}(\gamma)$ has both endpoints on the same lid of the containing pocket of γ.*

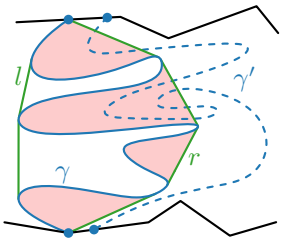


Figure 6 Pockets of γ with lids on r (shaded).

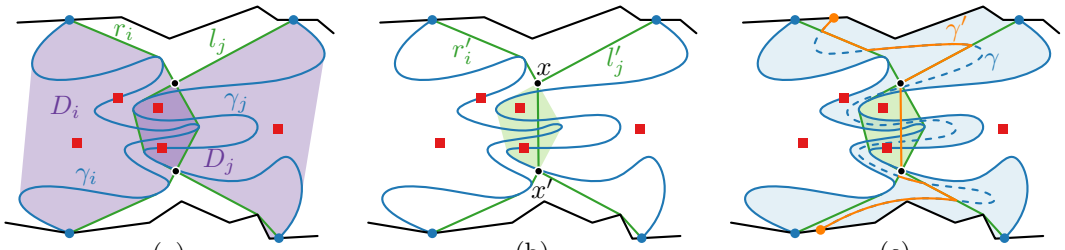


Figure 7 (a) The curves r_i and l_j on ∂D_i and ∂D_j, respectively. (b) The curves r'_i and l'_j obtained after replacing arcs with the geodesic between x and x'. (c) An example curve γ' in the homotopy class of r'_i that is shorter than γ (dashed).

Proof. Since the endpoints of γ' lie on the boundary of R, ψ must have both endpoints on the boundary of its pocket. Since ψ does not cross γ, and the pocket is bounded by an arc of γ and a lid, the endpoints of ψ lie on the lid. ◀

Consider a forwards-convex and a backwards-convex curve in Γ. If these curves intersect, they intersect in at most two points or geodesics, and occur in the same order and direction along the curves. As such, their first and last point of intersection are naturally well defined.

▶ **Lemma 6.** *Let γ_i and $\gamma_j \in \Gamma$ be two non-intersecting simple curves, and assume γ_i lies in the swept region of γ_j. Let $D_i = \mathrm{gh}(\gamma_i)$ and $D_j = \mathrm{gh}(\gamma_j)$ be their convex hulls. Let $r_i \in \Gamma$ be the forwards-convex arc of ∂D_i and let $l_j \in \Gamma$ be the backwards-convex arc of ∂D_j. If r_i and l_j intersect, let x and x' be the first and last points of intersection of r_i and l_j. Let r'_i and l'_j be the curves obtained from r_i and l_j by replacing their arcs in $D_i \cap D_j$ by the geodesic between x and x'. If r_i and l_j do not intersect, let $r'_i = r_i$ and $l'_j = l_j$. Assume the region between γ_i and γ_j contains no spikes in its interior and consider a third simple curve γ in this region. There is a curve γ' with the same endpoints as γ and $\|\gamma'\| \leq \|\gamma\|$, such that γ' lies in the homotopy class of r'_i and l'_j.*

Proof. The setup is illustrated in Figure 7. We consider three cases illustrated in Figure 8, depending on the number of bends of r_i and l_j on $\partial(D_i \cap D_j)$ that are induced by γ_i and γ_j.

(a) In the first case, assume there are no such bends on r_i or l_j, then the interior of $D_i \cap D_j$ is empty. Since D_i and D_j are disjoint, so are the pockets of γ_i and γ_j. If we replace all arcs of γ that lie in pockets of γ_i or γ_j by the geodesic between the endpoints on the corresponding lid, then we obtain a curve γ' between r'_i and l'_j with $\|\gamma'\| \leq \|\gamma\|$. Since there are no spikes between r'_i and l'_j, γ' lies in the same homotopy class as r'_i.

(b) In the second case, assume either r_i or l_j has no such bend on $\partial(D_i \cap D_j)$, but the other has at least one bend. Without loss of generality, assume that r_i has at least one

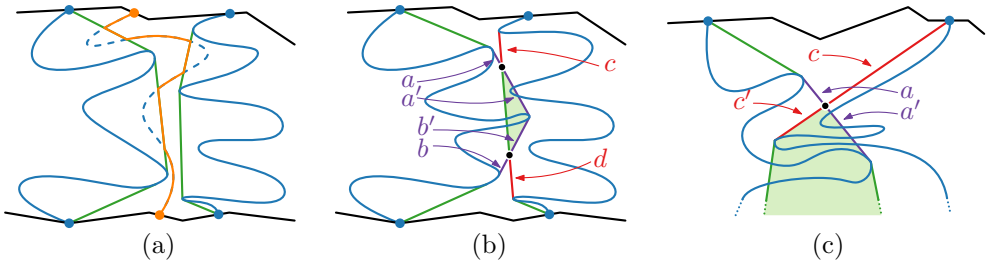


Figure 8 The three cases of Lemma 6. $D_i \cap D_j$ shaded.

bend. If l_j has one, a symmetric argument applies. We replace γ by a curve that passes through both x and x'. If γ does not pass through x already, then let $\bar{\varphi}$ and $\bar{\psi}$ be the lids of pockets of γ_i and γ_j, respectively, that intersect in x. It is also possible that $\bar{\varphi}$ is an edge of γ_i; the proof is then similar. Denote by a the arc $\bar{\varphi} \setminus D_j$, by a' the arc $\bar{\varphi} \cap D_j$ and by c the arc $\bar{\psi} \setminus D_i$. For the lids $\bar{\varphi}'$ and $\bar{\psi}'$ of r_i and l_j crossing in x', denote by b the arc $\bar{\varphi}' \setminus D_j$, by b' the arc $\bar{\varphi}' \cap D_j$ and by d the arc $\bar{\psi}' \setminus D_i$, see Figure 8 (b).
We claim that γ crosses either c, or both a and a'. If γ crosses c, we are done, so assume it does not. As $a \cup c$ connects γ_i and γ_j, γ must cross $a \cup c$ (and therefore a) an odd number of times to connect f and g. Lemma 5 implies that γ crosses $a \cup a'$ an even number of times. Hence, a' is crossed an odd number of times. We can thus find an arc of γ with endpoints on a and a', and since this arc does not cross c, it lies in the pocket of $\bar{\varphi}$. Replace this arc by the arc of $\bar{\varphi}$ between those endpoints, which is a shortest path in R that passes through x. We now have a curve with the same endpoints as γ that passes through c or x, and this curve is not longer than γ. We allow the resulting curve to cross γ_i and γ_j along $\bar{\varphi}$, however the resulting curve contains a subcurve of γ that connects a, a' or c to g. Analogously, we can replace this subcurve by a curve that crosses either x' or d. This yields a curve from f to g that passes through c or x, and then through x' or d. Since l_j has no bends, c and d lie on the same lid, which passes through x and x'. Since this lid is a shortest path, we can replace the subpath between c or x and x' or d by a shortest path in R that passes through both x and x'.
The portions of the curve before x and after x' can be shortcut using the techniques of case (a) such that they lie between l_j and r_i and not in $D_i \cap D_j$. This yields a curve γ' in the homotopy class of r_i' with the same endpoints as γ, and $\|\gamma'\| \leq \|\gamma\|$.

(c) In the final case, both r_i and l_j have bends on $\partial(D_i \cap D_j)$. As before, we shortcut γ such that it first passes through x and then through x'. Let the arcs a, a', c be as before, and let c' be the arc $\bar{\psi} \cap D_i$. As $a \cup a'$, as well as $c \cup c'$ are crossed an even number of times, but $a \cup c$ is crossed an odd number of times, we have that either both a and a', or both c and c' are crossed by γ. Replacing the arc between the crossings by the shortest path on the corresponding lid yields a path through x with length at most that of γ. Similarly, we can replace the remainder of the resulting path to also pass through x'. From here, we can use the same technique as in case (b) to obtain a curve γ' in the homotopy class of r_i' with the same endpoints as γ, and $\|\gamma'\| \leq \|\gamma\|$. ◀

3.1 Regular homotopies

Let $G(a, b)$ denote the geodesic in R between $f(a)$ and $g(b)$. Given a homotopy class σ, $G_\sigma(a, b)$ denotes the geodesic between $f(a)$ and $g(b)$ in σ. For $\gamma \in \Gamma$ denote its homotopy class by $\sigma(\gamma)$. If γ is a geodesic in R, we say that $\sigma(\gamma)$ is a *straight* homotopy class.

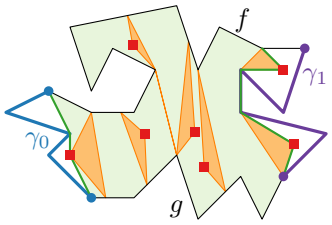


Figure 9 A decomposed homotopy. Subhomotopies S_i shaded green, and B_i shaded orange.

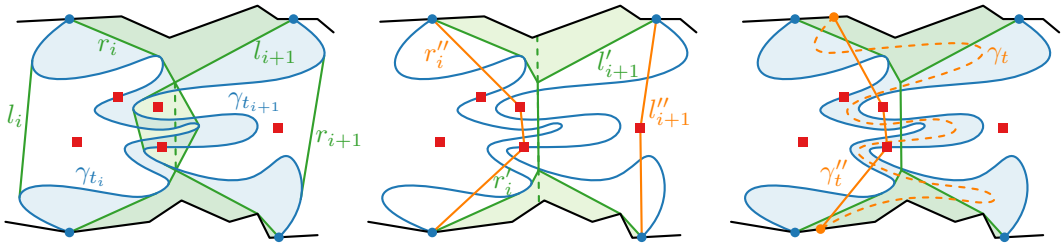


Figure 10 (a) The region between r_i and l_{i+1} obtained from the geodesic hulls of γ_{t_i} and $\gamma_{t_{i+1}}$. (b) The curves r'_i and l'_{i+1}, and the corresponding geodesics r''_i and l''_{i+1} in the same homotopy class. (c) For all $t_i \le t \le t_{i+1}$, the curve γ_t lies in the region between the curves γ_{t_i} and $\gamma_{t_{i+1}}$.

For an optimal homotopy, we may without loss of generality start by shortening γ_0 into the geodesic $G_{\sigma(\gamma_0)}(0,0)$ in its homotopy class, use an optimal homotopy h from $G_{\sigma(\gamma_0)}(0,0)$ to $G_{\sigma(\gamma_1)}(1,1)$, and end by lengthening the geodesic $G_{\sigma(\gamma_1)}(1,1)$ into γ_1. The resulting homotopy has cost $\max\{\|\gamma_0\|, \text{cost}(h), \|\gamma_1\|\}$, and the computational challenge is to efficiently find an optimal homotopy h.

We call a homotopy from $G_{\sigma(\gamma_0)}(0,0)$ to $G_{\sigma(\gamma_1)}(1,1)$ a *regular* homotopy of order m if it can be decomposed into a sequence of homotopies S_0, B_1, S_1, …, B_i, S_i, …, B_m, S_m, subject to the following constraints, see also Figure 9.

- $S_i(1) = B_{i+1}(0)$ for $0 \le i \le m-1$ and $B_i(1) = S_i(0)$ for $1 \le i \le m$, that is, the last leash of a homotopy matches the first leash of the next homotopy.
- For each homotopy B_i, the leashes all have the same endpoints on f and g, but the leashes can move over spikes. Moreover, the longest leash in B_i is either $B_i(0)$ or $B_i(1)$.
- For each homotopy S_i, all leashes are geodesics in the same homotopy class σ_i, but the endpoints of leashes can move along f and g.
- The respective homotopy classes of leashes in S_0 and S_m are $\sigma_0 = \sigma(\gamma_0)$ and $\sigma_m = \sigma(\gamma_1)$.

In Lemma 7, we show that there exists a minimum-cost homotopy h between $G_{\sigma(\gamma_0)}(0,0)$ and $G_{\sigma(\gamma_1)}(1,1)$ that is a regular homotopy of order at most k. In Lemma 8, we show that each σ_i (except possibly σ_0 and σ_m) can be assumed to be a straight homotopy class. For each homotopy B_i, some leash can be assumed be the geodesic in R between its endpoints.

For a homotopy h, denote by $\alpha_h(t) \in [0,1]$, the position of the start of leash $h(t)$ in the parameter space of f (such that $f(\alpha_h(t)) = h(t,0)$). Symmetrically, denote by $\beta_h(t) \in [0,1]$ the position of the end of that leash in the parameter space of g (such that $g(\beta_h(t)) = h(t,1)$). If h is monotone, then α_h and $\beta_h \colon [0,1] \to [0,1]$ are continuous nondecreasing surjections.

▶ **Lemma 7.** *Let h be a monotone homotopy from $G_{\sigma(\gamma_0)}(0,0)$ to $G_{\sigma(\gamma_1)}(1,1)$ of cost less than L, then there exists a regular homotopy of cost at most L.*

Proof. By monotonicity, as t increases, the number of spikes in the swept region of γ_t cannot decrease. Let $\{t_1, \ldots, t_k\}$ be the minimum value t_i for which the swept region of γ_{t_i} contains at least i spikes. For each leash γ_{t_i} of h, let $D_i = \mathrm{gh}_R(\gamma_{t_i})$ be its geodesic hull, and let l_i and $r_i \in \Gamma$ respectively be the backwards-convex and forwards-convex curves on the boundary of D_i connecting $\gamma_{t_i}(0)$ with $\gamma_{t_i}(1)$. For $1 \leq i \leq k$, define r'_i and l'_{i+1} respectively to be the curves obtained by replacing the arcs of r_i and respectively l_{i+1} inside $D_i \cap D_{i+1}$ by the geodesic between the crossings of l_{i+1} and r_i (if any), dashed in Figure 10(a). Then l'_i lies in the swept region of r'_i, and r'_i lies in that of l'_{i+1}. Let r''_i and l''_i be the geodesics with the same endpoints and homotopy class as r'_i and l'_i, respectively (see Figure 10(b)).

Lemma 3 gives us a homotopy B_i from l''_i to r''_i whose leashes have length at most $\max\{\|l''_i\|, \|r''_i\|\}$. Including the cost of spikes, the cost of B_i is at most $\max\{\|l''_i\|, \|r''_i\|\}+1+\varepsilon$ (for any $\varepsilon > 0$) by perturbing leashes to each encounter at most one spike. Because l''_i is a geodesic in the same homotopy class as l'_i, we have $\|l''_i\| \leq \|l'_i\|$. Moreover, l'_i is a copy of l_i with a subpath replaced by a shortest path, we have $\|l'_i\| \leq \|l_i\|$. Finally, because l_i lies on arcs of the convex hull of γ_{t_i}, we have $\|l_i\| \leq \|\gamma_{t_i}\|$. Therefore, we have $\|l''_i\| \leq \|\gamma_{t_i}\|$, and by symmetry , $\|r''_i\| \leq \|\gamma_{t_i}\|$. Since γ_{t_i} encounters a spike, and the cost of h is less than L, we have $\|\gamma_{t_i}\| < L - 1$. Hence, $\mathrm{cost}(B_i) \leq \max\{\|l''_i\|, \|r''_i\|\} + 1 + \varepsilon \leq \|\gamma_{t_i}\| + 1 + \varepsilon \leq L$.

Define $r''_0 = G_{\sigma(\gamma_0)}(0,0)$ and $l''_{k+1} = G_{\sigma(\gamma_1)}(1,1)$. Moreover, let $t_0 = 0$ and $t_{k+1} = 1$. It remains to construct homotopies S_i between r''_i and l''_{i+1} for $0 \leq i \leq k$. Since there are no spikes interior to the region between r'_i and l'_{i+1}, they lie in the same homotopy class, which we denote by σ_i. To construct a homotopy from r''_i to l''_{i+1}, we consider the curves γ_t with $t_i \leq t \leq t_{i+1}$, and replace them by the geodesic $\gamma''_t = G_{\sigma_i}(\alpha_h(t), \beta_h(t))$ in homotopy class σ_i (see Figure 10(c)). These geodesics move continuously with t, so it remains to show that $\|\gamma''_t\| \leq \|\gamma_t\|$. This is not immediate since γ''_t may lie in a different homotopy class than γ_t. Instead, we use Lemma 6, which tells us that there is a curve with the same endpoints in the homotopy class of r'_i with length at most that of γ_t. Because γ''_t is a geodesic in the same homotopy class, its length is also at most that of γ_t. ◀

The straight homotopy classes in R can be enumerated by taking the geodesic between any two spikes in R, and extending it to the two points on the boundary of R. It is at these points where the geodesic hits the boundary of R that the homotopy class of the geodesic between points on the boundary of R changes: one can slide these points clockwise or counter-clockwise such the geodesic between them ends up in a different straight homotopy class. There are $O(k^2)$ straight homotopy classes.

▶ **Lemma 8.** *For $1 \leq i \leq k-1$, we can assume σ_i to be a straight homotopy class without increasing its cost.*

Proof. Curve r_i is forwards-convex and l_{i+1} is backwards-convex, and the endpoints of r_i on f and g are not ahead of those of l_{i+1}. If r_i and l_{i+1} are disjoint, then we can find a geodesic in R separating r_i and l_{i+1}, and hence r''_i and l''_{i+1}.

If they are not disjoint, then the shortest path between their points of intersection lies on a geodesic between f and g, separating r'_i and l'_{i+1}, and hence r''_i and l''_{i+1}. ◀

3.2 Computation

A tool that is commonly used to compute the Fréchet distance is the *free space diagram* [1]. This tool captures between which points of f and g the geodesic is sufficiently short to be used as a leash in a homotopy of a given cost L. Formally, the free space diagram is defined as $\mathcal{F}(L) = \{(a,b) \in [0,1] \times [0,1] \mid \|G(a,b)\| \leq L\}$. More generally, for a given homotopy class σ, we define $\mathcal{F}_\sigma(L) = \{(a,b) \in [0,1] \times [0,1] \mid \|G_\sigma(a,b)\| \leq L\}$ to capture the geodesics in σ of length at most L.

Let Σ be the set of homotopy classes consisting of $\sigma(\gamma_0)$, $\sigma(\gamma_1)$, and all straight homotopy classes. There are $2 + O(k^2) = O(k^2)$ such homotopy classes, assuming $k \geq 1$. Let h be a regular homotopy of cost at most L, and let t_i and t'_i be the values of t in h at which the constituent homotopy S_i starts and stops, respectively. For all $t \in [t_i, t'_i]$, we have $\|h(t)\| \leq \text{cost}(S_i) \leq L$, so $(\alpha_h(t), \beta_h(t)) \in \mathcal{F}_{\sigma_i}(L)$, where σ_i is the homotopy class of the leashes in S_i. For $t \in [t'_{i-1}, t_i]$, the leashes $h(t)$ are part of homotopy B_i, and we even have $\|h(t)\| + 1 \leq L$, such that accounting for the spikes the leash passes over, we have $\text{cost}(B_i) \leq \max\{\|h(t'_{i-1})\| + 1, \|h(t_i)\| + 1\} \leq L$ by the construction of Lemma 7. Recall that the endpoints of leashes do not move throughout any homotopy B_i, so $\alpha_h(t) = \alpha_h(t_i)$ and $\beta_h(t) = \beta_h(t_i)$ for all $t \in [t'_{i-1}, t_i]$. Additionally, as the construction of Lemma 7 preserves monotonicity, we can assume α_h and β_h to both be continuous nondecreasing surjections. By Lemma 8, we can also assume that each σ_i lies in Σ. For the sake of presentation, since α_h and β_h are constant in the intervals $[t'_{i-1}, t_i]$, we assume from now on that $t'_{i-1} = t_i$, and prove that any homotopy with the structure imposed by Lemma 9 can be turned into a regular homotopy of cost L.

▶ **Lemma 9.** *We can construct a regular homotopy of cost at most L if we can find appropriate α_h, β_h, t_i and t'_i and values of $\sigma_i \in \Sigma$, with the following conditions. Let α and $\beta\colon [0,1] \to [0,1]$ be continuous nondecreasing surjections. Let $\sigma_0 = \sigma(\gamma_0)$, $\sigma_m = \sigma(\gamma_1)$, and $\sigma_i \in \Sigma$ for $i \in \{1, \ldots, m-1\}$. Let $0 = t_0 \leq t_1 \leq \cdots \leq t_{m+1} = 1$. Then, if $(\alpha(t), \beta(t)) \in \mathcal{F}_{\sigma_i}(L)$ for each $t \in [t_i, t_{i+1}]$, and additionally $(\alpha(t), \beta(t)) \in \mathcal{F}_{\sigma_i}(L-1) \cap \mathcal{F}_{\sigma_{i+1}}(L-1)$ for each t_i with $i \in \{1, \ldots, m\}$, this corresponds to a regular homotopy of cost at most L.*

Proof. We use geodesics of σ_i for $t \in [t_i, t_{i+1}]$, and they move continuously. By Lemma 3, we can find a homotopy B_i of cost at most L if the geodesics of σ_i and σ_{i+1} based at $\alpha(t)$ and $\beta(t)$ both have length at most $L-1$. Furthermore, since $(\alpha(t), \beta(t)) \in \mathcal{F}_{\sigma_i}(L)$ for each $t \in [t_i, t_{i+1}]$, we can find a homotopy S_i of cost at most L between σ_i and σ_{i+1}. ◀

Computing free space diagrams. To compute the free space diagram in our setting, we subdivide the edges of f and g in such a way that for each pair of (subdivided) edges, the length of the geodesic can be described as a quadratic function in two parameters a and b. This subdivision is based on the lines through any pair of spikes and vertices of ∂R, and finding their projection onto f or g, if any. In total, this yields subdivided curves f' and g' of $O((n+k)^2)$ vertices. Using a standard rotating sweep around every spike and vertex, we can compute the projections in $O((n+k)^2 \log(n+k))$ time and sort them along every edge of f and g in the same time, giving the subdivided curves f' and g'.

Now, given any straight homotopy class, or the homotopy class of γ_0 or γ_1, we compute the quadratic function for each pair of edges $(e_{f'}, e_{g'})$ of f' and g'. To this end, we take a straight homotopy class σ and determine the induced partition of K into K_1 and K_2. Then we compute the convex hulls $\text{gh}_R(K_1)$ and $\text{gh}_R(K_2)$ of K_1 and K_2 relative to the domain R. Using the common inner tangents of $\text{gh}_R(K_1)$ and $\text{gh}_R(K_2)$ in R we can find all pairs of edges $(e_{f'}, e_{g'})$ of f' and g' for which the geodesic in σ is straight, and determine the corresponding quadratic functions (which are ellipses). For other pairs of edges of f' and g', the geodesic contains vertices of $\text{gh}_R(K_1)$, $\text{gh}_R(K_2)$, and R itself, and their lengths are determined by a hyperbolic part in a, a hyperbolic part in b, and a constant part (between the first and last vertices not on f' and g'). We fix an edge $e_{g'}$ of g' and traverse all edges of f' sequentially, updating the three parts of the quadratic function when needed. Updates of the constant part happen only at the ends of the geodesic, and amortized we can do all updates in time linear in the number of parts of f', that is, $O((n+k)^2)$.

Hence, we can compute all quadratic functions for all straight homotopy classes in time $O(k^2)$ (for the straight homotopy classes) times $O((n+k)^2)$ (for the number of segments of g') times $O((n+k)^2)$ (for the number of segments of f'). In total, this is $O((n+k)^4k^2) = O(n^4k^2+k^6)$ time. The $O(n^4+k^4)$ cells of the free space diagram each have $O(k^2)$ quadratic functions, at most one for each of the homotopy classes.

Decision algorithm. For a parameter L, we define the *reachable free space* as the set of coordinates $(\sigma, a, b) \in \Sigma \times [0,1] \times [0,1]$, such that there exist continuous nondecreasing surjections $\alpha\colon [0,1] \to [0,a]$ and $\beta\colon [0,1] \to [0,b]$, a value m, values $0 = t_0 \leq t_1 \leq \cdots \leq t_{m+1} = 1$, and homotopy classes $\sigma_i \in \Sigma$ with $\sigma_0 = \sigma(\gamma_0)$ and $\sigma_m = \sigma$, such that for each $t \in [t_i, t_{i+1}]$, we have $(\alpha(t), \beta(t)) \in \mathcal{F}_{\sigma_i}(L)$ and for each $i \in \{1, \ldots, m\}$, we have $(\alpha(t_i), \beta(t_i)) \in \mathcal{F}_{\sigma_i}(L-1) \cap \mathcal{F}_{\sigma_{i+1}}(L-1)$. The reachable free space corresponds to the classes σ and points $f(a)$ and $g(b)$ that have a monotone regular homotopy from $G_{\sigma_0}(0,0)$ to $G_\sigma(f(a), g(b))$ of cost at most L. Deciding whether a regular homotopy of at most a certain cost L exists is then equivalent to testing whether $(\sigma(\gamma_1), 1, 1)$ lies in the reachable free space for parameter L. We can compute the reachable free space using dynamic programming. In contrast algorithms for most variants of the Fréchet distance, which need only information about the free space on the boundary of cells, we also need information about their interiors. In our dynamic program, we compute the reachable free space on the boundary of each cell.

The restriction of the free space to any cell and homotopy class is convex. Therefore, if a point (σ, a, b) lies in the reachable free space, then for all $a' \geq a$ and $b' \geq b$, in the same cell as (a, b), if $(a', b') \in \mathcal{F}_\sigma(e)(L)$, then (σ, a', b') also lies in the reachable free space. Moreover, for σ and $\sigma' \in \Sigma$, if $(a, b) \in \mathcal{F}_\sigma(L-1)$ and $(a, b) \in \mathcal{F}_{\sigma'}(L-1)$, then (σ, a, b) lies in the reachable free space if and only if (σ', a, b) lies in the reachable free space. Our dynamic program starts as follows: for a homotopy of cost at most L to exist, check whether $(0,0) \in \mathcal{F}_{\sigma_0}(L)$. If so, $(\sigma_0, 0, 0)$ lies in the reachable free space, and otherwise the reachable free space is empty. Now we propagate the reachable free space on a cell-by-cell basis, maintaining for each cell $[a, a'] \times [b, b']$ and for each homotopy class σ, two pieces of information. First, the minimum $a^* \in [a, a']$ for which (σ, a^*, b) lies in the reachable free space (if any); and second, the minimum $b^* \in [b, b']$ for which (σ, a, b^*) lies in the reachable free space (if any). The first piece of information can be propagated to a neighboring cell $[a, a'] \times [b', b'']$, and the second piece can be propagated to a neighboring cell $[a', a''] \times [b, b']$.

To propagate this information, we use a horizontal sweep line that maintains the reachable free space intersecting the sweep line for the cell $[a, a'] \times [b, b']$ in each of the $O(k^2)$ relevant homotopy classes, based on the coordinates (σ, a, b^*) and (σ, a^*, b) in each of those homotopy classes. Naively, we can propagate this information in $O(k^6)$ time per cell, using the coordinates of the $O(k^4)$ intersections of the boundary of free space cells from different homotopy classes as events for the sweep line.

After propagating the information through all $O(n^4 + k^4)$ cells of the free space in $O(n^4k^6 + k^{10})$ time, we can return whether $(\sigma(\gamma_1), 1, 1)$ lies in the reachable free space to decide whether there exists a homotopy of cost at most L.

Exact computation. The candidate values for the minimum-cost regular homotopy depend on the values of L where the a- or b-coordinates of different intersections align. There are $O(((n+k)^2k^4)^2)$ intersections that can align in this way, which yields $O(n^4k^8 + k^{12})$ critical values, which we can enumerate in $O(1)$ time per value. We perform a binary search over these critical values, using linear-time median finding and running the decision procedure $O(\log nk)$ times to find the minimum cost of a regular homotopy. Doing so, we compute the homotopy height in $O(n^4k^6 \log n + n^4k^8 + k^{12})$ time.

4 Conclusion

We have shown that in a spiked plane with polygonal boundary, we can compute the homotopy height between two curves on the boundary in polynomial time for various cases. In particular, this holds if all spikes have the same (unit) weight, or if the two curves together form the entire boundary of the domain. We also provide a 2-approximation algorithm for the general case. We have also shown that intermediate leashes may require many inflection points for an optimal homotopy, even if the polygonal domain is convex. This complexity of the leash has been preventing us from developing a polynomial-time algorithm, and thus it remains open whether the general case can be solved optimally in polynomial time.

Future work. Various other settings can also be studied. The case where f and g are not on the boundary of the polygonal domain is a natural first step. However, the monotonicity [5] that supports our results is not known to hold in this case, which is likely a premise for efficient optimal algorithms. Our approximation algorithm (Lemma 4) extends to deal with the case that γ_0 and γ_1 are still on the boundary, since the algorithm upon which it is based [13] does not require f and g to lie on the boundary. If the initial leashes γ_0 and γ_1 are no longer specified, we readily get a 2-approximation algorithm by using the algorithm by Chambers *et al.* [6] to solve the decision variant combined with an appropriate search.

References

1. Helmut Alt and Michael Godau. Computing the Fréchet distance between two polygonal curves. *International Journal on Computational Geometry and Application*, 5(1–2):75–91, 1995.
2. Patrizio Angelini, Giordano Da Lozzo, Giuseppe Di Battista, Fabrizio Frati, Maurizio Patrignani, and Vincenzo Roselli. Morphing planar graph drawings optimally. In *Automata, Languages, and Programming (ICALP)*, LNCS 8572, pages 126–137, 2014.
3. Patrizio Angelini, Fabrizio Frati, Maurizio Patrignani, and Vincenzo Roselli. Morphing planar graph drawings efficiently. In *Graph Drawing*, volume 8242 of *LNCS 8242*, pages 49–60, 2013.
4. Graham R. Brightwell and Peter Winkler. Submodular percolation. *SIAM Journal on Discrete Mathematics*, 23(3):1149–1178, 2009.
5. Erin W. Chambers, Gregory R. Chambers, Arnaud de Mesmay, Tim Ophelders, and Regina Rotman. Monotone contractions of the boundary of the disc. Computing Research Repository (arXiv):1704.06175, 2017.
6. Erin W. Chambers, Éric Colin de Verdière, Jeff Erickson, Sylvain Lazard, Francis Lazarus, and Shripad Thite. Homotopic Fréchet distance between curves or, walking your dog in the woods in polynomial time. *Computational Geometry: Theory and Applications*, 43(3):295–311, 2010.
7. Erin W. Chambers and David Letscher. On the height of a homotopy. In *Proceedings of the 21st Canadian Conference on Computational Geometry*, pages 103–106, 2009.
8. Erin W. Chambers and Mikael Vejdemo-Johansson. Computing minimum area homologies. *Computer Graphics Forum*, 34(6):13–21, 2015.
9. Erin W. Chambers and Yusu Wang. Measuring similarity between curves on 2-manifolds via homotopy area. In *Proceedings of the 29th Annual Symposium on Computational Geometry*, pages 425–434, 2013.
10. Gregory R. Chambers. *Optimal Homotopies of Curves on Surfaces*. PhD thesis, University of Toronto, Canada, 2014.

11 Gregory R. Chambers and Yevgeny Liokumovich. Converting homotopies to isotopies and dividing homotopies in half in an effective way. *Geometric and Functional Analysis*, 24(4):1080–1100, 2014.

12 Gregory R. Chambers and Regina Rotman. Monotone homotopies and contracting discs on Riemannian surfaces. *Journal of Topology and Analysis*, 2016.

13 Atlas F. Cook and Carola Wenk. Geodesic Fréchet distance inside a simple polygon. *ACM Transactions on Algorithms*, 7(1):Art. 9, 2009.

14 Tamal K. Dey, Anil N. Hirani, and Bala Krishnamoorthy. Optimal homologous cycles, total unimodularity, and linear programming. *SIAM Journal on Computing*, 40(4):1026–1044, 2011.

15 Alon Efrat, Quanfu Fan, and Suresh Venkatasubramanian. Curve matching, time warping, and light fields: New algorithms for computing similarity between curves. *Journal of Mathematical Imaging and Vision*, 27(3):203–216, 2007. doi:10.1007/s10851-006-0647-0.

16 Alon Efrat, Leonidas J. Guibas, Sariel Har-Peled, Joseph S. B. Mitchell, and T. M. Murali. New similarity measures between polylines with applications to morphing and polygon sweeping. *Discrete & Computational Geometry*, 28(4):535–569, 2002.

17 Brittany Fasy, Selcuk Karakoç, and Carola Wenk. On minimum area homotopies. In *Computational Geometry: Young Researchers Forum*, pages 49–50, 2016.

18 Sariel Har-Peled, Amir Nayyeri, Mohammad Salavatipour, and Anastasios Sidiropoulos. How to walk your dog in the mountains with no magic leash. In *Proceedings of the 28th Annual Symposium on Computational Geometry*, pages 121–130, 2012.

19 Zipei Nie. On the minimum area of null homotopies of curves traced twice. Computing Research Repository (arXiv):1412.0101, 2014.

20 Godfried Toussaint. An optimal algorithm for computing the relative convex hull of a set of points in a polygon. School of Computer Science, McGill University, 1986.

Online Submodular Maximization Problem with Vector Packing Constraint*†

T.-H. Hubert Chan[1], Shaofeng H.-C. Jiang[2], Zhihao Gavin Tang[3], and Xiaowei Wu[4]

1 **Department of Computer Sicence, The University of Hong Kong, Hong Kong**
`hubert@cs.hku.hk`
2 **Department of Computer Sicence, The University of Hong Kong, Hong Kong**
`sfjiang@cs.hku.hk`
3 **Department of Computer Sicence, The University of Hong Kong, Hong Kong**
`zhtang@cs.hku.hk`
4 **Department of Computer Sicence, The University of Hong Kong, Hong Kong**
`xwwu@cs.hku.hk`

Abstract

We consider the online vector packing problem in which we have a d dimensional knapsack and items u with weight vectors $\mathbf{w}_u \in \mathbb{R}_+^d$ arrive online in an arbitrary order. Upon the arrival of an item, the algorithm must decide immediately whether to discard or accept the item into the knapsack. When item u is accepted, $\mathbf{w}_u(i)$ units of capacity on dimension i will be taken up, for each $i \in [d]$. To satisfy the knapsack constraint, an accepted item can be later disposed of with no cost, but discarded or disposed of items cannot be recovered. The objective is to maximize the utility of the accepted items S at the end of the algorithm, which is given by $f(S)$ for some non-negative monotone submodular function f.

For any small constant $\epsilon > 0$, we consider the special case that the weight of an item on every dimension is at most a $(1-\epsilon)$ fraction of the total capacity, and give a polynomial-time deterministic $O(\frac{k}{\epsilon^2})$-competitive algorithm for the problem, where k is the (column) sparsity of the weight vectors. We also show several (almost) tight hardness results even when the algorithm is computationally unbounded. We first show that under the ϵ-slack assumption, no deterministic algorithm can obtain any $o(k)$ competitive ratio, and no randomized algorithm can obtain any $o(\frac{k}{\log k})$ competitive ratio. We then show that for the general case (when $\epsilon = 0$), no randomized algorithm can obtain any $o(k)$ competitive ratio.

In contrast to the $(1+\delta)$ competitive ratio achieved in Kesselheim et al. (STOC 2014) for the problem with random arrival order of items and under large capacity assumption, we show that in the arbitrary arrival order case, even when $\|\mathbf{w}_u\|_\infty$ is arbitrarily small for all items u, it is impossible to achieve any $o(\frac{\log k}{\log \log k})$ competitive ratio.

1998 ACM Subject Classification F.1.2 Modes of Computation, G.1.6 Optimization

Keywords and phrases Submodular Maximization, Free-disposal, Vector Packing

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.24

1 Introduction

Online Vector Packing Problem. We consider the following online submodular maximization problem with vector packing constraint. Suppose we have a d dimensional knapsack,

* The full version of this paper can be found at `https://arxiv.org/abs/1706.06922`.
† This work was partially supported by the Hong Kong RGC under the grant 17202715.

and items arrive online in an arbitrary order. Each item $u \in \Omega$ has a *weight vector* $\mathbf{w}_u \in \mathbb{R}_+^d$, i.e., when item $u \in \Omega$ is accepted, for each $i \in [d]$, item u will take up $\mathbf{w}_u(i)$ units of capacity on every dimension i of the knapsack. By rescaling the weight vectors, we can assume that each of the d dimensions has capacity 1. Hence we can assume w.l.o.g. that $\mathbf{w}_u \in [0,1]^d$ for all $u \in \Omega$. The *(column) sparsity* [6, 26] is defined as the minimum number k such that every weight vector $\mathbf{w}_u$ has at most k non-zero coordinates. The objective is to pack a subset of items with the maximum utility into the knapsack, where the utility of a set S of items is given by a non-negative monotone submodular function $f : 2^\Omega \to \mathbb{R}_+$.

The vector packing constraint requires that the accepted items can take up a total amount of at most 1 capacity on each of the d dimensions of the knapsack. However, as items come in an arbitrary order, it can be easily shown that the competitive ratio is arbitrarily bad, if the decision of acceptance of each item is decided online and cannot be revoked later. In the literature, when the arrival order is arbitrary, the *free disposal* feature [19] is considered, namely, an accepted item can be disposed of when later items arrive. On the other hand, we cannot recover items that are discarded or disposed of earlier.

We can also interpret the problem as solving the following program online, where variables pertaining to u arrive at step $u \in \Omega$. We assume that the algorithm does not know the number of items in the sequence. The variable $x_u \in \{0,1\}$ indicates whether item u is accepted. During the step u, the algorithm decides to set x_u to 0 or 1, and may decrease $x_{u'}$ from 1 to 0 for some $u' < u$ in order to satisfy the vector packing constraints.

$$\begin{aligned} \max \quad & f(\{u \in \Omega : x_u = 1\}) \\ \text{s.t.} \quad & \textstyle\sum_{u \in \Omega} \mathbf{w}_u(i) \cdot x_u \le 1, && \forall i \in [d] \\ & x_u \in \{0,1\}, && \forall u \in \Omega. \end{aligned}$$

In some existing works [8, 18, 27, 26], the items are decided by the adversary, who sets the value (the utility of a set of items is the summation of their values) and the weight vector of each item, but the items arrive in a uniformly random order. This problem is sometimes referred to as Online Packing LPs with random arrival order, and each choice is irrevocable. To emphasize our setting, we refer to our problem as Online Vector Packing Problem (with submodular objective and free disposal).

Competitive Ratio. After all items have arrived, suppose $S \subset \Omega$ is the set of items currently accepted (excluding those that are disposed of) by the algorithm. The objective is $\mathsf{ALG} := f(S)$. Note that to guarantee feasibility, we have $\sum_{u \in S} \mathbf{w}_u \le \mathbf{1}$, where $\mathbf{1}$ denotes the d dimensional all-one vector. The competitive ratio is defined as the ratio between the optimal objective OPT that is achievable by an offline algorithm and the (expected) objective of the algorithm: $\mathsf{r} := \frac{\mathsf{OPT}}{\mathsf{E[ALG]}} \ge 1$.

1.1 Our Results and Techniques

We first consider the Online Vector Packing Problem *with slack*, i.e., there is a constant $\epsilon > 0$ such that for all $u \in \Omega$, we have $\mathbf{w}_u \in [0, 1-\epsilon]^d$, and propose a deterministic $O(\frac{k}{\epsilon^2})$-competitive algorithm, where k is the sparsity of weight vectors.

▶ **Theorem 1.** *For the Online Vector Packing Problem with ϵ slack, there is a (polynomial-time) deterministic $O(\frac{k}{\epsilon^2})$-competitive algorithm for the Online Vector Packing Problem.*

Observe that by scaling weight vectors, Theorem 1 implies a bi-criteria $(1+\epsilon, \frac{k}{\epsilon^2})$-competitive algorithm for general weight vectors, i.e., by relaxing the capacity constraint by an ϵ fraction, we can obtain a solution that is $O(\frac{k}{\epsilon^2})$-competitive compared to the optimal solution (with the augmented capacity).

We show that our competitive ratio is optimal (up to a constant factor) for deterministic algorithms, and almost optimal (up to a logarithmic factor) for any (randomized) algorithms. Moreover, all our following hardness results (Theorem 2, 3 and 4) hold for algorithms with **unbounded computational power**.

▶ **Theorem 2** (Hardness with Slack). *For the Online Vector Packing Problem with slack $\epsilon \in (0, \frac{1}{2})$, any deterministic algorithm has a competitive ratio $\Omega(k)$, even when the utility function is linear and all items have the same value, i.e., $f(S) := |S|$; for randomized algorithms, the lower bound is $\Omega(\frac{k}{\log k})$.*

We then consider the hardness of the Online Vector Packing Problem (without slack) and show that no (randomized) algorithm can achieve any $o(k)$-competitive ratio.

▶ **Theorem 3** (Hardness without Slack). *Any (randomized) algorithm for the Online Vector Packing Problem has a competitive ratio $\Omega(k)$, even when $f(S) := |S|$.*

As shown by [26], for the Online Vector Packing Problem with random arrival order, if we have $\|\mathbf{w}_u\|_\infty = O(\frac{\epsilon^2}{\log k})$ for all items $u \in \Omega$, then a $(1+\epsilon)$ competitive ratio can be obtained. Hence, a natural question is whether better ratio can be achieved under this "small weight" assumption. For example, if $\max_{u\in\Omega}\{\|\mathbf{w}_u\|_\infty\}$ is arbitrarily small, is it possible to achieve a $(1+\epsilon)$ competitive ratio like existing works [18, 16, 27, 26]? Unfortunately, we show that even when all weights are arbitrarily small, it is still not possible to achieve any constant competitive ratio.

▶ **Theorem 4** (Hardness under Small Weight Assumption). *There does not exist any (randomized) algorithm with an $o(\frac{\log k}{\log\log k})$ competitive ratio for the Online Vector Packing Problem, even when $\max_{u\in\Omega}\{\|\mathbf{w}_u\|_\infty\}$ is arbitrarily small and $f(S) := |S|$.*

Our hardness result implies that even with free disposal, the problem with arbitrary arrival order is strictly harder than its counter part when the arrival order is random. For space reason, we defer the proof the Theorem 3 and 4 to the full version of the paper [10].

Our Techniques. To handle submodular functions, we use the standard technique by considering marginal cost of an item, thereby essentially reducing to linear objective functions. However, observe that the hardness results in Theorems 2, 3 and 4 hold even for linear objective function where every item has the same value. The main difficulty of the problem comes from the weight vectors of items, i.e., when items conflict with one another due to multiple dimensions, it is difficult to decide which items to accept. Indeed, even the offline version of the problem has an $\Omega(\frac{k}{\log k})$ NP-hardness of approximation result [24, 28].

For the case when $d = 1$, i.e., $\mathbf{w}_u \in [0, 1]$, it is very natural to compare items based on their *densities* [23], i.e., the value per unit of weight, and accept the items with maximum densities. A naive solution is to use the maximum weight $\|w_u\|_\infty$ to reduce the problem to the 1-dimensional case, but this can lead to $\Omega(d)$-competitive ratio, even though each weight vector has sparsity $k \ll d$. To overcome this difficulty, we define for each item a density on **each** of the d dimensions, and make items comparable on any particular dimension.

Even though our algorithm is deterministic, we borrow techniques from randomized algorithms for a variant of the problem with matroid constraints [7, 9]. Our algorithm maintains a fractional solution, which is rounded at every step to achieve an integer solution. When a new item arrives, we try to accept the item by **continuously** increasing its accepted fraction (up to 1), while for each of its k non-zero dimensions, we decrease the fraction of the currently least dense accepted item, as long as the rate of increase in value due to the new item is at least some factor times the rate of loss due to disposing of items fractionally.

The rounding is simple after every step. If the new item is accepted with a fraction larger than some threshold α, then the new item will be accepted completely in the integer solution; at the same time, if the fraction of some item drops below some threshold β, then the corresponding item will be disposed of completely in the integer solution. The ϵ slack assumption is used to bound the loss of utility due to rounding. The high level intuition of why the competitive ratio depends on the sparsity k (as opposed to the total number d of dimensions) is that when a new item is fractionally increased, at most k dimensions can cause other items to be fractionally disposed of.

Then, we apply a standard argument to compare the value of items that are eventually accepted (the utility of our algorithm) with the value of items that are **ever** accepted (but maybe disposed of later). The value of the latter is in turn compared with that of an optimal solution to give the competitive ratio.

1.2 Related Work

The Online Vector Packing Problem (with free disposal) is general enough to subsume many well-known online problems. For instance, the special case $d = 1$ becomes the Online Knapsack Problem [23]. The offline version of the problem captures the k-Hypergraph b-Matching Problem (with sparsity k and $\mathbf{w}_u \in \{0, \frac{1}{b}\}^d$, where d is the number of vertices), for which an $\Omega(\frac{k}{b \log k})$ NP-hardness of approximation is known [24, 28], for any $b \leq \frac{k}{\log k}$. In contrast, our hardness results are due to the online nature of the problem and hold even if the algorithms have unbounded computational power.

Free Disposal. The free disposal setting was first proposed by Feldman et al. [19] for the online edge-weighted bipartite matching problem with arbitrary arrival order, in which the decision whether an online node is matched to an offline node must be made when the online node arrives. However, an offline node can dispose of its currently matched node, if the new online node is more beneficial. They showed that the competitive ratio approaches $1 - \frac{1}{e}$ when the number of online nodes each offline node can accept approaches infinity. It can be shown that in many online (edge-weighted) problems with arbitrary arrival order, no algorithm can achieve any bounded competitive ratio without the free disposal assumption. Hence, this setting has been adopted by many other works [13, 5, 17, 14, 22, 20, 11, 23, 7].

Online Generalized Assignment Problem (OGAP). Feldman et al. [19] also considered a more general online biparte matching problem, where each edge e has both a value v_e and a weight w_e, and each offline node has a capacity constraint on the sum of weights of matched edges (assume without loss of generality that all capacities are 1). It can be easily shown that the problem is a special case of the Online Vector Packing Problem with d equal to the total number of nodes, and sparsity $k = 2$: every item represents an edge e, and has value v_e, weight 1 on the dimension corresponding to the online endpoint, and weight w_e on the dimension corresponding to the offline endpoint.

For the problem when each edge has arbitrary weight and each offline node has capacity 1, it is well-known that the greedy algorithm that assigns each online node to the offline node with maximum marginal increase in the objective is 2-competitive, while no algorithm is known to have a competitive ratio strictly smaller than 2. However, several special cases of the problem were analyzed and better competitive ratios have been achieved [2, 15, 11, 1].

Apart from vector packing constraints, the online submodular maximization problem with free disposal has been studied under matroid constraints [7, 9]. In particular, the uniform and the partition matroids can be thought of special cases of vector packing constraints,

where each item's weight vector has sparsity one and the same value for non-zero coordinate. However, using special properties of partition matroids, the exact optimal competitive ratio can be derived in [9], from which we also borrow relevant techniques to design our online algorithm.

Other Online Models. Kesselheim et al. [26] considered a variant of the problem when items (which they called requests) arrive in random order and have small weights compared to the total capacity; this is also known as the *secretary setting*, and free disposal is not allowed. They considered a more general setting in which an item can be accepted with more than one option, i.e., each item has different utilities and different weight vectors for different options. For every $\delta \in (0, \frac{1}{2})$, for the case when every weight vector is in $[0, \delta]^d$, they proposed an $O(k^{\frac{\delta}{1-\delta}})$-competitive algorithm, and a $(1+\epsilon)$-competitive algorithm when $\delta = O(\frac{\epsilon^2}{\log k})$, for $\epsilon \in (0, 1)$. In the random arrival order framework, many works assumed that the weights of items are much smaller than the total capacity [18, 16, 27, 26]. In comparison, our algorithm just needs the weaker ϵ slack assumption that no weight is more than $1 - \epsilon$ fraction of the total capacity.

The Online Vector Bin Packing problem [4, 3] is similar to the problem we consider in this paper. In the problem, items (with weight $\mathbf{w}_u \in [0, 1]^d$) arrive online in an arbitrary order and the objective is to pack all items into a minimum number of knapsacks, each with capacity $\mathbf{1}$. The current best competitive ratio for the problem is $O(d)$ [21] while the best hardness result is $\Omega(d^{1-\epsilon})$ [4], for any constant $\epsilon > 0$.

Future Work. We believe that it is an interesting open problem to see whether an $O(k)$-competitive ratio can be achieved for general instances, i.e., $\mathbf{w}_u \in [0, 1]^d$. However, at least we know that it is impossible to do so using deterministic algorithms (see Lemma 5).

Actually, it is interesting to observe that similar slack assumptions on the weight vectors of items have been made by several other literatures [12, 4, 26]. For example, for the Online Packing LPs problem (with random arrival order) [26], the competitive ratio $O(k^{\frac{\delta}{1-\delta}})$ holds only when $\mathbf{w}_u \in [0, \delta]^d$ for all $u \in \Omega$, for some $\delta \leq \frac{1}{2}$. For the Online Vector Bin Packing problem [4], while a hardness result $\Omega(d^{1-\epsilon})$ on the competitive ratio is proof for general instances with $\mathbf{w}_u \in [0, 1]^d$; when $\mathbf{w}_u \in [0, \frac{1}{B}]^d$ for some $B \geq 2$, they proposed an $O(d^{\frac{1}{B-1}} (\log d)^{\frac{B}{B-1}})$-competitive algorithm.

Another interesting open problem is whether the $O(k)$-competitive ratio can be improved for the problem under the "small weight assumption". Note that we have shown in Theorem 4 that achieving a constant competitive ratio is impossible.

2 Preliminaries

We use Ω to denote the set of items, which are not known by the algorithm initially and arrive one by one. Assume that each of the d dimensions of the knapsack has capacity 1. For $u \in \Omega$, the weight vector $\mathbf{w}_u \in [0, 1]^d$ is known to the algorithm only when item u arrives. A set $S \subset \Omega$ of items is *feasible* if $\sum_{u \in S} \mathbf{w}_u \leq \mathbf{1}$. The utility of S is $f(S)$, where f is a non-negative monotone submodular function. For a positive integer t, we use $[t]$ to denote $\{1, 2, \ldots, t\}$. We say that an item u is *discarded* if it is not accepted when it arrives; it is *disposed of* if it is accepted when it arrives, but later dropped to maintain feasibility.

Note that in general (without constant slack), no deterministic algorithm for the problem is competitive, even with linear utility function and when $d = k$. A similar result when $k = 1$ has been shown by Iwama and Zhang [25].

▶ **Lemma 5** (Generalization of [25]). *Any deterministic algorithm has a competitive ratio* $\Omega(\sqrt{\frac{k}{\epsilon}})$ *for the Online Vector Packing Problem with weight vectors in* $[0, 1-\epsilon]^d$*, even when the utility function is linear and* $d = k$*.*

Proof. Since the algorithm is deterministic, we can assume that the instance is adaptive.

Consider the following instance with $k = d$. Let the first item have value 1 and weight $1-\epsilon$ on all d dimensions; the following (small) items have value $\sqrt{\frac{\epsilon}{k}}$ and weight 2ϵ on one of the d dimension (and 0 otherwise). Stop the sequence immediately if the first item is not accepted. Otherwise let there be $\frac{1}{2\epsilon}$ items on each of the d dimensions. Note that to accept any of the "small" items, the first item must be disposed of. We stop the sequence immediately once the first item is disposed of.

It can be easily observe that we have either $\mathsf{ALG} = 1$ and $\mathsf{OPT} = \sqrt{\frac{k}{4\epsilon}}$, or $\mathsf{ALG} = \sqrt{\frac{\epsilon}{k}}$ and $\mathsf{OPT} \geq 1$, in both cases the competitive ratio is $\Omega(\sqrt{\frac{k}{\epsilon}})$. ◀

Note that the above hardness result (when $k = 1$) also holds for the Online Generalized Assignment Problem (with one offline node). We use OPT to denote both the optimal utility, and the feasible set that achieves this value. The meaning will be clear from the context.

3 Online Algorithm for Weight Vectors with Slack

In this section, we give an online algorithm for weight vectors with constant slack $\epsilon > 0$. Specifically, the algorithm is given some constant parameter $\epsilon > 0$ initially such that for all items $u \in \Omega$, its weight vector satisfies $\|\mathbf{w}_u\|_\infty \leq 1 - \epsilon$. On the other hand, the algorithm does not need to know upfront the upper bound k on the sparsity of the weight vectors.

3.1 Deterministic Online Algorithm

Notation. During the execution of an algorithm, for each item $u \in \Omega$, we use S^u and A^u to denote the feasible set of maintained items and the set of items that have ever been accepted, respectively, at the moment **just before** the arrival of item u.

We define the *value* of u as $\mathsf{v}(u) := f(u|A^u) = f(A^u \cup \{u\}) - f(A^u)$. Note that the value of an item depends on the algorithm and the arrival order of items. For $u \in \Omega$, for each $i \in [d]$, define the *density* of u at dimension i as $\rho_u(i) := \frac{\mathsf{v}(u)}{\mathbf{w}_u(i)}$ if $\mathbf{w}_u(i) \neq 0$ and $\rho_u(i) := \infty$ otherwise. By considering a lexicographical order on Ω, we may assume that all ties in values and densities can be resolved consistently.

For a vector $\mathbf{x} \in [0,1]^\Omega$, we use $\mathbf{x}(u)$ to denote the component corresponding to coordinate $u \in \Omega$. We overload the notation $\lceil \mathbf{x} \rceil$ to mean either the support $\lceil \mathbf{x} \rceil := \{u \in \Omega : \mathbf{x}(u) > 0\}$ or its indicator vector in $\{0,1\}^\Omega$ such that $\lceil \mathbf{x} \rceil(u) = \lceil \mathbf{x}(u) \rceil$.

Online Algorithm. The details are given in Algorithm 1, which defines the parameters $\beta := 1 - \epsilon$, $\alpha := \sqrt{\beta} = 1 - \Theta(\epsilon)$ and $\gamma := \frac{1}{2}(1 - \frac{\beta}{\alpha}) = \Theta(\epsilon)$. The algorithm keeps a (fractional) vector $\mathbf{s} \in [0,1]^\Omega$, which is related to the actual feasible set S maintained by the algorithm via the loop invariant (of the **for** loop in lines 2-24): $S = \lceil \mathbf{s} \rceil$. Specifically, when an item u arrives, the vector $\mathbf{s}$ might be modified such that the coordinate $\mathbf{s}(u)$ might be increased and/or other coordinates might be decreased; after one iteration of the loop, the feasible set S is changed according to the loop invariant. The algorithm also maintains an auxiliary vector $\mathbf{a} \in [0,1]^\Omega$ that keeps track of the maximum fraction of item u that has ever been accepted.

Algorithm Intuition. The algorithm solves a fractional variant behind the scenes using a linear objective function defined by v. For each dimension $i \in [d]$, it assumes that the capacity is $\beta < 1$. Upon the arrival of a new element $u \in \Omega$, the algorithm tries to increase the fraction of item u accepted via the parameter $\theta \in [0,1]$ in the **do**...**while** loop starting at line 16. For each dimension $i \in [d]$ whose capacity is saturated (at β) and $\mathbf{w}_u(i) > 0$, to further increase the fraction of item u accepted, some item u_i^θ with the least density ρ_i will have its fraction decreased in order to make room for item u. Hence, with respect to θ, the value decreases at a rate at most $\sum_i \mathbf{w}_u(i) \cdot \rho_i(u_i^\theta)$ due to disposing of fractional items. We keep on increasing θ as long as this rate of loss is less than γ times $\mathsf{v}(u)$ (which is the rate of increase in value due to item u).

After trying to increase the fraction of item u (and disposing of other items fractionally), the algorithm commits to this change only if at least α fraction of item u is accepted, in which case any item whose accepted fraction is less than β will be totally disposed of.

3.2 Competitive Analysis

For notational convenience, we use the superscripted versions (e.g., $\mathbf{s}^u$, $\mathbf{a}^u$, $S^u = \lceil \mathbf{s}^u \rceil$, $A^u = \lceil \mathbf{a}^u \rceil$) to indicate the state of the variables at the beginning of the iteration in the **for** loop (starting at line 2) when item u arrives. When we say the **for** loop, we mean the one that runs from lines 2 to 24. When the superscripts of the variables are removed (e.g., S and A), we mean the variables at some moment just before or after an iteration of the **for** loop.

We first show that the following properties are loop invariants of the **for** loop.

▶ **Lemma 6** (Feasibility Loop Invariant). *The following properties are loop invariants of the **for** loop:*

(a) *For every $i \in [d]$, $\sum_{v \in \Omega} \mathbf{s}(v) \cdot \mathbf{w}_v(i) \le \beta$, i.e., for every dimension, the total capacity consumed by the fractional solution $\mathbf{s}$ is at most β.*

(b) *The set $S = \lceil \mathbf{s} \rceil \subset \Omega$ is feasible for the original problem.*

Proof. Statement (a) holds initially because $\mathbf{s}$ is initialized to $\vec{0}$. Next, assume that for some item $u \in \Omega$, statement (a) holds for $\mathbf{s}^u$. It suffices to analyze the non-trivial case when the changes to $\mathbf{s}$ are committed at the end of the iteration. Hence, we show that statement (a) holds throughout the execution of the **do**...**while** loop starting at line 16. It is enough show that for each $i \in [d]$, $g_i(\theta) := \sum_{v \in \Omega} \mathbf{x}^\theta(v) \cdot \mathbf{w}_v(i) \le \beta$ holds while θ is being increased.

To this end, it suffices to prove that if $g_i(\theta) = \beta$, then $\frac{dg_i(\theta)}{d\theta} \le 0$. We only need to consider the case $\mathbf{w}_u(i) > 0$, because otherwise $g_i(\theta)$ cannot increase. By the rules updating $\mathbf{x}$, we have in this case $\frac{dg_i(\theta)}{d\theta} \le \frac{d\mathbf{x}^\theta(u)}{d\theta} \mathbf{w}_u(i) + \frac{d\mathbf{x}^\theta(u_i^\theta)}{d\theta} \mathbf{w}_{u_i^\theta}(i) \le 0$, as required.

We next show that statement (b) follows from statement (a). Line 20 ensures that between iterations of the **for** loop, for all $v \in S = \lceil \mathbf{s} \rceil$, $\mathbf{s}(v) \ge \beta$.

Hence, for all $i \in [d]$, we have $\sum_{v \in S} \mathbf{w}_v(i) \le \frac{1}{\beta} \sum_{v \in S} \mathbf{s}(v) \cdot \mathbf{w}_v(i) = \frac{1}{\beta} \sum_{v \in \Omega} \mathbf{s}(v) \cdot \mathbf{w}_v(i) \le 1$, where the last inequality follows from statement (a). ◀

For a vector $\mathbf{x} \in [0,1]^\Omega$, we define $\mathsf{v}(\mathbf{x}) := \sum_{u \in \Omega} \mathsf{v}(u) \cdot \mathbf{x}(u)$; for a set $X \subset \Omega$, we define $\mathsf{v}(X) := \sum_{u \in X} \mathsf{v}(u)$. Note that the definitions of $\mathsf{v}(\lceil \mathbf{x} \rceil)$ are consistent under the set and the vector interpretations.

The following simple fact (which is similar to Lemma 2.1 of [9]) establishes the connection between the values of items (defined by our algorithm) and the utility of the solution (defined by the submodular function f).

▶ **Fact 7** (Lemma 2.1 in [9]). *The **for** loop maintains the invariants $f(A) = f(\emptyset) + \mathsf{v}(A)$ and $f(S) \ge f(\emptyset) + \mathsf{v}(S)$, where $A = \lceil \mathbf{a} \rceil$ and $S = \lceil \mathbf{s} \rceil$.*

Algorithm 1: Online Algorithm

Parameters: $\alpha := \sqrt{1-\epsilon}, \beta := 1-\epsilon, \gamma := \frac{1}{2}(1-\sqrt{1-\epsilon})$

initialize $\mathbf{s}, \mathbf{a} \in [0,1]^{\Omega}$ as all zero vectors; ▷ $\lceil \mathbf{s} \rceil$ is the current feasible solution

for *each round when u arrives* **do**

  Define $\mathsf{v}(u) := f(u|\lceil \mathbf{a} \rceil)$;

  Initialize $\theta \leftarrow 0$, $\mathbf{x}^0 \leftarrow \mathbf{s}$;

  do

    Increase θ continuously (variables $\mathbf{x}^\theta$ and u_i^θ all depend on θ):

    for *every* $i \in [d]$ **do**

      if $\sum_{v\in\Omega} \mathbf{x}^\theta(v)\mathbf{w}_v(i) = \beta$ *and* $\mathbf{w}_u(i) > 0$ **then**

        Set $u_i^\theta \leftarrow \arg\min\{\rho_i(v) : v \in \Omega \setminus \{u\}, \mathbf{x}^\theta(v)\mathbf{w}_v(i) > 0\}$;

      end

      if $\sum_{v\in\Omega} \mathbf{x}^\theta(v)\mathbf{w}_v(i) < \beta$ *or* $\mathbf{w}_u(i) = 0$ **then**

        Set $u_i^\theta \leftarrow \perp$ and $\rho_i(u_i^\theta) \leftarrow 0$;

      end

    end

    Change $\mathbf{x}^\theta(v)$ (for all $v \in \Omega$) at rate:

$$\frac{d\mathbf{x}^\theta(v)}{d\theta} = \begin{cases} 1, & v = u; \\ -\max_{i\in[d]:u_i^\theta = v} \left\{ \frac{\mathbf{w}_u(i)}{\mathbf{w}_{u_i^\theta}(i)} \right\}, & v \in \{u_i^\theta\}_{i\in[d]}; \\ 0, & \text{otherwise.} \end{cases}$$

  while $\theta < 1$ *and* $\gamma \cdot \mathsf{v}(u) > \sum_{i\in[d]} \mathbf{w}_u(i) \cdot \rho_i(u_i^\theta)$;

  if $\theta \geq \alpha$ **then**

    $\mathbf{s} \leftarrow \mathbf{x}^\theta$, $\mathbf{a}(u) \leftarrow \mathbf{x}^\theta(u)$; ▷ update phase

    for $v \in \Omega$ *with* $\mathbf{s}(v) < \beta$ **do**

      $\mathbf{s}(v) \leftarrow 0$; ▷ dispose of small fractions

    end

  end

  ▷ if $\theta < \alpha$, then $\mathbf{s}$ and $\mathbf{a}$ will not be changed

end

return $\lceil \mathbf{s} \rceil$.

Our analysis consists of two parts. We first show that $\mathsf{v}(\mathbf{a})$ is comparable to the value of our real solution S in Lemma 8. Then, we compare in Lemma 9 the value of an (offline) optimal solution with $\mathsf{v}(\mathbf{a})$. Combining the two lemmas we are able to prove Theorem 10.

▶ **Lemma 8.** *The* ***for*** *loop maintains the invariant:* $(1-\frac{\beta}{\alpha}) \cdot \mathsf{v}(S) \geq (1-\frac{\beta}{\alpha}-\gamma) \cdot \mathsf{v}(\mathbf{a})$*, where* $S = \lceil \mathbf{s} \rceil$*. In particular, our choice of the parameters implies that* $\mathsf{v}(\mathbf{a}) \leq 2 \cdot \mathsf{v}(S)$*.*

Proof. We prove the stronger loop invariant that:

$$\mathsf{v}(\mathbf{s}) \geq (1-\gamma-\frac{\beta}{\alpha}) \sum_{r\in A\setminus S} \mathsf{v}(r) \cdot \mathbf{a}(r) + (1-\gamma) \sum_{r\in S} \mathsf{v}(r) \cdot \mathbf{a}(r),$$

where $S = \lceil \mathbf{s} \rceil$ is the current feasible set and $A \setminus S$ is the set of items that have been accepted at some moment but are already discarded.

The invariant holds trivially initially when $S = A = \emptyset$ and $\mathbf{s} = \vec{0}$. Suppose the invariant holds at the beginning of the iteration when item $u \in \Omega$ arrives. We analyze the non-trivial case when the item u is accepted into S, i.e., $\mathbf{s}$ and $\mathbf{a}$ are updated at the end of the iteration. Recall that $\mathbf{s}^u$ and $\mathbf{a}^u$ refer to the variables at the beginning of the iteration, and for the rest of the proof, we use the $\widehat{\mathbf{s}}$ and $\widehat{\mathbf{a}}$ to denote their states at the end of the iteration.

Suppose in the **do**...**while** loop, the parameter θ is increased from 0 to $\mathbf{a}(u) \geq \alpha$. Since for all $r \neq u$, $\mathbf{a}^u(r) = \widehat{\mathbf{a}}(r)$, we can denote this common value by $\mathbf{a}(r)$ without risk of ambiguity. We use $\mathbf{x}^u$ to denote the vector $\mathbf{x}^\theta$ when $\theta = \mathbf{a}(u)$. Then, we have

$$\begin{aligned}\mathsf{v}(\mathbf{x}^u) - \mathsf{v}(\mathbf{s}^u) &\geq \mathsf{v}(u) \cdot \mathbf{a}(u) - \int_0^{\mathbf{a}(u)} \sum_{i \in [d]: u_i^\theta \neq \perp} \left(\frac{\mathbf{w}_u(i)}{\mathbf{w}_{u_i^\theta}(i)} \cdot \mathsf{v}(u_i^\theta)\right) d\theta \\ &> \mathsf{v}(u) \cdot \mathbf{a}(u) - \int_0^{\mathbf{a}(u)} \gamma \cdot \mathsf{v}(u) d\theta = (1-\gamma) \cdot \mathsf{v}(u) \cdot \mathbf{a}(u),\end{aligned}$$

where the second inequality holds by the criteria of the **do**...**while** loop.

Next, we consider the change in value $\mathsf{v}(\widehat{\mathbf{s}}) - \mathsf{v}(\mathbf{x}^u)$, because some (fractional) items are disposed of in line 20. Let $D \subseteq S^u$ be such discarded items. Since an item is discarded only if its fraction is less than β, the value lost is at most $\beta \sum_{r \in D} \mathsf{v}(r) \leq \frac{\beta}{\alpha} \sum_{r \in D} \mathsf{v}(r) \cdot \mathbf{a}(r)$, where the last inequality follows because $\mathbf{a}(r) \geq \alpha$ for all items r that are ever accepted. Therefore, we have

$$\mathsf{v}(\widehat{\mathbf{s}}) - \mathsf{v}(\mathbf{x}^u) \geq -\frac{\beta}{\alpha} \sum_{r \in D} \mathsf{v}(r) \cdot \mathbf{a}(r).$$

Combining the above two inequalities, we have

$$\mathsf{v}(\widehat{\mathbf{s}}) - \mathsf{v}(\mathbf{s}^u) \geq (1-\gamma) \cdot \mathsf{v}(u) \cdot \mathbf{a}(u) - \frac{\beta}{\alpha} \sum_{r \in D} \mathsf{v}(r) \cdot \mathbf{a}(r).$$

Hence, using the induction hypothesis that the loop invariant holds at the beginning of the iteration, it follows that

$$\begin{aligned}\mathsf{v}(\widehat{\mathbf{s}}) \geq& (1 - \gamma - \frac{\beta}{\alpha}) \sum_{r \in A^u \setminus S^u} \mathsf{v}(r) \cdot \mathbf{a}(r) + (1-\gamma) \sum_{r \in S^u} \mathsf{v}(r) \cdot \mathbf{a}(r) + (1-\gamma) \cdot \mathsf{v}(u) \cdot \mathbf{a}(u) \\ &- \frac{\beta}{\alpha} \sum_{r \in D} \mathsf{v}(r) \cdot \mathbf{a}(r) \\ \geq& (1 - \gamma - \frac{\beta}{\alpha}) \sum_{r \in \widehat{A} \setminus \widehat{S}} \mathsf{v}(r) \cdot \mathbf{a}(r) + (1-\gamma) \sum_{r \in \widehat{S}} \mathsf{v}(r) \cdot \mathbf{a}(r),\end{aligned}$$

where $\widehat{A} = \lceil \widehat{\mathbf{a}} \rceil$ and $\widehat{S} = \lceil \widehat{\mathbf{s}} \rceil$, as required.

We next show that the stronger invariant implies the result of the lemma. Rewriting the invariant gives

$$\mathsf{v}(\mathbf{s}) \geq (1 - \gamma - \frac{\beta}{\alpha}) \sum_{r \in A} \mathsf{v}(r) \cdot \mathbf{a}(r) + \frac{\beta}{\alpha} \sum_{r \in S} \mathsf{v}(r) \cdot \mathbf{a}(r) \geq (1 - \gamma - \frac{\beta}{\alpha}) \sum_{r \in A} \mathsf{v}(r) \cdot \mathbf{a}(r) + \frac{\beta}{\alpha} \cdot \mathsf{v}(\mathbf{s}),$$

where the last inequality follows because $\mathbf{a}(r) \geq \mathbf{s}(r)$ for all $r \in S$. Finally, the lemma follows because $\mathsf{v}(S) = \mathsf{v}(\lceil \mathbf{s} \rceil) \geq \mathsf{v}(\mathbf{s})$. ◀

The following lemma gives an upper bound on the value of the items in a feasible set that are discarded right away by the algorithm.

▶ **Lemma 9.** *The* ***for*** *loop maintains the invariant that if* OPT *is a feasible subset of items that have arrived so far, then* $\gamma \cdot \mathsf{v}(\mathsf{OPT} \setminus A) \le \frac{k}{\beta(1-\alpha)} \cdot \mathsf{v}(\mathbf{a})$*, where* $A = \lceil \mathbf{a} \rceil$*. In particular, our choice of the parameters implies that* $\mathsf{v}(\mathsf{OPT} \setminus A) \le O(\frac{k}{\epsilon^2}) \cdot \mathsf{v}(S)$*.*

Proof. Consider some $u \in \mathsf{OPT} \setminus A$. Since $u \notin A$, in iteration u of the **for** loop, we know that at the end of the **do**...**while** loop, we must have $\theta < \alpha$, which implies $\gamma \cdot \mathsf{v}(u) \le \sum_{i \in [d]} \mathbf{w}_u(i) \cdot \rho_i(u_i^\theta)$ at this moment.

Recall that by definition, $\rho_i(u_i^\theta)$ is either (i) 0 in the case $\sum_{v \in \Omega} \mathbf{x}^\theta(v) \cdot \mathbf{w}_v(i) < \beta$ and $\mathbf{w}_u(i) > 0$, or (ii) the minimum density $\rho_i(v)$ in dimension i among items $v \neq u$ such that $\mathbf{x}^\theta(v) \cdot \mathbf{w}_v(i) > 0$.

Hence, in the second case, we have

$$\rho_i(u_i^\theta) \le \frac{\sum_{v \neq u : \mathbf{x}^\theta(v)\mathbf{w}_v(i) > 0} \mathbf{x}^\theta(v) \cdot \mathsf{v}(v)}{\sum_{v \neq u : \mathbf{x}^\theta(v)\mathbf{w}_v(i) > 0} \mathbf{x}^\theta(v) \cdot \mathbf{w}_v(i)} = \frac{\sum_{v \neq u : \mathbf{x}^\theta(v)\mathbf{w}_v(i) > 0} \mathbf{x}^\theta(v) \cdot \mathsf{v}(v)}{\beta - \theta \cdot \mathbf{w}_u(i)}$$
$$\le \frac{\sum_{v : \mathbf{w}_v(i) > 0} \mathbf{a}(v) \cdot \mathsf{v}(v)}{\beta(1-\alpha)} = \frac{V_i}{\beta(1-\alpha)},$$

where $V_i := \sum_{v : \mathbf{w}_v(i) > 0} \mathbf{a}(v) \cdot \mathsf{v}(v)$ depends only on the current $\mathbf{a}$ and $i \in [d]$. In the last inequality, we use $\theta \cdot \mathbf{w}_u(i) \le \alpha\beta$ and a very loose upper bound on the numerator. Observe that for the case (i) $\rho_i(u_i^\theta) = 0$, the inequality $\rho_i(u_i^\theta) \le \frac{V_i}{\beta(1-\alpha)}$ holds trivially.

Hence, using this uniform upper bound on $\rho_i(u_i^\theta)$, we have $\gamma \cdot \mathsf{v}(u) \le \sum_{i \in [d]} \mathbf{w}_u(i) \cdot \frac{V_i}{\beta(1-\alpha)}$.

Therefore, we have

$$\gamma \cdot \mathsf{v}(\mathsf{OPT} \setminus A) \le \sum_{u \in \mathsf{OPT} \setminus A} \sum_{i \in [d]} \mathbf{w}_u(i) \cdot \frac{V_i}{\beta(1-\alpha)} = \sum_{i \in [d]} \left(\sum_{u \in \mathsf{OPT} \setminus A} \mathbf{w}_u(i) \right) \cdot \frac{V_i}{\beta(1-\alpha)}$$
$$\le \sum_{i \in [d]} \frac{V_i}{\beta(1-\alpha)} \le \frac{k \cdot \mathsf{v}(\mathbf{a})}{\beta(1-\alpha),}$$

where the second to last inequality follows because $\mathsf{OPT} \setminus A$ is feasible, and $\sum_{i \in [d]} V_i \le k \cdot \mathsf{v}(\mathbf{a})$, because for each $v \in \Omega$, $|\{i \in [d] : \mathbf{w}_v(i) > 0\}| \le k$. ◀

▶ **Theorem 10.** *Algorithm 1 is* $O(\frac{k}{\epsilon^2})$*-competitive.*

Proof. Suppose OPT is a feasible subset. Recall that S is the feasible subset currently maintained by the algorithm. Then, by the monotonicity and the submodularity of f, we have $f(\mathsf{OPT}) \le f(\mathsf{OPT} \cup A) \le f(A) + \sum_{u \in \mathsf{OPT} \setminus A} f(u|A) \le f(\emptyset) + \mathsf{v}(A) + \mathsf{v}(\mathsf{OPT} \setminus A)$, where we use Fact 7 and submodularity $f(u|A) \le f(u|A^u) = \mathsf{v}(u)$ in the last inequality.

Next, observe that for all $u \in A$, $\mathbf{a}(u) \ge \alpha$. Hence, we have $\mathsf{v}(A) \le \frac{\mathsf{v}(\mathbf{a})}{\alpha} = O(1) \cdot \mathsf{v}(\mathbf{a})$. Combining with Lemma 9, we have $f(\mathsf{OPT}) \le f(\emptyset) + O(\frac{k}{\epsilon^2}) \cdot \mathsf{v}(\mathbf{a})$.

Finally, using Lemma 8 and Fact 7 gives $f(\mathsf{OPT}) \le O(\frac{k}{\epsilon^2}) \cdot f(S)$, as required. ◀

3.3 Hardness Results: Proof of Theorem 2

We show that for the Online Vector Packing Problem with slack $\epsilon \in (0, \frac{1}{2})$, no deterministic algorithm can achieve $o(k)$-competitive ratio, and no randomized algorithm can achieve $o(\frac{k}{\log k})$-competitive ratio. To prove the hardness result for randomized algorithms, we apply Yao's principle [29] and construct a distribution of hard instances, such that any deterministic

algorithm cannot perform well in expectation. Specifically, we shall show that each instance in the support of the distribution has offline optimal value $\Theta(\frac{k}{\log k})$, but any deterministic algorithm has expected objective value $O(1)$, thereby proving Theorem 2.

In our hard instances, the utility function is linear, and all items have the same value, i.e., the utility function is $f(S) := |S|$. Moreover, we assume all weight vectors are in $\{0, 1-\epsilon\}^d$, for any arbitrary $\epsilon \in (0, \frac{1}{2})$. Hence, we only need to describe the arrival order of items, and the non-zero dimensions of weight vectors. In particular, we can associate each item u with a k-subset of $[d]$. We use $\binom{[d]}{k}$ to denote the collection of k-subsets of $[d]$.

Notations. We say that two items are *conflicting*, if they both have non-zero weights on some dimension i (in which case, we say that they *conflict* with each other on dimension i). We call two items *non-conflicting* if they do not conflict with each other on any dimension.

Our hard instances show that in some case when items conflict with one another on different dimensions, the algorithm might be forced to make difficult decisions on choosing which item to accept. By utilizing the nature of unknown future, we show that it is very unlikely for any algorithm to make the right decisions on the hard instances. Although accepted items can be later disposed of to make room for (better) items, by carefully setting the weights and arrival order, we show that disposing of accepted items cannot help to get a better objective (hence in a sense, disabling free-disposal).

Hard instance for deterministic algorithms. Let $d := 2k^2$. Recall that each item is specified by an element of $\binom{[d]}{k}$, indicating which k dimensions are non-zero. Consider any deterministic algorithm. An arriving sequence of length at most $2k$ is chosen adaptively. The first item is picked arbitrarily, and the algorithm must select this item, or else the sequence stops immediately. Subsequently, in each round, the non-zero dimensions for the next arriving item u are picked according to the following rules.

1. Exactly $k-1$ dimensions from $[d]$ are chosen such that no previous item has picked them.
2. Suppose $\widehat{u} \in \binom{[d]}{k}$ is the item currently kept by the algorithm. Then, the remaining dimension i is picked from $\widehat{u}$ such that no other arrived item conflicts with $\widehat{u}$ on dimension i. If no such dimension i can be picked, then the sequence stops.

▶ **Lemma 11.** *Any deterministic algorithm can keep at most 1 item, while there exist at least k items that are mutually non-conflicting, implying that an offline optimal solution contains at least k items.*

Proof. By adversarial choice, every arriving item conflicts with the item currently kept by the algorithm. Hence, the algorithm can keep at most 1 item at any time.

We next show that when the sequence stops, there exist at least k items in the sequence that are mutually non-conflicting. For the case when there are $2k$ items in the sequence, consider the items in reversed order of arrival. Observe that each item conflicts with only one item that arrives before it. Hence, we can scan the items one by one backwards, and while processing a remaining item, we remove any earlier item that conflicts with it. After we finish with the scan, there are at least k items remaining that are mutually non-conflicting.

Suppose the sequence stops with less than $2k$ items. It must be the case that while we are trying to add a new item u, we cannot find a dimension i contained in the item $\widehat{u}$ currently kept by the algorithm such that no already arrived item conflicts with $\widehat{u}$ on dimension i. This implies that for every non-zero dimension i of $\widehat{u}$, there is already an item u_i conflicting with $\widehat{u}$ on that dimension. Since by choice, each dimension can cause a conflict between at most 2 items, these k items u_i's must be mutually non-conflicting. ◀

Distribution of Hard Instances. To use Yao's principle [29], we give a procedure to sample a random sequence of items. For some large enough integer ℓ that is a power of 2, define $k := 100\ell \log_2 \ell + 1$, which is the sparsity of the weight vectors. Observe that $\ell = \Theta(\frac{k}{\log k})$, and define $d := \ell + 400\ell^2 \log_2 \ell = O(\frac{k^2}{\log k})$ to be the number of dimensions. We express the set of dimensions $[d] = I \cup J$ as the disjoint union of $I := [\ell]$ and $J := [d] \setminus I$. The items arrive in ℓ phases, and for each $i \in [\ell]$, $4\ell - i + 1$ items arrive. Recall that each item is characterized by its k non-zero dimensions (where the non-zero coordinates all equal $1 - \epsilon > \frac{1}{2}$). We initialize $J_1 := J$. For i from 1 to ℓ, we describe how the items in phase i are sampled as follows.

1. Each of the $4\ell - i + 1$ items will have $i \in I = [\ell]$ as the only non-zero dimension in I.
2. Observe that (inductively) we have $|J_i| = (4\ell - i + 1) \cdot 100\ell \log_2 \ell$. We partition J_i randomly into $4\ell - i + 1$ disjoint subsets, each of size exactly $k - 1 = 100\ell \log_2 \ell$. Each such subset corresponds to the remaining $(k-1)$ non-zero dimensions of an item in phase i. These items in phase i can be revealed to the algorithm one by one.
3. Pick S_i from those $4\ell - i + 1$ subsets uniformly at random; define $J_{i+1} := J_i \setminus S_i$. Observe that the algorithm does not know S_i until the next phase $i + 1$ begins.

▶ **Claim 12.** *In the above procedure, the items corresponding to S_i's for $i \in [\ell]$ are mutually non-conflicting. This implies that there is an offline optimal solution containing $\ell = \Theta(\frac{k}{\log k})$ items. We say that those ℓ items are* good*, while other items are* bad*.*

We next show that bad items are very likely to be conflicting.

▶ **Lemma 13.** *Let $\mathcal{E}$ be the event that there exist two bad items that are non-conflicting. Then,* $\Pr[\mathcal{E}] \leq \frac{1}{\ell^2}$.

Proof. An alternative view of the sampling process is that the subsets $S_1, S_2, \ldots, S_\ell$ are first sampled for the good items. Then, the remaining bad items can be sampled independently across different phases (but note that items within the same phase are sampled in a dependent way).

Suppose we condition on the subsets $S_1, S_2, \ldots, S_\ell$ already sampled. Consider phases i and j, where $i < j$. Next, we further condition on all the random subsets generated in phase j for defining the corresponding items. We fix some bad item v in phase j.

We next use the remaining randomness (for picking the items) in phase i. Recall that each bad item in phase i corresponds to a random subset of size $k - 1 = 100\ell \log_2 \ell$ in $J_i \setminus S_i$, where $|J_i \setminus S_i| \leq 4(k-1)\ell$. If we focus on such a particular (random) subset from phase i, the probability that it is disjoint from the subset corresponding to item v (that we fixed from phase j) is at most $(1 - \frac{k-1}{4(k-1)\ell})^{k-1} \leq \exp(-25 \log_2 \ell) \leq \frac{1}{\ell^7}$.

Observe that there are in total at most $4\ell^2$ items. Hence, taking a union over all possible pairs of bad items, the probability of the event $\mathcal{E}$ is at most $(4\ell^2)^2 \cdot \frac{1}{\ell^7} \leq \frac{1}{\ell^2}$. ◀

▶ **Lemma 14.** *For any deterministic algorithm* ALG *applied to the above random procedure, the expected number of items kept in the end is $O(1)$.*

Proof. Let X denote the number of good items and Y denote the number of bad items kept by the algorithm at the end.

Observe that the sampling procedure allows the good item (corresponding to S_i) in phase i to be decided after the deterministic algorithm finishes making all its decisions in phase i. Hence, the probability that the algorithm keeps the good item corresponding to S_i is at most $\frac{1}{4\ell - i + 1} \leq \frac{1}{3\ell}$. Since this holds for every phase, it follows that $E[X] \leq \frac{1}{3\ell} \cdot \ell = \frac{1}{3}$.

Observe that conditioning on the complementing event $\overline{\mathcal{E}}$ (refer to Lemma 13), at most 1 bad item can be kept by the algorithm, because any two bad items are conflicting. Finally, because the total number of items is at most $4\ell^2$, we have $E[Y] = \Pr[\mathcal{E}]E[Y|\mathcal{E}]+\Pr[\overline{\mathcal{E}}]E[Y|\overline{\mathcal{E}}] \leq \frac{1}{\ell^2} \cdot 4\ell^2 + 1 \cdot 1 \leq 5$.

Hence, $E[X + Y] \leq 6$, as required. ◀

▶ **Corollary 15.** *By Claim 12 and Lemma 14, Yao's principle implies that for any randomized algorithm, there exists a sequence of items such that the value of an offline optimum is at least* $\Theta(\frac{k}{\log k})$*, but the expected value achieved by the algorithm is* $O(1)$*.*

References

1 Melika Abolhassani, T.-H. Hubert Chan, Fei Chen, Hossein Esfandiari, MohammadTaghi Hajiaghayi, Hamid Mahini, and Xiaowei Wu. Beating ratio 0.5 for weighted oblivious matching problems. In *ESA*, volume 57 of *LIPIcs*, pages 3:1–3:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

2 Gagan Aggarwal, Gagan Goel, Chinmay Karande, and Aranyak Mehta. Online vertex-weighted bipartite matching and single-bid budgeted allocations. In *SODA*, pages 1253–1264, 2011.

3 Yossi Azar, Ilan Reuven Cohen, Amos Fiat, and Alan Roytman. Packing small vectors. In *SODA*, pages 1511–1525. SIAM, 2016.

4 Yossi Azar, Ilan Reuven Cohen, Seny Kamara, and F. Bruce Shepherd. Tight bounds for online vector bin packing. In *STOC*, pages 961–970. ACM, 2013.

5 Moshe Babaioff, Jason D. Hartline, and Robert D. Kleinberg. Selling ad campaigns: online algorithms with cancellations. In *EC*, pages 61–70. ACM, 2009.

6 Nikhil Bansal, Nitish Korula, Viswanath Nagarajan, and Aravind Srinivasan. On k-column sparse packing programs. In *IPCO*, volume 6080 of *Lecture Notes in Computer Science*, pages 369–382. Springer, 2010.

7 Niv Buchbinder, Moran Feldman, and Roy Schwartz. Online submodular maximization with preemption. In *SODA*, pages 1202–1216. SIAM, 2015.

8 Niv Buchbinder and Joseph Naor. Online primal-dual algorithms for covering and packing. *Math. Oper. Res.*, 34(2):270–286, 2009.

9 T.-H. Hubert Chan, Zhiyi Huang, Shaofeng H.-C. Jiang, Ning Kang, and Zhihao Gavin Tang. Online submodular maximization with free disposal: Randomization beats ¼ for partition matroids. In *SODA*, pages 1204–1223. SIAM, 2017.

10 T.-H. Hubert Chan, Shaofeng H.-C. Jiang, Zhihao Gavin Tang, and Xiaowei Wu. Online submodular maximization problem with vector packing constraint. *CoRR*, abs/1706.06922, 2017.

11 Moses Charikar, Monika Henzinger, and Huy L. Nguyen. Online bipartite matching with decomposable weights. In *ESA*, volume 8737 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2014.

12 Chandra Chekuri, Jan Vondrák, and Rico Zenklusen. Submodular function maximization via the multilinear relaxation and contention resolution schemes. *SIAM J. Comput.*, 43(6):1831–1879, 2014.

13 Florin Constantin, Jon Feldman, S. Muthukrishnan, and Martin Pál. An online mechanism for ad slot reservations with cancellations. In *SODA*, pages 1265–1274. SIAM, 2009.

14 Nikhil R. Devanur, Zhiyi Huang, Nitish Korula, Vahab S. Mirrokni, and Qiqi Yan. Whole-page optimization and submodular welfare maximization with online bidders. In *EC*, pages 305–322. ACM, 2013.

15 Nikhil R. Devanur, Kamal Jain, and Robert D. Kleinberg. Randomized primal-dual analysis of ranking for online bipartite matching. In *SODA*, pages 101–107, 2013. doi:10.1137/1.9781611973105.7.

16 Nikhil R. Devanur, Kamal Jain, Balasubramanian Sivan, and Christopher A. Wilkens. Near optimal online algorithms and fast approximation algorithms for resource allocation problems. In *EC*, pages 29–38. ACM, 2011.

17 Leah Epstein, Asaf Levin, Danny Segev, and Oren Weimann. Improved bounds for online preemptive matching. In *STACS*, volume 20 of *LIPIcs*, pages 389–399. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

18 Jon Feldman, Monika Henzinger, Nitish Korula, Vahab S. Mirrokni, and Clifford Stein. Online stochastic packing applied to display ad allocation. In *ESA (1)*, volume 6346 of *Lecture Notes in Computer Science*, pages 182–194. Springer, 2010.

19 Jon Feldman, Nitish Korula, Vahab S. Mirrokni, S. Muthukrishnan, and Martin Pál. Online ad assignment with free disposal. In *WINE*, volume 5929 of *Lecture Notes in Computer Science*, pages 374–385. Springer, 2009.

20 Stanley P. Y. Fung. Online scheduling with preemption or non-completion penalties. *J. Scheduling*, 17(2):173–183, 2014.

21 M. R. Garey, Ronald L. Graham, David S. Johnson, and Andrew Chi-Chih Yao. Resource constrained scheduling as generalized bin packing. *J. Comb. Theory, Ser. A*, 21(3):257–298, 1976.

22 Xin Han, Yasushi Kawase, and Kazuhisa Makino. Online unweighted knapsack problem with removal cost. *Algorithmica*, 70(1):76–91, 2014.

23 Xin Han, Yasushi Kawase, and Kazuhisa Makino. Randomized algorithms for online knapsack problems. *Theor. Comput. Sci.*, 562:395–405, 2015.

24 Elad Hazan, Shmuel Safra, and Oded Schwartz. On the complexity of approximating k-set packing. *Computational Complexity*, 15(1):20–39, 2006.

25 Kazuo Iwama and Guochuan Zhang. Optimal resource augmentations for online knapsack. In *APPROX-RANDOM*, volume 4627 of *Lecture Notes in Computer Science*, pages 180–188. Springer, 2007.

26 Thomas Kesselheim, Klaus Radke, Andreas Tönnis, and Berthold Vöcking. Primal beats dual on online packing lps in the random-order model. In *STOC*, pages 303–312. ACM, 2014.

27 Marco Molinaro and R. Ravi. Geometry of online packing linear programs. In *ICALP (1)*, volume 7391 of *Lecture Notes in Computer Science*, pages 701–713. Springer, 2012.

28 Mourad El Ouali, Antje Fretwurst, and Anand Srivastav. Inapproximability of b-matching in k-uniform hypergraphs. In *WALCOM*, volume 6552 of *Lecture Notes in Computer Science*, pages 57–69. Springer, 2011.

29 Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *FOCS*, pages 222–227. IEEE Computer Society, 1977.

Faster Approximate Diameter and Distance Oracles in Planar Graphs

Timothy M. Chan[1] **and Dimitrios Skrepetos**[2]

1 **Department of Computer Science, University of Illinois at Urbana-Champaign, IL, USA**
tmc@illinois.edu

2 **Cheriton School of Computer Science, University of Waterloo, Ontario, Canada**
dskrepet@uwaterloo.ca

Abstract

We present an algorithm that computes a $(1+\varepsilon)$-approximation of the diameter of a weighted, undirected planar graph of n vertices with non-negative edge lengths in $O\left(n \log n \left(\log n + (1/\varepsilon)^5\right)\right)$ expected time, improving upon the $O\left(n\left((1/\varepsilon)^4 \log^4 n + 2^{O(1/\varepsilon)}\right)\right)$-time algorithm of Weimann and Yuster [ICALP 2013]. Our algorithm makes two improvements over that result: first and foremost, it replaces the exponential dependency on $1/\varepsilon$ with a polynomial one, by adapting and specializing Cabello's recent abstract-Voronoi-diagram-based technique [SODA 2017] for approximation purposes; second, it shaves off two logarithmic factors by choosing a better sequence of error parameters during recursion.

Moreover, using similar techniques, we improve the $(1+\varepsilon)$-approximate distance oracle of Gu and Xu [ISAAC 2015] by first replacing the exponential dependency on $1/\varepsilon$ on the preprocessing time and space with a polynomial one and second removing a logarithmic factor from the preprocessing time.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases planar graphs, diameter, abstract Voronoi diagrams

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.25

1 Introduction

In this paper we study the problem of computing the diameter of a weighted, undirected planar graph of n vertices with non-negative edge lengths[1], defined as the longest shortest path distance between two vertices of the graph. Since Frederickson in 1983 [7] solved the problem in $O\left(n^2\right)$ time (by determining the all-pairs shortest paths distance matrix and returning the largest value therein), a natural question arose as to whether the diameter can be computed in subquadratic time. Poly-logarithmic speedups were given by Chan [5] in 2006 and by Wulff-Nilsen [22] in 2010; the algorithm of the former works for the unweighted case and requires $O\left(n^2 \log\log n / \log n\right)$ time; the algorithm of the latter requires the same amount of time for the unweighted case and $O\left(n^2 (\log\log n)^4 / \log n\right)$ time for the weighted. However, a truly subquadratic algorithm, i.e., an algorithm running in $O\left(n^{2-\delta}\right)$ time for some constant $\delta > 0$, still eluded researchers for many years.

Thus, not surprisingly, the dearth of truly subquadratic algorithms led to the consideration of approximation algorithms. A c-approximation of the diameter, Δ, of a graph is

[1] For the rest of the introduction, we assume, unless otherwise stated, that all the discussed graphs are weighted, undirected planar graphs with non-negative edge lengths.

a value $\widetilde{\Delta}$ such that $\Delta \leq \tilde{\Delta} \leq c\Delta$. Using the linear-time SSSP algorithm of Henzinger et al. [12] one can trivially compute a 2-approximation. The first non-trivial approximation result was given by Berman et al. [2] in 2007; their algorithm requires $O\left(n^{3/2}\right)$ time and gives a 3/2-approximation. Weimann and Yuster [21] in 2012, in a breakthrough, presented an algorithm computing a $(1+\varepsilon)$-approximation of the diameter in near-linear time, namely $O\left(n\left((1/\varepsilon)^4 \log^4 n + 2^{O(1/\varepsilon)}\right)\right)$. Nevertheless, their solution did not settle the problem completely because the running time has *exponential* dependency on $1/\varepsilon$. Another problem with their solution is the multiple (four) logarithmic factors.

Unexpectedly, the next result came in the context of exact algorithms. In 2017, Cabello [3] (full paper in [4]) made headway, by giving the first exact truly subquadratic algorithm, requiring $\tilde{O}\left(n^{11/6}\right)$ expected time. The techniques used by Cabello are as interesting as the result itself, as he used a seemingly alien concept to planar graphs, *abstract Voronoi diagrams*, originating from computational geometry.

Cabello's seminal result bifurcates the study of the diameter problem into two main avenues. First, one could try to improve its running time. This has been partially treated in a recent paper by Gawrychowski et al. [8], who presented an algorithm requiring $\tilde{O}\left(n^{5/3}\right)$ worst-case time. No lower bound is available presently, but Cabello [4] conjectured that the diameter cannot be computed exactly in time faster than $O\left(n^{1+\delta}\right)$, for some constant $\delta > 0$. Second, one could try to use some of the techniques in Cabello's paper to approximate the diameter.

In this paper we take the second avenue. Namely, we improve the running time of Weimann and Yuster [21] by eliminating the $2^{O(1/\varepsilon)}$ factor. To do this, we adapt Cabello's technique involving abstract Voronoi diagrams. It turns out that a much simplified version of his technique is sufficient for approximation purposes, and can be combined nicely with Weimann and Yuster's algorithm. Our contribution however does not stop here; we also eliminate two of the four $\log n$ factors along the way, by using a better sequence of error parameters in the recursion from Weimann and Yuster's algorithm. Our main result is summarized by the following theorem.

▶ **Theorem 1** (Diameter). *Given a weighted, undirected planar graph of n vertices with non-negative edge lengths, we can compute a $(1+\varepsilon)$-approximation of its diameter in expected $O\left(n \log n \left(\log n + (1/\varepsilon)^5\right)\right)$ time.*

Another important problem in planar graphs is the construction of efficient $(1+\varepsilon)$-approximate distance oracles, i.e., data structures that in a query for a pair of vertices u, v of a planar graph G, return a value $\tilde{d}$ such that $d_G(u,v) \leq \tilde{d} \leq (1+\varepsilon)d_G(u,v)$, where $d_G(u,v)$ is the shortest path distance from u to v in G. Thorup [20] presented a $(1+\varepsilon)$-approximate distance oracle, requiring $O\left((1/\varepsilon)^2 n \log^3 n\right)$ preprocessing time, $O\left((1/\varepsilon) n \log n\right)$ space, and $O(1/\varepsilon)$ query time, later simplified by Klein [15]. Kawarabayashi et al. [14] improved the dependency on $1/\varepsilon$ of the space-query time product from $1/\varepsilon^2$ to $1/\varepsilon$. Gu and Xu [9] combined the ideas of those results with the techniques of the diameter algorithm of Weimann and Yuster [21] to obtain the first distance oracle with constant query time (independent of both n and ε); it requires $O\left(n \log n \left((1/\varepsilon)^2 \log^3 n + 2^{O(1/\varepsilon)}\right)\right)$ preprocessing time and $O\left(n \log n \left((1/\varepsilon) \log n + 2^{O(1/\varepsilon)}\right)\right)$ space.

Using similar techniques as the ones for our diameter result, we can also improve the $(1+\varepsilon)$-approximate distance oracle of Gu and Xu [9]; namely, we eliminate the exponential dependency on $1/\varepsilon$ on the preprocessing time and space and at the same time remove a logarithmic factor from the preprocessing time.

▶ **Theorem 2** (Distance Oracle). *Given a weighted, undirected planar graph of n vertices with non-negative edge lengths, we can construct a $(1+\varepsilon)$-approximate distance oracle, requiring*

$O(1)$ *query time,* $O\left(n \log n \left(\log n + (1/\varepsilon)^5\right) + (1/\varepsilon)^6\right)$ *expected preprocessing time, and* $O\left(n \log n \left(\log n + (1/\varepsilon)^6\right)\right)$ *space.*

Throughout the paper we operate under the standard RAM model of computation. Let $[W] = \{1, \ldots, W\}$. We assume that all the planar graphs under discussion have a fixed, combinatorial embedding and are triangulated.

2 A Streamlined Version of Cabello's Technique

The main purpose of this section is to construct the following farthest neighbor data structure, which will be crucial in obtaining our diameter result in Section 3:

▶ **Theorem 3** (Farthest neighbor). *Let H be a weighted, undirected planar graph of n vertices with non-negative edge lengths and W be an integer. Let X be a set of b vertices on the boundary of the outer face of H. Let H^+ be the graph obtained by adding to H a vertex z_0, with an edge from z_0 to each vertex $x \in X$ of unspecified length.*

We can preprocess H in $O\left(nb^3W^2\right)$ expected time, such that the following query can be answered in $O(b \log b)$ time: given lengths drawn from $[W]$ for the b edges z_0x $(x \in X)$, find the distance to the farthest neighbor of z_0 in H^+, i.e., compute $\max_{u \in V(H)} d_{H^+}(z_0, u)$.

In our application, $b = W = O(1/\varepsilon)$, so the preprocessing time would be near linear in n. Cabello established a similar theorem [4, Theorem 21] for the more general setting where each edge z_0x $(x \in X)$ may have a real length, but his preprocessing time bound is $\tilde{O}\left(n^2b^3 + b^4\right)$. We show that when the length of each edge z_0x $(x \in X)$ is a small integer, the preprocessing time can be greatly improved, and at the same time the method becomes simpler.

To avoid degeneracies we need to ensure uniqueness of the shortest paths. That can be done by perturbing the lengths of the edges of H with known techniques (e.g., see [11]). Note that we do not need to perturb the weights of the sites, which remain integers.

2.1 Defining Voronoi diagrams in planar graphs

The general concept of abstract Voronoi diagrams in $\mathbb{R}^2$ was defined by Klein [17]. Cabello [4] applied the concept to planar graphs with weighted sites. We reiterate here the main definitions for the sake of completeness.

Each *site* s of a Voronoi diagram in a planar graph is a pair (v_s, w_s), where v_s is the site's placement, i.e., a vertex of the graph, and w_s is its weight. Given a graph G and a set of sites S, the *Voronoi region* of a site $s \in S$ is defined as $\mathrm{VR}_G(s, S) = \{u \in V(G) \mid d_G(v_s, u) + w_s \leq d_G(v_t, u) + w_t, \forall\, t \in S - \{s\}\}$, i.e., as the set of all vertices closer to s than to any other site under the weighted metric; the *Voronoi diagram* of S is defined as $\mathrm{VD}_G(S) = \mathbb{R}^2 \setminus \bigcup_{s \in S} \mathrm{VR}_G(s, S)$.

A key concept in Voronoi diagrams is bisectors. The bisector of two sites s and t, $\mathrm{bis}_G(s, t)$, is defined as the set of the duals of the edges in $E_G(s, t) = \{uv \in E(G) \mid d_G(u, v_s) + w_s \leq d_G(u, v_t) + w_t$ and $d_G(v, v_t) + w_t \leq d_G(v, v_s) + w_s\}$, i.e., the bisector contains the duals of all the edges whose endpoints are not both closer to the same site. Let $\{p, q\}$ be a generic (i.e., for each $u \in V(G)$ we have $d_G(u, v_p) + w_p \neq d_G(u, v_q) + w_q$) and independent (i.e., each Voronoi region is non-empty) set of sites on the boundary of the *outer* face of a planar graph G. Then the bisector of p and q is a simple cycle in the dual, passing through the dual vertex, v_∞, of the outer face ([4, Lemma 5]).

As Cabello [4] showed, a Voronoi diagram in a planar graph for b sites on the boundary of the outer face fulfills Klein's axioms of abstract Voronoi diagrams [17], so it can be represented abstractly as a collection of *Voronoi vertices* and *Voronoi edges*, forming a

planar graph itself of size $O(b)$. A Voronoi edge corresponds to a simple path in the dual (subpath of a bisector), and a Voronoi vertex corresponds to the meeting point of three Voronoi edges.

2.2 Computing abstract Voronoi diagrams in planar graphs

Since abstract Voronoi diagrams can be constructed efficiently by an existing algorithm by Klein et al. [18] based on randomized incremental construction, we have:

▶ **Theorem 4** (Abstract Voronoi diagram construction)**.** *We can construct the abstract Voronoi diagram in a planar graph with b sites on the outer face, using an expected $O(b \log b)$ number of elementary operations. Here, an elementary operation refers to the computation of the abstract Voronoi diagram of any four sites.*

To prove Theorem 3, we need to construct the abstract Voronoi diagram in the graph H for the b sites at X quickly for any given assignment of weights on X from $[W]$, after an initial preprocessing that does not depend on the weights. By Theorem 4, it suffices to show how to compute the abstract Voronoi diagram of any four such sites.

We start by showing how to compute all different bisectors, given two vertices of X as placements of sites, whose weights are drawn from $[W]$, by building upon [4, Lemma 17]. We need $O(nW)$ total time for constructing the bisectors, whereas Cabello needed $O\left(n^2\right)$ time for general real weights; we can return a pointer to a bisector in $O(1)$ time instead of $O(\log n)$ time.

▶ **Lemma 5** (Bisectors)**.** *Given two vertices $v_s, v_t \subseteq X$ as placements of sites, the family of bisectors $\mathrm{bis}_H((v_s, w_s), (v_t, w_t))$, over all possible weights w_s and w_t drawn from $[W]$, has at most $O(W)$ different bisectors. We can compute all these bisectors in $O(nW)$ total time, such that, given two weights $w_s, w_t \in [W]$, we can return a pointer to the relevant bisector in $O(1)$ time.*

Proof. Assuming w.l.o.g. that $w_s \geq w_t$, we can write $\mathrm{bis}_H((v_s, w_s), (v_t, w_t))$ as $\mathrm{bis}_H((v_s, w), (v_t, 0))$, where $w = w_s - w_t$, so we need to consider only $\mathrm{bis}_H((v_s, w), (v_t, 0))$, where $w \in [W]$. Hence, there can be at most $O(W)$ different bisectors for a pair of sites.

Let $s = (v_s, w_s), t = (v_t, w_t)$, and $S = \{s, t\}$. For each vertex $u \in V(H)$ we compute the value $\eta_u = d_H(v_t, u) - d_H(v_s, u)$, by first running the linear-time SSSP algorithm of Henzinger et al. [12] from v_s and v_t and then visiting each vertex; $u \in V(H)$ belongs to $\mathrm{VD}_H(s, \{s, t\})$ when $w \leq \eta_u$ and to $\mathrm{VD}_H(t, \{s, t\})$ otherwise. For each $w \in [W]$ we compute the bisector $\mathrm{bis}_H((v_s, w), (v_t, 0))$, by marking each edge $uv \in E(G)$ such that $w \leq \eta_u$ and $w > \eta_v$. The bisector is composed of the duals of the marked edges and is a cycle in the dual, passing through v_∞; we represent it as a linked list, $\mathrm{LL}_{s,t,w}$. Finally, we store the linked-list representation of every different bisector $\mathrm{bis}_H((v_s, w), (v_t, 0))$ in a table, $\mathrm{T}_{s,t}$, indexed by w.

The total time spent is $O(nW)$ because we have at most W different bisectors, and computing each takes $O(n)$ time. Given two sites s and t with weights w_s and $w_t \in [W]$ respectively, we can return a pointer to the pertinent bisector in $O(1)$ time by looking up $\mathrm{T}_{s,t}[w]$, assuming w.l.o.g. that $w_s \geq w_t$. ◀

Next, we show how to compute all different Voronoi diagrams, given three vertices of X as placements of sites, whose weights are drawn from $[W]$, by building upon [4, Lemma 18]. We need $O\left(nW^2\right)$ time, whereas Cabello had $O\left(n^2\right)$; we can return a pointer to a Voronoi diagram in $O(1)$ time instead of $O(\log n)$. Furthermore, our proof is simpler since it does not involve line arrangements and amortization.

▶ **Lemma 6** (Abstract Voronoi diagrams of three sites). *Given three vertices $v_s, v_t, v_q \subseteq X$ as placements of sites, the family of Voronoi diagrams over all possible weights $w_s, w_t, w_q \in [W]$, has at most $O\left(W^2\right)$ different Voronoi diagrams. We can compute all these Voronoi diagrams in $O\left(nW^2\right)$ total time, such that given weights $w_s, w_t, w_q \in [W]$ we can return a pointer to the relevant Voronoi diagram in $O(1)$ time.*

Proof. We invoke Lemma 5 to compute and store all the different bisectors of each pair of the three sites in $O(nW)$ time. We can assume w.l.o.g. that $w_q = 0$, so there are at most $O\left(W^2\right)$ different Voronoi diagrams. Let $s = (v_s, w_s), t = (v_t, w_t), q = (v_q, w_q)$, and $S = \{s, t, q\}$. For each vertex $u \in V(H)$ we compute the values $\eta_x^{st} = d_H(v_s, u) - d_H(v_t, u)$, $\eta_x^{qt} = d_H(v_q, u) - d_H(v_t, u)$, and $\eta_x^{sq} = d_H(v_s, u) - d_H(v_q, u)$ by running the SSSP algorithm of [12]; u belongs to $\mathrm{VR}_H(s, \{s, t, q\})$ if $\eta_u^{st} \leq w_t - w_s$ and $\eta_u^{sq} \leq -w_s$; similar statements can be made for $\mathrm{VR}_H(t, \{s, t, q\})$ and $\mathrm{VR}_H(q, \{s, t, q\})$.

For every $w_s, w_t \in [W]$, we find in linear time the Voronoi diagram $\mathrm{VD}_H(S)$ as follows. Each bisector (i) does not participate at all, (ii) participates wholly, or (iii) only a subpath of it, passing through v_∞, participates in $\mathrm{VD}_H(S)$ (that is implied by that fact that $\mathrm{VD}_H(S)$ has at most one vertex besides v_∞; see [4, Lemma 13]). We provide two pointers for each bisector, which mark the bisector's part that constitutes a Voronoi edge: one for its first and one for its last edge participating in $\mathrm{VD}_H(S)$. Starting from v_∞, we scan the edges of each bisector in clockwise order; the first pointer of $\mathrm{bis}_H(s, t)$ is created for the first encountered edge uv such that u is closer to s than to t and to q, and v is closer to t or q than to s (which can be determined using their η values). The second pointer is created for the last such edge. If no such edges are encountered, both pointers are set to NULL. We can find in $O(1)$ time if there exists a Voronoi vertex; to do so, we scan each triple of pointers of the bisectors to see if the corresponding edges meet at a common dual vertex. If that is the case, we set that vertex to be a Voronoi vertex.

The representation of $\mathrm{VD}_H(S)$ for weights $w_s, w_t \in [W]$ is composed of (i) a linked list of each bisector participating in it, (ii) the first and last pointers of each such bisector, and (iii) the one, if any, Voronoi vertex therein, besides v_∞. Each different Voronoi diagram is stored in a two-dimensional table $\mathrm{T}_{s,t,q}$, indexed by w_s and w_t. Given weights w_s, w_t, and w_q, where w.l.o.g $w_s, w_t \geq q_q$ we can return a pointer to the pertinent Voronoi diagram in $O(1)$ time by looking up $\mathrm{T}_{s,t,q}[w_s - w_q, w_t - w_q]$, assuming w.l.o.g that $w_s \geq w_q$ and $w_t \geq w_q$. ◀

The final step before using Theorem 4 is to provide a data structure that, given four vertices of X as placements of sites, whose weights are drawn from $[W]$, returns their Voronoi diagram. The following lemma builds upon [4, Lemma 19]; we refer the reader therein for the proof. Our preprocessing time is $O\left(nb^3W^2\right)$, whereas Cabello had $O\left(n^2b^3\right)$; also our query time is $O(1)$ instead of $O(\log n)$.

▶ **Lemma 7** (Abstract Voronoi diagrams of four sites). *We can construct a data structure, such that (i) its preprocessing time is $O\left(nb^3W^2\right)$, and (ii) for any four vertices of X as placements of sites that are generic, independent and have weights drawn from $[W]$, their abstract Voronoi diagram can be computed in $O(1)$ time.*

Now we can use Lemma 7 and Theorem 4 to compute the abstract Voronoi diagram of b sites, given all the vertices of X as placements of sites, whose weights are drawn from $[W]$. Our preprocessing time is $O\left(nb^3W^2\right)$ expected, whereas Cabello had $O\left(n^2b^3\right)$; our query time is $O(b \log b)$, while Cabello had multiple $\log n$ factors.

▶ **Theorem 8** (Abstract Voronoi diagrams in planar graphs)**.** *Let H, X, n, b, and W be as in Theorem 3. We can preprocess H in $O\left(nb^3W^2\right)$ time, such that, given the vertices of X as placements of sites, whose weights are drawn from $[W]$, we can compute the Voronoi diagram of the sites in $O(b \log b)$ expected time.*

Proof. We first construct the data structure of Lemma 7, which after $O\left(nb^3W^2\right)$ preprocessing time can compute the abstract Voronoi diagram of any set of four sites in $O(1)$ time. Then, using Theorem 4, we compute the abstract Voronoi diagram of the b sites in $O(b \log b)$ time. ◀

2.3 Constructing the farthest neighbor data structure

As one last ingredient for our farthest neighbor data structure, we need the following lemma, taken almost verbatim from [4, Corollary 6].

▶ **Lemma 9.** *Let F be an undirected planar graph of n vertices, each having a* cost $c(u) > 0$*, and let q_0 be one of them. Let $\Pi = \{\pi_1, \ldots, \pi_\ell\}$ be a family of simple paths in the dual of F with a total of h edges, counted with multiplicity. After $O(n+h)$ preprocessing time, we can answer the following query in $O(k)$ time: given a q_0-star-shaped cycle γ in the dual, i.e., a cycle such that (i) q_0 is in the interior of γ, and (ii) for every vertex in the interior of γ its shortest path to q_0 is fully contained in γ, described as a concatenation of k subpaths from Π, return $\max_{u \in U}\left(V_{\text{int}}\left(\gamma, F\right)\right)$, where $V_{\text{int}}(\gamma, F)$ is the set of vertices of F enclosed by γ.*

We can now prove the main theorem of this section.

Proof of Theorem 3. We construct the data structure of Theorem 8 for H, for a set S of b sites where each one is placed in a different vertex of X and apply Lemma 5 to compute the bisector of each pair of sites. For each site $s \in S$ and each bisector $\text{bis}_H(s, \cdot)$, we assign a cost to every vertex $u \in V(H)$, equal to $d_H(v_s, u)$, and construct the data structure of Lemma 9 (a bisector $\text{bis}_H(s,t)$ enclosing s is an s-star shaped cycle in the dual), where $F = H$, Π is the set of bisectors, $\ell = bW$, $h = nbW$, and $k = b$. The preprocessing time is $O\left(nb^3W^2\right)$.

In a query, we compute the abstract Voronoi diagram of H, where for each $s \in S$ we set w_s to be equal to the given length of the edge w_0v_s, by using the data structure of Theorem 8. For each site $s \in S$, we query the data structure of Lemma 9 for s to find the vertex of $\text{VR}_H(s, S)$ with the largest distance from s by walking along its boundary, which is the concatenation of at most b subpaths of the bisectors $\text{bis}_H(s, \cdot)$. From Lemma 9 we need $O(b)$ time to find $\max_{u \in \text{VR}_H(s,S)}\{d_H(v_s, u) + w_s\}$. We return the maximum of those distances. Thus, the total query time is $O(b \log b)$. ◀

3 Improving Weimann and Yuster's Diameter Approximation Algorithm

For approximating the diameter, we employ the recursive scheme of Weimann and Yuster [21], which is as follows.

Let $\mathcal{G}$ be the original graph and N its size. Let $d\left(G_1, G_2, G_3\right)$ denote the longest shortest path distance between a marked vertex of G_1 and a marked vertex of G_2 in G_3. Initially $G = \mathcal{G}$, n is the size of G, all vertices are marked, and we want to approximate $d\left(G, G, G\right)$. Let $\varepsilon > 0$. The outline of the recursive scheme is as follows.

1. Find a cycle C of G, such that the removal of C's vertices decomposes G into two disjoint and connected planar graphs A and B, each having between $n/3$ and $2n/3$ vertices; C may have up to n vertices. Let $G_{\text{in}} = A \cup C$ and $G_{\text{out}} = B \cup C$ and assume w.l.o.g. that A (resp. B) lies inside (resp. outside) C.

2. Approximate $d(G_{\text{in}}, G_{\text{out}}, G)$ (Sections 2.1 and 2.2 in [21]).
3. Unmark all vertices of C and build graphs G^+_{in} and G^+_{out} such that (i) they are planar, and connected graphs, (ii) each of G^+_{in} and G^+_{out} has at most roughly $2n/3$ vertices, (iii) together they have at most roughly n vertices, and (iv) $d\left(G_{\text{in}}, G_{\text{in}}, G^+_{\text{in}}\right)$ is a $(1+\varepsilon')$-approximation of $d(G_{\text{in}}, G_{\text{in}}, G)$ for an appropriate choice of parameter ε'. Then recurse in G^+_{in} to approximate $d\left(G_{\text{in}}, G_{\text{in}}, G^+_{\text{in}}\right)$ and do the same for G^+_{out} (Section 2.3 in [21]).
4. Return $\max\left\{d(G_{\text{in}}, G_{\text{out}}, G), d\left(G_{\text{in}}, G_{\text{in}}, G^+_{\text{in}}\right), d\left(G_{\text{out}}, G_{\text{out}}, G^+_{\text{out}}\right)\right\}$.

3.1 Decomposing G to G_{in} and G_{out}

To decompose the graph G into two subgraphs G_{in} and G_{out}, we use a shortest path separator, similarly to Thorup's work [20]. We compute the shortest path tree T of an arbitrarily selected marked vertex z of G in linear time by employing the algorithm of Henzinger et al. [12]. Let $\widetilde{\Delta} = \max_{u \in V(G)} d_G(v,u)$; we know that $\widetilde{\Delta} \leq \Delta \leq 2\widetilde{\Delta}$. We can find in linear time (see [19, Lemma 2]) two paths P and Q, both starting at v, such that the removal of the vertices on $V(C)$, where $C = P \cup Q$, from G gives us two disjoint planar subgraphs A and B, where $V(A)$ (resp. $V(B)$) contains the vertices of $V(G)$ that are strictly inside (resp. outside) C and $|V(A)|, |V(B)| \leq 2n/3$. The size of C, however, can be as big as n. The graph G_{in} (resp. G_{out}) is the graph induced by $A \cup C$ (resp. $B \cup C$). The time to decompose the graph is $O(n)$.

3.2 Reducing $d(G_{\text{in}}, G_{\text{out}}, G)$ to $d(G_{\text{in}}, G_{\text{out}}, G_{\text{p}})$

Before approximating $d(G_{\text{in}}, G_{\text{out}}, G)$ we need to address the following issue. A shortest path between a marked vertex of G_{in} and another in G_{out} has to go through a vertex of C. However, since C can have as many as n vertices, we cannot consider for each such pair all the vertices of C; instead, we select only a small subset of vertices of C and construct a graph G_{p} that allows us to approximate the distance between every aforementioned pair. The following lemma can be found in [21, Section 2.1 and Lemma 2.1].

▶ **Lemma 10.** *We can select a set Y of $O(1/\varepsilon)$ vertices (called* portals*) on C in linear time, such that if $d(G_{\text{in}}, G_{\text{out}}, G) \geq \widetilde{\Delta}$, then $\max_{u \in V(G_{\text{in}}), v \in V(G_{\text{out}})} \min_{y \in Y}\{d_G(u,y) + d_G(y,v)\}$ is a $(1+2\varepsilon)$-approximation of $d(G_{\text{in}}, G_{\text{out}}, G)$. Otherwise, it is at most $(1+2\varepsilon)\widetilde{\Delta}$.*

We run an SSSP algorithm from each portal of Y in G; let ℓ be the largest distance found. We construct a graph $G_{\text{p}} = G_{\text{p,in}} \cup G_{\text{p,out}}$, such that $V(G_{\text{p,in}}) = V(A) \cup Y$ and $V(G_{\text{p,out}}) = Y \cup V(B)$. We create an edge between each vertex of $G_{\text{p,out}}$ and each portal, whose length is equal to their shortest path distance in G, after rounding it to the closest multiple of $\varepsilon\ell$ and dividing it by that number. The edges between vertices of $G_{\text{p,in}}$ are the same as in G, but their lengths are also divided by $\varepsilon\ell$. The total time for the reduction is $O((1/\varepsilon)n)$.

3.3 Approximating $d(G_{\text{in}}, G_{\text{out}}, G_{\text{p}})$

We construct the farthest neighbor data structure of Theorem 3 for $H = G_{\text{in}}$, $X = Y$, $b = O(1/\varepsilon)$, and $W = 1/\varepsilon$. Then, we query it n times, by using each vertex $u \in V(G_{\text{out}})$ as z_0 and setting $w(z_0, x) = d_{G_{\text{out}}}(u,x)$ for each $x \in X$, to find its farthest neighbor among the vertices of $V(G_{\text{in}})$. Finally, we return the maximum of the distances found, multiplied by $\varepsilon\ell$. The total time for approximating $d(G_{\text{in}}, G_{\text{out}}, G_p)$ is thus $O\left(nb^3W^2 + nb\log b\right) = O\left((1/\varepsilon)^5 n\right)$.

Weimann and Yuster's paper [21] did not use a farthest neighbor data structure but instead employed a brute-force search, observing that there are only $2^{O(1/\varepsilon)}$ combinatorially different vertices in A and B (in terms of their vectors of distances to the portals). This is where we eliminate the exponential dependency on ε from their algorithm.

3.4 Reducing $d(G_{\text{in}}, G_{\text{in}}, G)$ to $d\left(G_{\text{in}}, G_{\text{in}}, G^{+}_{\text{in}}\right)$

After approximating $d(G_{\text{in}}, G_{\text{out}}, G)$, we need to approximate $d(G_{\text{in}}, G_{\text{in}}, G)$ and $d(G_{\text{out}}, G_{\text{out}}, G)$; however, we cannot directly recurse in G_{in} and G_{out} respectively because two problems arise (since the treatment is symmetrical for both G_{in} and G_{out}, we concern ourselves only with the former). First, a path realizing $d(G_{\text{in}}, G_{\text{in}}, G)$ could have a subpath lying in G_{out}. Second, G_{in} can have up to $O(n)$ vertices because C, which is part of G_{in}, itself could have that many. However, G_{in} has at most $2n/3$ marked vertices, which are the only ones used for computing $d(G_{\text{in}}, G_{\text{in}}, G)$. Therefore, we need to construct graphs G^{+}_{in} and G^{+}_{out} such that (i) they are planar, and connected graphs, (ii) each of G^{+}_{in} and G^{+}_{out} has at most roughly $2n/3$ vertices, (iii) together they have at most roughly n vertices, and (iv) $d\left(G_{\text{in}}, G_{\text{in}}, G^{+}_{\text{in}}\right)$ is a $(1+\varepsilon')$-approximation of $d(G_{\text{in}}, G_{\text{in}}, G)$ for an appropriate choice of parameter ε'. The following lemma is from [21, Lemma 2.3].

▶ **Lemma 11.** *If $d(G_{\text{in}}, G_{\text{in}}, G) \geq \widetilde{\Delta}$, then $d\left(G_{\text{in}}, G_{\text{in}}, G^{+}_{\text{in}}\right)$ is a $(1+2\varepsilon')$-approximation of $d(G_{\text{in}}, G_{\text{in}}, G)$. Otherwise, $d\left(G_{\text{in}}, G_{\text{in}}, G^{+}_{\text{in}}\right) \leq (1+2\varepsilon')\widetilde{\Delta}$.*

As in the algorithm of Weimann and Yuster, to construct G^{+}_{in}, we start by umarking all vertices of C and selecting $O(1/\varepsilon')$ vertices therein, called *dense portals*, similarly to Section 3.2. Let B_{in} be the union of the shortest paths between every pair of dense portals in G_{out}. We produce a graph B'_{in}, where we keep all the vertices of B_{in} of degree more than two and shrink the rest. Since there are $O(1/\varepsilon')$ dense portals, there are $O\left((1/\varepsilon')^2\right)$ such shortest paths. Also, any pair of those paths shares at most one subpath since we assume that shortest paths are unique, so there are at most $O\left((1/\varepsilon')^4\right)$ vertices of B_{in} of degree more than two, i.e., $V(B'_{\text{in}}) = O\left((1/\varepsilon')^4\right)$. It remains to show (i) how to compute B'_{in} and (ii) how to set ε'. These two points are where we deviate from the approach of Weimann and Yuster.

First, to construct B'_{in}, we do not construct B_{in} explicitly, as their algorithm does, which would require $O((1/\varepsilon')n)$ time. By using a slightly modified version of the multiple-source shortest paths data structure of Klein [16] we construct B'_{in} in $O\left(n\log n + (1/\varepsilon')^4 \log n\right)$ time instead. Second, Weimann and Yuster chose a fixed value for ε' for every recursive call. Since the recursion has $O(\log N)$ levels and error accumulates, they were forced to set $\varepsilon' = \varepsilon/\log N$, and so the $(1/\varepsilon')^4$ factors in the running time resulted in four $\log N$ factors. Here, we make ε' adaptive, i.e., dependent on the current input size n. Specifically, we set $\varepsilon' = \varepsilon/n^{1/8}$. With this choice of ε' we show that the approximation factor of our algorithm remains $1 + O(\varepsilon)$ (Section 3.5) and the final running time has only two $\log N$ factors (Section 3.5).

▶ **Theorem 12.** *We can build the graph B'_{in} in $O\left(n\log n + (1/\varepsilon')^4 \log n\right)$ time.*

Proof. We apply the multiple-source shortest paths data structure of Klein, which preprocesses a planar graph F of n vertices in $O(n\log n)$ time, such that given a vertex u on the boundary of the outer face and another vertex v, we can query the shortest path tree of u to find the distance to v in $O(\log n)$ time. We need to augment that data structure to also support the following two queries on the shortest path tree of a vertex on the boundary of

the outer face: (i) find the lowest common ancestor of any two vertices; and (ii) find the level ancestor of any vertex and any level. To do that, we just replace the (persistent) dynamic trees used internally in Klein's data structure with the (persistent) *top-tree* structures of Alstrup et al. [1], so we can support both queries in $O(\log n)$ time. We construct the data structure for $F = G_{\text{out}}$, after redrawing in linear time such that C lies on the outer face.

We build a list Γ that contains all the vertices of G_{out} that have degree more than three in B_{in}; as argued before, $|\Gamma| = O\left((1/\varepsilon')^4\right)$. There are three possibilities for each pair of shortest paths between dense portals: the paths do not intersect, they intersect only at one vertex, or they share a common subpath starting on a vertex p_1 and ending at another vertex p_2. Our goal is to find for each such pair the vertices p_1 and p_2 (which may not exist, may be the same, or may be distinct) and insert them to Γ. We focus on finding p_1 since finding p_2 is similar. Suppose that the first shortest path is from a to b and the second from c to d. What makes the problem nontrivial is that the two paths are not available explicitly.

We find p_1 by performing a binary search on the a-to-b shortest path as follows. Let p' and p'' be initially set to a and b respectively. Let p be the vertex midway between p' and p'' on the a-to-b path, which can be found by a level ancestor query. We want to find whether p is (i) between p_1 and p_2 (i.e., on the c-to-d path), (ii) between a and p_1, or (iii) between p_2 and b. To do so, we find the lowest common ancestor *lca* of p and d on the shortest path tree of c. If $lca = p$, then we are in case (i). Else, we perform a level ancestor query for p and d to find the children $\hat{p}$ and $\hat{d}$ of *lca* that lie on the *lca*-to-p and *lca*-to-d paths respectively and compare the order of $\hat{p}$ and $\hat{d}$ around *lca*. We assume w.l.o.g. that c is between a and b in P. If $\hat{p}$ is to the left of $\hat{d}$, then we are in case (ii), else we are in case (iii).[2] For case (i) or (iii) we recurse with $p'' = p$; for case (ii) we recurse with $p' = p$. We stop when $p' = p''$.

Once we are done with every pair of paths, we shrink every vertex in $V(G_{\text{out}}) - \Gamma$, thus procuring B'_{in}. ◀

We unmark all vertices of B'_{in} and append it to G_{in} to create the graph G'_{in}, which is a planar graph and has $|V(G_{\text{in}})| + O\left((1/\varepsilon')^4\right)$ vertices. Then, we have to shrink G'_{in}, such that it will have at most $2n/3$ vertices (remember that $|V(G_{\text{in}})|$ could be as big as $O(n)$). As in [21], we walk down on C and do the following steps. For any consecutive pair y_i and y_{i+1} of dense portals on C we create an edge between them of weight equal to their shortest path distance in G. Then we visit all the vertices $p_1, \ldots, p_k$ between y_i and y_{i+1} on C. For each vertex u having an edge to such a vertex, we create an edge between u and y_i of weight equal to $\min_j\{\ell(u, p_j) + d_G(p_j, y_i)\}$. Finally, we delete all vertices $p_1, \ldots, p_k$ and their incident edges. We call the resulting graph G^+_{in}; it has $2n/3 + O\left(1/(\varepsilon')^4\right)$ vertices. G^+_{out} is constructed similarly. The total time spent is $O\left(n \log n + (1/\varepsilon')^4 \log n\right)$.

3.5 Analyzing the approximation factor and the running time

▶ **Lemma 13.** *The approximation factor of our algorithm is* $1 + O(\varepsilon)$.

Proof. Let $G^{(\mu)}$ be the graph of a node μ of the recursion tree, and $G^{(\mu)}_{\text{in}}$ and $G^{(\mu)}_{\text{out}}$ be the two graphs created by decomposing it, as in Section 3.1. We $(1 + O(\varepsilon))$-approximate $d\left(G^{(\mu)}_{\text{in}}, G^{(\mu)}_{\text{out}}, G^{(\mu)}\right)$ for each node μ of the recursion tree, so the approximation factor of our algorithm is $(1+O(\varepsilon)) \max_\mu \frac{d\left(G^{(\mu)}_{\text{in}}, G^{(\mu)}_{\text{out}}, G^{(\mu)}\right)}{d\left(G^{(\mu)}_{\text{in}}, G^{(\mu)}_{\text{out}}, \mathcal{G}\right)} \leq (1+O(\varepsilon)) \prod_i (1+\varepsilon_i)$ where $\varepsilon_i = \varepsilon / n_i^{1/8}$,

[2] If the shortest path tree is not unique, we pick the right-most one; see [16] for details.

for some sequence $n_1, n_2, \ldots, n_k$ satisfying $n_{i-1}/3+\Theta((1/\varepsilon_i)^4) \le n_i \le 2n_{i-1}/3+\Theta((1/\varepsilon_i)^4)$ with $n_1 = N$ and $n_k = O\left((1/\varepsilon)^4\right)$.

Now, $\prod_i(1+\varepsilon_i) \le \exp\left(\sum_i \varepsilon_i\right)$. Since n_i decreases at least exponentially, ε_i grows at least exponentially; thus the sum $\sum_i \varepsilon_i$ is similar to a geometric series and can be bounded by the last term, which is $O(\varepsilon)$. Therefore, the approximation factor of our algorithm is $(1+O(\varepsilon))(1+O(\varepsilon)) = 1+O(\varepsilon)$ (which can be refined to $1+\varepsilon$ after adjusting ε by a constant factor). ◀

▶ **Lemma 14.** *The running time of our algorithm is* $O\left(N \log N \left(\log N + (1/\varepsilon)^5\right)\right)$.

Proof. The running time satisfies the following recurrence relation:

$$T(n) \le \max_{1/3\le\alpha\le 2/3} \left(T\left(\alpha n + O\left((1/\varepsilon)^4\sqrt{n}\right)\right) + T\left((1-\alpha)n + O\left((1/\varepsilon)^4\sqrt{n}\right)\right) + O\left(n\left(\log n + (1/\varepsilon)^5\right)\right)\right).$$

In the base case $n = O\left((1/\varepsilon)^4\right)$, so we can run a quadratic-time APSP algorithm in $O\left(n^2\right)$ time. Since we have $O\left(\varepsilon^4 N\right)$ such graphs, the total time for the base case is $O\left((1/\varepsilon)^4 N\right)$. The solution of the recurrence is $T(N) = O\left(N \log N \left(\log N + (1/\varepsilon)^5\right)\right)$. ◀

This completes the proof of Theorem 1. It is not difficult to see that in the same amount of time we can also compute a $(1+\varepsilon)$-approximation of the radius and of the Wiener index of the graph and of the eccentricity of each node.

4 Conclusion

Gawrychowski et al. [8] recently improved Cabello's algorithm [4] for computing the exact diameter in planar graphs; their algorithm is *deterministic* instead of randomized and requires $\tilde{O}(n^{5/3})$ time instead of $\tilde{O}(n^{11/6})$. It is worth investigating whether the techniques therein could be used to make our approximation algorithm deterministic and perhaps shave off some $1/\varepsilon$ factors. Another possible research direction is generalizing the techniques for the case of directed graphs.

An interesting consequence of our result is that we can compute the *exact* diameter of an *unweighted* planar graph in $O\left(n \log n \left(\log n + \Delta^{O(1)}\right)\right)$ expected time, where Δ is the diameter, simply by setting ε near $1/\Delta$. If one wants running time near linear in n, the best previous result we are aware of was by Eppstein [6] and had exponential dependence in Δ (namely, the time bound is $O\left(n2^{\Delta \log \Delta}\right)$). Note that our result beats Cabello's or Gawrychowski et al.'s algorithm when the diameter is smaller than n^δ for some constant δ.

References

1 Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005.

2 Piotr Berman and Shiva Prasad Kasiviswanathan. Faster approximation of distances in graphs. In *Proceedings of the Tenth International Workshop on Algorithms and Data Structures*, pages 541–552. Springer, 2007.

3 Sergio Cabello. Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs. In *Proceedings of the Twenty-eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2143–2152, 2017.

4 Sergio Cabello. Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs. *CoRR*, abs/1702.07815v1, 2017.

5 Timothy M. Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. *ACM Transactions on Algorithms*, 8:1–17, 2012.

6 David Eppstein. Subgraph isomorphism in planar graphs and related problems. In *SODA*, volume 95, pages 632–640, 1995.

7 Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

8 Paweł Gawrychowski, Haim Kaplan, Shay Mozes, Micha Sharir, and Oren Weimann. Voronoi diagrams on planar graphs, and computing the diameter in deterministic $\tilde{O}(n^{5/3})$ time. *CoRR*, abs/1704.02793v1, 2017.

9 Qian-Ping Gu and Gengchun Xu. Constant query time $(1+\varepsilon)$-approximate distance oracle for planar graphs. In *Proceedings of the Twenty-sixth International Symposium on Algorithms and Computation*, pages 625–636. Springer, 2015.

10 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

11 David Hartvigsen and Russell Mardon. The all-pairs min cut problem and the minimum cycle basis problem on planar graphs. *SIAM Journal on Discrete Mathematics*, 7(3):403–418, 1994.

12 Monika R. Henzinger, Philip N. Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.

13 Ken-ichi Kawarabayashi, Philip N. Klein, and Christian Sommer. Linear-space approximate distance oracles for planar, bounded-genus and minor-free graphs. In *Proceedings of the Thirty-eight International Colloquium on Automata, Languages, and Programming*, pages 135–146. Springer, 2011.

14 Ken-ichi Kawarabayashi, Christian Sommer, and Mikkel Thorup. More compact oracles for approximate distances in undirected planar graphs. In *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 550–563, 2013.

15 Philip N. Klein. Preprocessing an undirected planar network to enable fast approximate distance queries. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 820–827, 2002.

16 Philip N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the Sixteenth Annual Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 146–155, 2005.

17 Rolf Klein. *Concrete and abstract Voronoi diagrams*, volume 400. Springer Science & Business Media, 1989.

18 Rolf Klein, Kurt Mehlhorn, and Stefan Meiser. Randomized incremental construction of abstract Voronoi diagrams. In *Informatik*, pages 283–308. Springer, 1992.

19 Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

20 Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM*, 51(6):993–1024, 2004.

21 Oren Weimann and Raphael Yuster. Approximating the diameter of planar graphs in near linear time. *ACM Transactions on Algorithms*, 12(1):1–13, 2016.

22 Christian Wulff-Nilsen. *Algorithms for planar graphs and graphs in metric spaces*. PhD thesis, PhD thesis, University of Copenhagen, 2010.

A Approximate Distance Oracles

To construct an approximate distance oracle, we build upon the general framework of the oracles of Thorup, Kawarabayashi et al., and Gu and Xu ([20, 14] and [9] respectively). Given a weighted, undirected planar graph $\mathcal{G}$ with non-negative edge lengths, we will focus on constructing a distance oracle with additive stretch $\varepsilon\Delta$ (also called *additive distance oracle*), where Δ is the diameter of $\mathcal{G}$. Such an oracle returns, given two vertices u and v of $\mathcal{G}$, an approximation $\hat{d}$ of their distance $d_{\mathcal{G}}(u,v)$ in $\mathcal{G}$, such that $d_{\mathcal{G}}(u,v) \leq \hat{d} \leq d_{\mathcal{G}}(u,v)+\varepsilon\Delta$. A known *scaling* technique (see Kawarabayashi et al. [13] and Section 4 in [9]) can convert the additive distance oracle to a $(1+\varepsilon)$-approximate distance oracle.

To construct the additive distance oracle, we recursively decompose the given graph $\mathcal{G}$ as in Sections 3.1 and 3.4, but here we also store the graphs in a tree, called the *recursive decomposition tree*. Let N be the size of $\mathcal{G}$. Let μ be an internal node of the recursive decomposition tree, $G^{(\mu)}$ its graph, and $n = |V\left(G^{(\mu)}\right)|$. In the root ν of the tree, $G^{(\nu)} = \mathcal{G}$. Let $C^{(\mu)}$ be the shortest path separator used to decompose $G^{(\mu)}$ into two disjoint and connected planar subgraphs $G^{(\mu)}_{\text{in}}$ and $G^{(\mu)}_{\text{out}}$ as described in Section 3.1. We find a set $Y^{(\mu)}$ of $O(1/\varepsilon)$ vertices on $C^{(\mu)}$, called *portals*, such that we can approximate *any* shortest path between any $u \in V\left(G^{(\mu)}_{\text{in}}\right)$ and $v \in V\left(G^{(\mu)}_{\text{out}}\right)$ by routing it through one of these portals. Let $\widetilde{\Delta}$ be a 2-approximation of the diameter of $\mathcal{G}$, computed as in 3.1. To find the portals we use Lemma 10, slightly changed for the current setting.

▶ **Lemma 15.** *We can select a set $Y^{(\mu)}$ of $O(1/\varepsilon)$ vertices, where $\varepsilon > 0$, on $C^{(\mu)}$ in linear time, such that $d_{G^{(\mu)}}(u,v) \leq \min_{y\in Y^{(\mu)}}\{d_{G^{(\mu)}}(u,y) + d_{G^{(\mu)}}(y,v)\} \leq d_{G^{(\mu)}}(u,v) + 2\varepsilon\widetilde{\Delta}$ for any $u \in V\left(G^{(\mu)}_{\text{in}}\right)$ and $v \in V\left(G^{(\mu)}_{\text{out}}\right)$.*

We prove the following theorem (similar to Theorem 3), which is crucial into substituting the exponential dependency on $1/\varepsilon$ in the space and the preprocessing time of the additive distance oracle in [9] with a polynomial one, while retaining the constant query time.

▶ **Theorem 16.** *Let H be a weighted, undirected planar graph of n vertices with non-negative edge lengths and W be an integer. Let X be a set of b vertices on the boundary of the outer face of H. Let H^+ be the graph obtained by adding to H a vertex z_0, with an edge from z_0 to each vertex $x \in X$ of unspecified length. Let there be $O(n)$ different b-tuples of lengths for those edges.*

We can preprocess H in $O\left(nb^3W^2 + b^4W^2\right)$ expected preprocessing time and space, such that the following query can be answered in $O(1)$ time: given an $O(\log n)$-bit identifier of one of those tuples and a vertex $u \in V(H)$, return $d_{H^+}(z_0,u)$.

Proof. We create a set S of b sites where each site is placed on a different vertex of X. For each pair of sites we construct all the different bisectors by using Lemma 5. There are $O(W)$ bisectors for each pair of sites (Lemma 5), so there are $O\left(b^2W\right)$ bisectors in total. For each bisector and each vertex we store a boolean flag which is set to true if the vertex is enclosed by the bisector and false otherwise (that can be done by a variant of BFS) in $O\left(nb^2W\right)$ time. We find all the $O\left(b^4W^2\right)$ pieces of the graph into which it is decomposed by all the bisectors in that much time. The boundary of each such piece is the concatenation of at most b bisectors. For each vertex of H we find in $O(b)$ time the piece that it belongs to and store a pointer to it in $O(nb)$ total time.

For each tuple of lengths we construct the abstract Voronoi diagram of S in $O\left(nb^3W^2\right)$ time, by using Lemma 8, find in $O\left(b^4W^2\right)$ time the Voronoi region enclosing each piece of the previous paragraph, and create a pointer to it. We store the pointer of each piece in a hash table using the identifier of each tuple, but we do not store any abstract Voronoi

diagram. The total preprocessing time is $O\left(nb^3W^2 + b^4W^2\right)$ expected. The space required is $O\left(n + b^4W^2\right)$.

In a query, given a $O(\log n)$-bits identifier representing a tuple of lengths and a vertex $u \in V(H)$, we find the piece containing u, by using u's pointer, and then query the hash table of that piece to find the site s, such that $v_s = \arg\min_{t \in S}\{d_H(v_t, u) + w_t\}$ by using the given identifier as key. Then, we return $d_H(v_s, u) + w_s$, which is $d_{H^+}(z_0, u)$. The query time is constant. ◀

We run an SSSP algorithm from the portals of every internal node μ of the recursive decomposition tree. Let ℓ be the largest distance found. We construct the data structure of Theorem 16 for $H = G_{\text{in}}^{(\mu)}$, after dividing the length of every edge therein by $\varepsilon\ell$, $X = Y^{(\mu)}$, $b = O(1/\varepsilon)$, and $W = 1/\varepsilon$. Each of the n tuples of lengths corresponds to the tuple of the shortest path distances of a different vertex of $G_{\text{out}}^{(\mu)}$, after rounding each to the closest multiple of $\varepsilon\ell$ and dividing by that number, to the portals. Each such tuple is provided with a unique $O(\log n)$-bits identifier.

We create graphs $G_{\text{in}}^{(\mu)+}$ and $G_{\text{out}}^{(\mu)+}$, where $\varepsilon' = \varepsilon/n^{1/8}$, in $O\left(n \log n + (1/\varepsilon')^4 \log n\right)$ time, as in Section 3.4, assign them to the children of μ, and recurse. We stop when the size of the graph is $O(1/\varepsilon)$. The height of the recursive decomposition tree is $O(\log n)$. For each leaf node of the tree we run a brute-force APSP algorithm and store the distance matrix. We also preprocess the tree as in [10], such that we can answer lowest common ancestor queries in $O(1)$ time.

To answer a query, given two vertices u and v, let μ_u and μ_v respectively be the nodes of the recursive decomposition tree containing it. If $\mu_u = \mu_v$ and μ_u is a leaf, we return the shortest path distance from u to v by visiting the distance matrix therein. Else, we find in $O(1)$ time their lowest common ancestor $\mu_{u,v}$; supposing w.l.o.g. that $u \in V\left(G_{\text{in}}^{(\mu_{u,v})}\right)$ and $v \in V\left(G_{\text{out}}^{(\mu_{u,v})}\right)$, we properly query the data structure of Theorem 16 of $\mu_{u,v}$, and return the distance found, multiplied with $\varepsilon\ell$. Similarly to Lemma 13 we have the following lemma.

▶ **Lemma 17.** *For two vertices $u, v \in V(\mathcal{G})$ the additive oracle returns an value $\hat{d}$ such that $d_{\mathcal{G}}(u, v) \leq \hat{d} \leq d_{\mathcal{G}}(u, v) + O(\varepsilon)\Delta$.*

Finally, we bound the query time, the space, and the preprocessing time of our additive distance oracle.

▶ **Theorem 18.** *The space occupied by the additive oracle is $O\left(N\left(\log N + (1/\varepsilon)^6\right)\right)$, the preprocessing time required is $O\left(N\left(\log N\left(\log N + (1/\varepsilon)^5\right) + (1/\varepsilon)^6\right)\right)$, and a query can be answered in $O(1)$ time.*

Proof. It takes $O(1)$ time to find the lowest common ancestor of two nodes and to query the distance of any vertices therein. It also takes $O(1)$ time to query the distance matrix in a leaf. Therefore, the query time is $O(1)$.

The preprocessing time $T(n)$ and space $S(n)$ satisfy the following recurrence relations:

$$\begin{aligned} T(n) \quad &\leq \quad \max_{1/3 \leq \alpha \leq 2/3} \left(T\left(\alpha n + O\left((1/\varepsilon)^4\sqrt{n}\right)\right) + T\left((1-\alpha)n + O\left((1/\varepsilon)^4\sqrt{n}\right)\right) + \right. \\ &\qquad \left. O\left(n\left(\log n + (1/\varepsilon)^5\right) + (1/\varepsilon)^6\right)\right), \\ S(n) \quad &\leq \quad \max_{1/3 \leq \alpha \leq 2/3} \left(S\left(\alpha n + O\left((1/\varepsilon)^4\sqrt{n}\right)\right) + S\left(\beta n + O\left((1/\varepsilon)^4\sqrt{n}\right)\right) + O\left(n + (1/\varepsilon)^6\right)\right). \end{aligned}$$

In the base case $n = O\left(1/\varepsilon^4\right)$, so we run a quadratic-time APSP algorithm in $O\left(n^2\right)$ time. Since we have $O\left(\varepsilon^4 N\right)$ such graphs, the total time for the base case is $O\left((1/\varepsilon)^4 N\right)$. The solutions to the recurrences are $T(N) = O\left(N\left(\log N\left(\log N + (1/\varepsilon)^5\right) + (1/\varepsilon)^6\right)\right)$ and $S(N) = O\left(N\left(\log N + (1/\varepsilon)^6\right)\right)$. ◀

Stability and Recovery for Independence Systems*

Vaggos Chatziafratis[1], Tim Roughgarden†[2], and Jan Vondrak[3]

1 **Stanford University, Computer Science Department, Stanford, CA, USA**
vaggos@stanford.edu
2 **Stanford University, Computer Science Department, Stanford, CA, USA**
tim@stanford.edu
3 **Stanford University, Department of Mathematics, Stanford, CA, USA**
jvondrak@stanford.edu

Abstract

Two genres of heuristics that are frequently reported to perform much better on "real-world" instances than in the worst case are *greedy algorithms* and *local search algorithms*. In this paper, we systematically study these two types of algorithms for the problem of maximizing a monotone submodular set function subject to downward-closed feasibility constraints. We consider *perturbation-stable* instances, in the sense of Bilu and Linial [11], and precisely identify the stability threshold beyond which these algorithms are guaranteed to recover the optimal solution. Byproducts of our work include the first definition of perturbation-stability for non-additive objective functions, and a resolution of the worst-case approximation guarantee of local search in p-extendible systems.

1998 ACM Subject Classification I.1.2 Analysis of Algorithms, G.2.1 Combinatorial Algorithms

Keywords and phrases Submodular, approximation, stability, Local Search, Greedy, p-systems

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.26

1 Introduction

Designing polynomial-time approximation algorithms with worst-case guarantees is one of the most common approaches to coping with NP-hard optimization problems. For many problems, even the best-achievable worst-case guarantee (assuming $P \neq NP$) is too weak to be immediately meaningful. Fortunately, it has been widely observed that most approximation algorithms typically compute solutions that are much better than their worst-case approximation guarantee would suggest (e.g. [17, 37]). Is there a mathematical explanation for this phenomenon?

One line of work addresses this question by restricting attention to instances that satisfy a *stability* condition, stating that there should be a "sufficiently prominent" optimal solution. Such conditions are analogs of the "large margin" assumptions that are often made in machine learning theory. Such assumptions reflect the belief that the instances arising in practice are ones that have a "meaningful solution". For example, if we run a clustering algorithm on a data set, it's because we're expecting that a "meaningful clustering" exists. The hope is that formalizing the assumption of a "meaningful solution" imposes additional structure on an instance that provably makes the problem easier than on worst-case instances.

Several such stability notions have been studied. In this work, we focus on the most well-studied one, that of *perturbation-stability* introduced by Bilu and Linial [11]. The idea

* Omitted proofs can be found in the full version of this paper: https://arxiv.org/abs/1705.00127v1.
† TR was supported by NSF award CCF-1524062.

behind the definition is that the optimal solution should be robust to small changes in the input (e.g., the edge weights of a graph). For if this is not true, then a minor misspecification of the data (which is often noisy in practice, anyways) can change the output of the algorithm. In data analysis, one is certainly hoping that the conclusions reached are not sensitive to small errors in the data. An informal definition of γ-perturbation-stability (henceforth simply γ-*stability*) is the following:

▶ **Definition 1** (γ-stability)**.** Given a weighted graph and an optimal solution S^* for some problem, we say that the instance is γ-*stable* if S^* remains the unique optimal solution, even when each edge weight is increased by an (edge-dependent) factor between 1 and γ.

Thus 1-stability is equivalent to the assumption that the optimal solution is unique. The bigger the γ, the stronger the assumption (since there are fewer instances we are required to solve), and hence, the easier the problem. The basic question is then **whether sufficiently stable instances of computationally hard problems are easier to solve**. The ultimate goal is to determine the *stability threshold* of a problem: the smallest value of γ such that the problem is polynomial-time solvable on γ-stable instances. We note that there is no general connection between hardness of approximation thresholds and stability thresholds of a problem – depending on the problem, each could be larger than the other (e.g., [7], where even though asymmetric k-center cannot be approximated to any constant factor, it can be solved optimally under 2-stability). Thus a good approximation algorithm need not recover an optimal solution in stable instances, and conversely.[1]

1.1 Our Results

Two genres of algorithms that are frequently reported to perform much better on "real-world" instances than in the worst case are *greedy algorithms* and *local search algorithms*. The goal of this paper is to systematically study these two types of algorithms through the lens of perturbation-stability. We carry this out for the rich and well-motivated class of problems that concern maximizing a monotone submodular set function subject to downward-closed feasibility constraints (as in e.g. [29, 34, 22]). Both greedy and local search algorithms can be naturally defined for all problems in this class. Special cases include [31] k-dimensional matching, asymmetric traveling salesman, influence maximization [24], welfare maximization in combinatorial auctions (with submodular valuations) [27, 41], and so on.

We organize our results along two different axes: whether the objective function is additive or submodular, and according to the "complexity" of the feasibility constraints. For the latter, we use the classic notions of the intersection of p matroids (for a parameter p), the more general notion of p-extendible systems (where a feasible solution can accommodate a new element after deleting at most p old ones), and the still more general notion of p-systems (where the cardinality of maximal independent sets can only differ by a p factor). Figure 1 summarizes our main results. We also prove that all of our results are tight.

Section 3 proves our results for the greedy algorithm in the case of additive objective functions. An interesting finding here is that for the most general set systems that we consider (p-systems), the greedy algorithm can have an infinite stability threshold, even

[1] For a silly example, consider an algorithm that checks if an instance is stable (by brute-force), if so returns the optimal solution (computed by brute force), and if not returns a terrible solution. Similarly, consider an α-approximation algorithm that uses brute force to always output a suboptimal solution, in every instance where one within α of optimal exists. For more natural (and polynomial-time) examples, see [7, 30].

Table 1: Additive Approximation

	Greedy	Local Search
p-Matroids	p	p
p-extendible	p	p^2 (new)
p-system	p	fails (new)

Table 2: Additive Recovery (new)

	Greedy	Local Search
p-Matroids	p	p
p-extendible	p	p^2
p-system	fails	fails

Table 3: Submodular Approximation

	Greedy	Local Search
p-Matroids	$p+1$	$p+1$
p-extendible	$p+1$	p^2+1 (new)
p-system	$p+1$	fails (new)

Table 4: Submodular Recovery (new)

	Greedy	Local Search
p-Matroids	$p+1$	$p+1$
p-extendible	$p+1$	p^2+1
p-system	fails	fails

Figure 1 Summary of old and new results. On the left we have previous approximation results about greedy and local search algorithms [25, 34, 22, 38] and our new local search approximation guarantees. (Each table entry indicates the worst-case approximation factor.) On the right are our recovery results for greedy and local search algorithms, with each table entry indicating the smallest γ such that the algorithm is optimal in every γ-stable instance. All of the results are tight.

though it is a good worst-case approximation algorithm. In fact, this crucial difference between approximation and stability also led us to give a different characterization of the p-extendible systems. Another interesting differentiation between stability and approximation shows up in the case of a uniform matroid (cardinality constraints).

Section 4 considers the greedy algorithm for maximizing a monotone submodular function. As all previous works on perturbation-stability have considered only problems with additive objective functions, here we need to formulate a notion of perturbation-stability for submodular functions, which boils down to defining the class of allowable perturbations of a submodular function f. The "sweet spot" – neither too restrictive nor too permissive – turns out to be the set of perturbed functions $\tilde{f}$ such that: (i) $\tilde{f}$ is also monotone and submodular; (ii) $\tilde{f}$ is a pointwise approximation of f ($\tilde{f}(S) \in [f(S), \gamma \cdot f(S)]$ for every S); and (iii) the marginal value of an element j with respect to a set S can only go up (in $\tilde{f}$), and by at most $(\gamma - 1)$ times the stand-alone value of j.[2] This definition specializes to the usual one in the special case of additive functions. Towards the end of this section, we also present an application for the welfare maximization problem, for which a weaker stability assumption is sufficient to guarantee recovery of the optimal allocation.

Section 5 identifies the smallest γ such that all local optima of γ-stable instances are also global optima, with both additive and submodular functions. A byproduct of our results here is new tight worst-case approximation guarantees for local search in p-extendible systems, which surprisingly were not known previously. The tight approximation guarantees are p^2 for additive functions and $p^2 + 1$ for monotone submodular functions.

[2] Each additional constraint on allowable perturbations $\tilde{f}$ weakens the stability assumption, resulting in a harder problem. For example, if one only assumes (i) and (ii) and not (iii), then the problem becomes "too easy", and every α-approximation algorithm automatically recovers the optimal solution in α-stable instances. If (iii) is replaced by the stronger condition that all marginal values change by a factor in $[1, \gamma]$, the problem becomes "too hard", with no positive recovery results possible (essentially because zero marginal values in f must stay zero in $\tilde{f}$).

1.2 Further Related Work

Perturbation-stability was defined by Bilu and Linial [11] in the context of the MaxCut problem. Subsequent work on perturbation-stability includes [10, 30, 2, 8, 7, 3, 39, 32, 6]. Independently of Bilu and Linial [11], Balcan, Blum and Gupta [5] introduced the related notion of *approximation stability* in the context of clustering problems like k-means and k-median. More technically distant analogs of these stability conditions (but with similar motivation) were proposed by [1, 19, 36]; see Ben-David [9] for further discussion.

2 Preliminaries

In this section we describe the notation and definitions which we use through the rest of the paper. We start by defining the family of p-systems and the problem of submodular maximization; then we present our two protagonist algorithms and the standard (additive) stability definition.

- p-Systems [25, 26]: Suppose we are given a (finite) ground set X of m elements (this could be the set of edges in a graph) and we are also given an *independence family* $\mathcal{I} \subseteq 2^X$, a family of subsets that is downward closed; that is, $A \in \mathcal{I}$ and $B \subseteq A$ imply that $B \in \mathcal{I}$. A set A is independent iff $A \in \mathcal{I}$. For a set $Y \subseteq X$, a set J is called a *base* of Y, if J is a maximal independent subset of Y; in other words $J \in \mathcal{I}$ and for each $e \in Y \setminus J$, $J + e \notin \mathcal{I}$. Note that Y may have multiple bases and that a base of Y may not be a base of a superset of Y. $(X, \mathcal{I})$ is said to be a p-system if for each $Y \subseteq X$ the following holds:

  $$\frac{\max_{J:J \text{ is a base of } Y} |J|}{\min_{J:J \text{ is a base of } Y} |J|} \leq p$$

  All set systems are assumed to be down-closed. There are some interesting special cases of p-systems [31, 12]:

  intersection of p matroids $\subseteq$ p-circuit-bounded systems $\subseteq$ p-extendible systems $\subseteq$ p-systems
- p-extendible: An independence system $(X, \mathcal{I})$ is p-extendible if the following holds: suppose we have $A \subseteq B, A, B \in \mathcal{I}$ and $A + e \in \mathcal{I}$; then there should exist a set $Z \subseteq B \setminus A$ such that $|Z| \leq p$ and $B \setminus Z + e \in \mathcal{I}$. We note here that p-extendible systems make sense only for integer values of p, whereas p-systems can have p being fractional and that 1-systems as well as 1-extendible systems are exactly matroids. It is a family of independence systems containing many important and seemingly unrelated problems like welfare maximization, k-dimensional Matching, Asymmetric Travelling Salesman Problem, weighted Δ-Independent Set (Δ: maximum degree) and others [31].
- Submodular Maximization: A set function $f : 2^X \to \mathbb{R}^+ \cup \{0\}$ is submodular if for every $A, B \subseteq X$, we have $f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$. Given a p-system $(X, \mathcal{I})$ and a monotone submodular function f, we are interested in the problem of maximizing $f(S)$ over the independent sets $S \in \mathcal{I}$; in other words we wish to find $\max_{S \in \mathcal{I}} f(S)$. If f is additive, we can associate a weight w_e with each element $e \in X$ and we want to find $\max_{S \in \mathcal{I}} w(S)$, where $w(S) = \sum_{e \in S} w_e$.
- Greedy algorithm: It starts with $S = \emptyset$ and greedily picks elements of X that will increase its objective value by the most, while remaining feasible i.e. picks $e^* = \arg\max_{e \in X, S+e \in \mathcal{I}} (f(S + e) - f(S))$. It is a well-known fact [25, 34], that for any p-system, the standard greedy algorithm is a $(p+1)$-approximation (if f is additive, Greedy is a p-approximation).

- (p, q)-Local Search: It starts from a feasible solution and at each iteration seeks for an improving move. In particular, starting from any $S \in \mathcal{I}$, it tries to find a *better* $S' \in \mathcal{I}$ with: $|S \setminus S'| \leq p$, $|S' \setminus S| \leq q$ and $f(S') > f(S)$. If it finds such a feasible solution S', it switches to S' and repeats. It stops when no improving move can be made. Note that the stopping condition and its performance depend on the size of the (p, q)-*neighbourhood* used. We note that $(p, 1)$-local search is necessary for p-extendible systems. For recent improvements on Local Search performance in the case of matroids, we refer the reader to [26].
- Stable instances: Stability can be defined in general for instances of weighted optimization problems [11], where the objective function w is additive. In our case, given a p-system and an *additive* function w we wish to maximize over the p-system, we call the instance γ-stable, if the optimal solution $S^* \in \mathcal{I}$ remains the unique optimum, even after assigning a new weight $\tilde{w}_e$ to an element e such that $w_e \leq \tilde{w}_e \leq \gamma \cdot w_e$. In an extreme case, we can keep the weights of the elements in optimum the same and increase all others by a factor of γ; the optimum should remain the same. Sometimes, we say that we γ-perturb the input when we multiply some weights by at most γ. We will see in Section 4 how to extend this additive stability definition to stability for submodular functions.

3 Warm-up: Additive Case and Greedy Recovery

In this section, as a warm-up, we deal with additive functions, proving the first positive recovery result for the greedy algorithm and showing that it is tight.

3.1 Exact Recovery for p-extendible, p-stable systems

We are given an independence set system $(X, \mathcal{I}, w)$ and we want to find an independent solution $S^* \in \mathcal{I}$ with maximum weight, where for $I \in \mathcal{I} : w(I) = \sum_{e \in I} w(e)$. We are interested in the performance of the standard greedy algorithm and we can prove the following:

▶ **Theorem 2.** *Given an instance of a p-extendible independence system $(X, \mathcal{I}, w)$, that has a p-stable optimal solution $S^* = \arg\max_{I \in \mathcal{I}} w(I)$, the Greedy algorithm exactly recovers S^*.*

Proof. From the definition of p-extendibility we know that for the system $\mathcal{I}$, the following holds: suppose $A \subseteq B, A, B \in \mathcal{I}$ and $A + e \in \mathcal{I}$, then there is a set $Z \subseteq B \setminus A$ such that $|Z| \leq p$ and $B \setminus Z + e \in \mathcal{I}$. The Greedy starts from the empty set and greedily picks elements with maximum weight subject to being feasible; it finally outputs S which is a maximal solution, i.e. $S \cup \{e\} \notin \mathcal{I}, \forall e \in X \setminus S$. In order to get exact recovery, we want to show that $S \equiv S^*$.

Let's suppose $S \setminus S^* \neq \emptyset$. Then, out of all the elements of $S \setminus S^*$ that the Greedy selected, let's focus on the first element $e_1 \in S \setminus S^*$. Let $S_{\{e_1\}}$ denote the greedy solution right before it picked element e_1. Note that before choosing e_1, greedy $S_{\{e_1\}}$ was in agreement with the optimal solution, i.e. $S_{\{e_1\}} \subseteq S^*$. Since $e_1 \notin S^*$, we can use the p-extendibility, where we specify $A = S_{\{e_1\}}, B = S^*, e = e_1$ ($A + e \equiv S_{\{e_1\}} + e_1 \in \mathcal{I}$, since Greedy is always feasible) and we get, following the above definition, that there exists set of elements $Z \subseteq S^* \setminus A \equiv S^* \setminus S_{\{e_1\}}$, with $|Z| \leq p$ and $(S^* \setminus Z) \cup \{e_1\} \in \mathcal{I}$. This intuitively means that the element e_1 has conflicts with the elements in $Z \subseteq S^* \setminus S_{\{e_1\}}$, but if we remove at most $|Z| \leq p$ elements from $S^* \setminus S_{\{e_1\}}$, we get no conflicts and thus an independent (feasible) solution according to the system $\mathcal{I}$.

We call this solution J, i.e. $J = (S^* \setminus Z) \cup \{e_1\} \in \mathcal{I}$ (note $J \neq S^*$) and we will show that we can perturb the instance (new weight function $\tilde{w}$) no more than a factor of p, so that J's weight is at least that of the optimal, i.e. $\tilde{w}(J) \geq \tilde{w}(S^*)$, which would be a contradiction to the p-stability of the given instance. All we have to do is perturb the instance by multiplying the weight of the element e_1 by p. By the greedy criterion for picking elements (note that all elements of Z were available to Greedy at the point it chose e_1) and the fact that $|Z| \leq p$ we get:

$$\forall e \in Z \subseteq \left(S^* \setminus S_{\{e_1\}}\right) : w(e_1) \geq w(e) \implies p \cdot w(e_1) \geq \sum_{e \in Z} w(e) = w(Z) \tag{1}$$

which implies that the weight of the set J is actually no less than the weight of S^* in the aforementioned perturbed instance (weight function $\tilde{w}$). Indeed:

$$\tilde{w}(J) = \tilde{w}((S^* \setminus Z) \cup e_1) = \tilde{w}(S^* \setminus Z) + \tilde{w}(e_1) = w(S^*) - w(Z) + p \cdot w(e_1) \geq w(S^*) = \tilde{w}(S^*)$$

where for the last inequality we used (1). This is a contradiction because it violates the p-stability property (the optimal solution should stand out as the unique optimum for any p-perturbation) and thus we conclude that $S \setminus S^* = \emptyset$. Since Greedy outputs a maximal solution, we conclude that S coincides with S^* and so Greedy exactly recovers the optimal solution. ◀

We next show that our result is tight both in terms of the stability factor and the generality of p-extendible systems.

▶ **Proposition 3.** *There exist p-extendible systems with a $(p - \epsilon)$-stable optimal solution S^*, for which the Greedy fails to recover it.*

Proof. Take a Maximum Weight Matching instance (here $p = 2$): a path of length 3 with weights $(1, 1 + \epsilon', 1)$. The Greedy fails to recover the optimal solution S^*, since it picks the $(1 + \epsilon')$ edge whereas it should have picked both the other edges. For the right choice of ϵ' ($\epsilon' < \frac{\epsilon}{2-\epsilon}$), we can make the instance arbitrarily close to $(p - \epsilon) = (2 - \epsilon)$ stable. Observe that we can give such examples for any value of p (consider the p-dimensional Matching problem) and that the example can be made arbitrarily large just by repeating it. ◀

▶ **Proposition 4.** *There are p-systems whose optimal solution S^* is M-stable (for arbitrary $M > 1$) and for which the greedy algorithm fails to recover it.*

Proof. The example is based on a knapsack constraint. Fix $M' > 1$ and let the size of the knapsack $B = M' + 1$. We will have elements of type A ($|A| = M'$), a special element e^* and elements of type C ($|C| = M'$). The pair (value, size) for elements in A, C is respectively: $(2, 1), (1, \frac{1}{M'})$ and for $e^* : (1 + \epsilon, 1), \epsilon > 0$. Note that the optimal solution S^* is $A \cup C$ with total value $2M' + M' = 3M'$ and size $M' + M' \cdot \frac{1}{M'} = M' + 1$ (fits in the knapsack). However, Greedy will pick $A \cup \{e^*\}$ for a total value of $2M' + 1 + \epsilon$ and size $M' + 1$. Note that this is a p-system for a value of $p < 2$ since any feasible solution S can be extended to a solution S' with $|S'| \geq M' + 1$ and the largest feasible solution has $2M'$ elements (there are only $2M' + 1$ elements in total). However, this is not a 2-extendible system (it is actually an M'-extendible system) and we see that even if it is $(M' - 1)$-stable, Greedy still fails to recover the optimal solution S^*. To see why it is $(M' - 1)$-stable, note that the only γ-perturbation (perturbations are allowed only on the values, not the sizes) we can make to favour the greedy solution is to the element e^*, thus we would need $\gamma(1 + \epsilon) \geq M' \implies \gamma > M' - 1$ (ϵ is small). Choose $M' = M + 1$ and this concludes the proof. We also note that a variation of this counterexample would trick as well the (more natural) Greedy that sorts the elements according to value density ($\frac{v_i}{s_i}$) instead of just their value. ◀

We find Proposition 4 surprising, given that the greedy algorithm is a good worst-case approximation algorithm for such problems. The above "bad" example leads us to the definition of *hereditary* systems; it turns out that this is another characterization of the p-extendible systems. Due to space constraints, we give the formal definition in the full version (in Appendix B).

4 The Case of Submodular Functions

This section considers recovery results for stable instances where the objective function is monotone and submodular. Submodular functions are widely used in many areas ranging from mathematics to economics, and they model situations with *diminishing returns*. Famous examples include influence maximization [24, 33, 20, 13] and welfare maximization in auctions and game theory [27, 35]. For example, in influence maximization, the goal is to "activate" a subset of the participants in a social network (e.g., provide with information, or a promotional product) so as to maximize the expected spread of the idea or product. The diffusion of information is usually modeled with submodular functions (indicating the probability that a node adopts a new idea or product as a function of how many of her neighbors in the social network have already done so). In practice, the submodular functions in the input are estimated from data and hence are noisy (e.g. [4]). One hopes that the output of an influence maximization algorithm (which is typically a greedy algorithm [24]) is robust to modest errors in the specification of the submodular function. This section proposes a definition to make this idea precise, and proves tight results for greedy and local search algorithms under this stability notion.

4.1 Stability for submodular functions

All previous work on perturbation-stability considered only additive objective functions. We next state our extension to submodular functions.

▶ **Definition 5** (γ-perturbation, $\gamma \geq 1$)**.** Given a monotone submodular function $f : 2^X \to \mathbb{R}^+ \cup \{0\}$, we define $f_S(j) = f(S + j) - f(S)$. A γ-perturbation of f is any function $\tilde{f}$ such that the following three properties hold:

1. $\tilde{f}$ is monotone and submodular.
2. $f \leq \tilde{f} \leq \gamma f$, or in other words $f(S) \leq \tilde{f}(S) \leq \gamma f(S)$ for all $S \subseteq X$.
3. For all $S \subseteq X$ and $j \in X \setminus S$, $0 \leq \tilde{f}_S(j) - f_S(j) \leq (\gamma - 1) \cdot f(\{j\})$.

The definition of a γ-stable instance is then defined as usual.

▶ **Definition 6** (γ-stability)**.** Given an independence system $(X, \mathcal{I})$ and a monotone submodular function $f : 2^X \to \mathbb{R}^+ \cup \{0\}$, let $S^* := \arg\max_{S \in \mathcal{I}} f(S)$. The instance is *$\gamma$-stable* if for every γ-perturbation of the initial function f, S^* remains the unique optimal solution.

As discussed in the Introduction, while Definition 5 is perhaps not the first one that comes to mind, it appears to be the "sweet spot". Natural modifications[3] of the definition are generally either too restrictive (rendering the problem impossible, e.g. if property 3 is replaced with

[3] If we dropped Property 3, then *any* c-approximation algorithm ($c \geq 1$) returning a solution S with $f(S) \geq \frac{1}{c} \cdot f(S^*)$, could be made to have value equal with S^* in the c-perturbed version $\tilde{f}(S) = cf(S) \geq f(S^*) = \tilde{f}(S^*)$. If we dropped Property 2, then the definition would not be a generalization for the case of additive perturbations as we could have $\tilde{f}(S) > \gamma f(S)$ for some set S, because of the quantity $f(\{j\})$ in Property 3, which is relative to the *empty set* and may be large compared to $f_S(j)$.

relative perturbations since then the zero marginal values in f must stay zero in $\tilde{f}$) or too permissive (rendering the problem uninteresting, with all α-approximation algorithms equally good).

▶ **Proposition 7.** *Definition 5 specializes to perturbation-stability in the special case of an additive objective function.*

Proof. This follows easily since if the function f is additive, then there will be no dependence of the element's j marginal value on the current set S and thus property 3 from the above **γ-perturbation** definition just becomes:

$$0 \leq \tilde{f}_S(j) - f_S(j) \leq (\gamma - 1) \cdot f(j) \iff 0 \leq \tilde{f}(j) - f(j) \leq (\gamma - 1) \cdot f(j) \iff f(j) \leq \tilde{f}(j) \leq \gamma \cdot f(j)$$

which is exactly the standard notion of stability introduced by [11]. Note that this also implies the first condition for all sets S: $f(S) \leq \tilde{f}(S) \leq \gamma f(S)$, by the additivity of f. ◀

We now prove a useful proposition that we will often use when proving recovery results for submodular maximization. Informally, we show that multiplying the marginal improvements of the choices made by an algorithm by γ is a valid γ-perturbation.

▶ **Proposition 8.** *Let f be a monotone submodular function. Fix an ordered sequence of elements $e_1, e_2, \ldots, e_k$, and let $\delta_i = f(\{e_1, \ldots, e_i\}) - f(\{e_1, \ldots, e_{i-1}\})$. Then $\tilde{f}$ defined by $\tilde{f}(S) := f(S) + (\gamma - 1) \sum_{i:e_i \in S} \delta_i$ is a valid γ-perturbation of f.*

Proof. Let us verify the conditions of a γ-perturbation.

First, $\tilde{f}$ is monotone submodular, since it is a sum of a monotone submodular and a monotone additive function ($\delta_i \geq 0$ by monotonicity).

Second, we have $f(S) \leq \tilde{f}(S) = f(S) + (\gamma - 1) \sum_{i:e_i \in S} \delta_i \leq f(S) + (\gamma - 1) f(S \cap \{e_1, \ldots, e_k\})$ by submodularity, and by monotonicity this is at most $\gamma f(S)$.

Third, the marginal values of $\tilde{f}$ are $\tilde{f}_S(e_i) = f_S(e_i) + (\gamma - 1)\delta_i \leq f_S(e_i) + (\gamma - 1) f(\{e_i\})$ (and unchanged for elements other than the e_i). ◀

4.2 Greedy recovery and submodularity

The main result here is that the standard greedy algorithm can recover the optimal solution of a p-extendible system, if the optimal solution is $(p + 1)$-stable (as it was defined in Section 4.1).

▶ **Theorem 9** (Greedy Recovery)**.** *Given a monotone submodular function f to maximize over a p-extendible system $(X, \mathcal{I})$, if the optimal solution $S^* = \arg\max_{S \in \mathcal{I}} f(S)$ is $(p + 1)$-stable, then the greedy algorithm recovers S^* exactly.*

Proof. The proof generalizes the argument we used in the additive case so that we handle submodularity and the proving strategy resembles the proof of the approximation guarantee for the greedy algorithm for submodular maximization on p-extendible systems [12]. Let's denote by $S = \{e_1, \ldots, e_k\}$ the solution produced by Greedy (in the order that Greedy picked them) and S^* the optimal solution. To give some intuition, in the additive case before, we used the property of p-extendibility in order to say that every element that appears in S but not in S^* could be "boosted" by a factor of p to obtain an even better optimal solution, which would be a contradiction, because of the p-stability. Now, due to submodularity, we need to be careful that we make this exchange argument in a cautious manner.

For $0 \leq i \leq k$, let $S_i = \{e_1, e_2, \ldots, e_i\}$ denote the first i elements picked by Greedy (with $S_0 = \emptyset$). Let $\delta_i = f_{S_{i-1}}(e_i) = f(S_i) - f(S_{i-1})$. Using the p-extendibility property, we can find a chain of sets $S^* = T_0 \supseteq T_1 \supseteq \cdots \supseteq T_k = \emptyset$ such that for $1 \leq i \leq k$:

$$S_i \cup T_i \in \mathcal{I},\ S_i \cap T_i = \emptyset \text{ and } |T_{i-1} \setminus T_i| \leq p.$$

The above means that every element in T_i is a candidate for Greedy in step $i+1$. We construct the chain as follows: Let $T_0 = S^*$; we show how to construct T_i from T_{i-1}:

1. If $e_i \in T_{i-1}$, we define $S_i^* = \{e_i\}$ and $T_i = T_{i-1} - e_i$. This corresponds to the trivial case when Greedy, at stage i, happens to choose an element e_i that also belongs to the optimal solution S^*.
2. Otherwise ($e_i \notin T_{i-1}$), we let S_i^* be a smallest subset of T_{i-1} such that $(S_{i-1} \cup T_{i-1}) \setminus S_i^* + e_i$ is independent and since $\mathcal{I}$ is p-extendible, we have $|S_i^*| \leq p$. We let $T_i = T_{i-1} \setminus S_i^*$.

By the above definitions for S_i, T_i, S_i^* it follows that $S_i \cup T_i \in \mathcal{I}$ and $S_i \cap T_i = \emptyset$. By the maximality of Greedy (stopping condition: $\{e | S_k + e \in \mathcal{I}\} = \emptyset$) and the fact that $S_k \cup T_k \in \mathcal{I}$, it also follows that $T_k = \emptyset$. Since Greedy could have picked, instead of e_i, any of the elements in S_i^* (in fact T_{i-1}) we get: $\delta_i \geq \frac{1}{p} f_{S_{i-1}}(S_i^*)$ (recall that $|S_i^*| \leq p$).

Let us assume now that the Greedy solution S is not optimal. We use Proposition 8 to define a $(p+1)$-perturbation that produces a new optimal solution. Let's suppose $|S \setminus S^*| = l$ and let's rename the elements e_i such that $|S \setminus S^*| = \{e_1, e_2, \ldots, e_l\}$ in the order that the Greedy picked the elements. Then we define $\tilde{f}(T)$ for every T by $f(T) = f(T) + p \sum_{1 \leq i \leq l : e_i \in T} \delta_i$, where $\delta_i = f_{S_i}(e_i) = f(\{e_1, \ldots, e_i\}) - f(\{e_1, \ldots, e_{i-1}\})$. Using Proposition 8, this is a valid $(p+1)$-perturbation. For the greedy solution S, we obtain:

$$\tilde{f}(S) = f(S) + p \sum_{i=0}^{l-1} f_{S_i}(e_{i+1}) \geq$$

$$\geq f(S) + \sum_{i=0}^{l-1} f_{S_i}(S_{i+1}^*) \geq f(S) + \sum_{i=0}^{l-1} f_S(S_{i+1}^*) \geq f(S) + f_S(S^* \setminus S) =$$

$$= f(S) + (f((S^* \setminus S) \cup S) - f(S)) = f(S^* \cup S) \geq f(S^*) = \tilde{f}(S^*).$$

We ended up with $\tilde{f}(S) \geq \tilde{f}(S^*)$ which means that S^* is no longer the unique optimum and hence we get a contradiction to the $(p+1)$-stability of S^*. ◀

▶ Remark. If instead of exact access to the values of the function f, we had an α-approximate oracle, then the proof easily extends to handle this case as well. In particular, suppose each element e_i picked by Greedy at stage i satisfies $f_{S_{i-1}}(e_i) \geq \alpha \max_{e \in A_i} f_{S_{i-1}}(e)$, where A_i is the set of all candidate augmentations of S_{i-1}. Here $\alpha \leq 1$. We would then have that the greedy marginal improvement $\delta_i \geq \frac{\alpha}{p} f_{S_{i-1}}(S_i^*)$ and thus we would need $\gamma - 1 = \frac{p}{\alpha}$ leading to exact recovery of $\left(\frac{p+\alpha}{\alpha}\right)$-stable instances ($\alpha \leq 1$).

4.3 Welfare Maximization

In many situations, like the welfare maximization problem [27, 35, 41], the submodular function f we wish to maximize has a special form, e.g. it may be written as a sum of other submodular functions f_i (each of which may correspond to the player's i valuation on different allocations of the items). In this special case, we have $f(S) = \sum_{i=1}^{n} f_i(S)$ and from Theorem 9 Greedy recovers the optimal solution S^* for the case of matroids, which are 1-extendible, if S^* is 2-stable.

However, for *sum* functions $f = \sum_i f_i$, we may as well hope that a stronger recovery result is true, i.e. that Greedy recovers the optimal solution of $\max\{f(S) = \sum_i f_i : S \in \mathcal{I}\}$, where the optimum is 2-stable only with respect to 2-perturbations of the individual functions f_i. This is indeed true (for the proof, we refer the reader to Appendix A of the full version).

▶ **Theorem 10.** *Let $(X, \mathcal{I})$ be a matroid on the elements of X, let $B_1, B_2, \ldots, B_k$ be a partition of X, $f_i : 2^{B_i} \to \mathbb{R}^+ \cup \{0\}$, for $i \in \{1, 2, \ldots, k\}$ be monotone submodular and let $f = \sum_{i=1}^{k} f_i$. Suppose the optimal solution S^* of $\max\{f(S) : S \in \mathcal{I}\}$ is 2-stable only with respect to individual perturbations of the functions f_i. Then, Greedy recovers S^*.*

5 Local Search Performance

In this section we discuss *local search* [28] (described in Section 2). Local search often gives better results than Greedy, at the cost of a slower running time – for example for submodular maximization subject to the intersection of k matroids [26, 21], and for k-set packing [40, 18, 23]. For some interesting recent results about local search in *beyond-worst-case* settings and on geometric optimization we refer the reader to [14, 15, 16].

Somewhat surprisingly, it was not known (to our knowledge) how local search performs for p-systems and p-extendible systems. (We recall that the greedy algorithm gives a factor of $1/p$ for maximization of an additive function and $1/(p+1)$ for maximization of a monotone submodular function under these constraints.) Here, we prove that local search in fact performs worse than Greedy for these constraints. Although it gives a $1/p$-approximation for cardinality maximization under a p-system constraint (essentially by definition), it does not give any bounded approximation factor for additive function maximization under a p-system, and only a $1/p^2$-approximation under a p-extendible system.

5.1 Local search fails for p-systems

We construct simple examples where local search will not recover any fraction of the maximum-weight solution for p-systems (even if it is arbitrarily stable, $p = 2$, and even if we allow large exchange neighborhoods). In particular, consider a ground set $X = A \cup \{e^*\}$ where $|A| = n$. The independent sets of $\mathcal{I}$ are:

- any subset of A, or
- e^* plus any subset of at most $n/2$ elements of A.

Note that this is a 2-system, because for $S \subseteq X$, any independent subset of S can be extended to an independent set of size at least $\min\{|S|, n/2\}$, and the maximum independent subset of S has size at most $\min\{|S|, n\}$. The weights could be 0 on A, and 1 on the special element e^*. So the optimum is $w(e^*) = 1$ (observe that the optimal solution is c-stable for arbitrarily large c). However, A is a local optimum, unless we are willing to swap out $n/2$ elements, which is not possible for efficient local search.

5.2 Lower bound for p-extendible systems

Let us consider the following instance. Let $X = A \cup B$ where A, B are disjoint sets. We define $\mathcal{I} \subseteq 2^X$ as follows: $S \in \mathcal{I}$ iff

- $|S \cap A| + p|S \cap B| \leq |A|$, or
- $p|S \cap A| + |S \cap B| \leq |B|$.

▶ **Lemma 11.** *For any A, B disjoint, the above is a p-extendible system.*

Proof. Let $S \subseteq T$ and $i \in X \setminus T$ be such that $S + i \in \mathcal{I}$ and $T \in \mathcal{I}$. We need to prove that there is $Z \subseteq T \setminus S, |Z| \leq p$ such that $(T \setminus Z) + i \in \mathcal{I}$.

We can assume that $|T \setminus S| > p$, because otherwise we can set $Z = T \setminus S$ and obviously $(T \setminus Z) + i = S + i \in \mathcal{I}$. Assuming $|T \setminus S| > p$, let Z be an arbitrary set of p elements from $T \setminus S$. We consider 2 cases: If $|T \cap A| + p|T \cap B| \leq |A|$, then $|(T \setminus Z) \cap A| + p|(T \setminus Z) \cap B| \leq |A| - p$. Adding the element i can increase the left-hand side by at most p, and so $|(T \setminus Z + i) \cap A| + p|(T \setminus Z + i) \cap B| \leq |A|$. Similarly, in the second case, if $p|T \cap A| + |T \cap B| \leq |B|$, then $p|(T \setminus Z) \cap A| + |(T \setminus Z) \cap B| \leq |B| - p$. Adding the element i can increase the left-hand side by at most p, and so $p|(T \setminus Z + i) \cap A| + |(T \setminus Z + i) \cap B| \leq |B|$. ◀

Now we choose the cardinalities of A and B and the weights of their elements appropriately to get a negative result.

▶ **Lemma 12.** *For $\epsilon > 0$, let $|A| = n$ and $|B| = (p - \epsilon)n$, and set the weights as $w_a = 1$ for $a \in A$ and $w_b = p - \epsilon$ for $b \in B$. Then A is a local optimum of value $w(A) = w(B)/(p - \epsilon)^2$, unless the local search explores exchanges of size at least $\frac{\epsilon}{p}n$.*

Proof. Both A and B are independent sets. Note that for any $i \in B$, we need to remove $Z \subseteq A$ of cardinality at least $|Z| = p$ to obtain $S = (A \setminus Z) + i$ satisfying $|S \cap A| + p|S \cap B| \leq |A|$. More generally, for $Y \subseteq B$, we need to remove $Z \subseteq A, |Z| = p|Y|$ to obtain $S = (A \setminus Z) \cup Y$ that satisfies $|S \cap A| + p|S \cap B| \leq |A|$. Possibly, we could satisfy the second condition, $p|S \cap A| + |S \cap B| \leq |B|$, but this will not happen unless $|A \setminus Z| = |S \cap A| \leq |B|/p = (1 - \frac{\epsilon}{p})n$. Therefore, we would need to remove Z of cardinality at least $\frac{\epsilon}{p}n$.

If the swaps considered are smaller than $\frac{\epsilon}{p}n$ then A is a local optimum because adding $Y \subseteq B$ and removing $Z \subseteq A, |Z| = p|Y|$ results in a solution of lower weight. In conclusion, A is a local optimum of value $w(A) = n$, while the optimum is $OPT = w(B) = (p - \epsilon)^2 n$. ◀

5.3 Upper bound for p-extendible systems

Here we prove that local search does in fact provide a $1/p^2$-approximation for weighted maximization under a p-extendible system. More generally, we prove (here, we will ignore the technicalities of stopping the local search in polynomial time as this can be handled using standard techniques, while losing $1/poly(n)$ in the approximation factor) the following:

▶ **Theorem 13.** *For any p-extendible system $\mathcal{I} \subseteq 2^X$ and a monotone submodular function $f : 2^X \to \mathbb{R}_+$, local search with $(p, 1)$-swaps (including at most 1 element and removing at most p elements) provides a $1/(p^2 + 1)$-approximation. For additive f, the factor is $1/p^2$.*

Proof. Let A be a local optimum under $(p, 1)$-swaps, and let B be an optimal solution. (For convenience, let us also assume that we always try to add elements to A if possible, even if they bring zero marginal value.) We proceed in two steps, the first one inspired by the analysis of the greedy algorithm for p-extendible systems [12] and the second one similar to other analyses of local search.

Let $A = \{a_1, \ldots, a_k\}$ be a greedy ordering of A in the sense that a_1 is the element of A maximizing $f_\emptyset(a_1)$; given a_1, a_2 is the element of $A - a_1$ maximizing $f_{\{a_1\}}(a_2)$, a_3 is the element of $A - a_1 - a_2$ maximizing $f_{\{a_1, a_2\}}(a_3)$, etc. Using the p-extendible property, there is a subset $B_1 \subseteq B, |B_1| \leq p$ such that $(B \setminus B_1) + a_1 \in \mathcal{I}$. Further, since $\{a_1, a_2\} \in \mathcal{I}$, there is a subset $B_2 \subseteq B \setminus B_1, |B_2| \leq p$ such that $(B \setminus (B_1 \cup B_2)) \cup \{a_1, a_2\} \in \mathcal{I}$, etc. Generally, there are disjoint subsets $B_1, \ldots, B_k \subseteq B, |B_i| \leq p$ such that $(B \setminus (B_1 \cup \ldots B_i)) \cup \{a_1, \ldots, a_i\} \in \mathcal{I}$. In fact, if $|A| = k$, the sets $B_1, \ldots, B_k$ form a partition of B. Otherwise there would be additional elements in $B \setminus (B_1 \cup \ldots \cup B_k)$ which can be added to A, which would contradict the local optimality of A.

Now, we claim that for each $b \in B_i$, we have $f_A(b) \leq pf_{\{a_1,\ldots,a_{i-1}\}}(a_i)$. If not, we would be able to add b and, since $\{a_1,\ldots,a_{i-1},b\} \in \mathcal{I}$, we could remove at most p elements $Z \subseteq A \setminus \{a_1,\ldots,a_{i-1}\}$ so that $(A \setminus Z) + b \in \mathcal{I}$. By submodularity and the greedy ordering, we would have $f(A \setminus Z) \geq f(A) - pf_{\{a_1,\ldots,a_{i-1}\}}(a_i)$ and again by submodularity, we would have $f((A \setminus Z) + b) \geq f(A \setminus Z) + f_A(b) > f(A \setminus Z) + pf_{\{a_1,\ldots,a_{i-1}\}}(a_i) \geq f(A)$. Therefore, this would be an improving local swap.

Since A is a local optimum, we conclude that $f_A(b) \leq pf_{\{a_1,\ldots,a_{i-1}\}}(a_i)$ for each $b \in B_i$. Since $B = B_1 \cup \ldots \cup B_k$ and $|B_i| \leq p$, we have by submodularity

$$f_A(B) \leq \sum_{i=1}^{k} \sum_{b \in B_i} f_A(b) \leq \sum_{i=1}^{k} |B_i| pf_{\{a_1,\ldots,a_{i-1}\}}(a_i) \leq p^2 \sum_{i=1}^{k} f_{\{a_1,\ldots,a_{i-1}\}}(a_i) \leq p^2 f(A)$$

For f monotone submodular, we have $f(B) \leq f(A) + f_A(B) \leq (p^2+1)f(A)$. For f additive, we have $f(B) = f_A(B) \leq p^2 f(A)$. This completes the proof. ◀

5.4 Recovery for p-extendible systems

▶ **Theorem 14.** *Given a p-extendible system $\mathcal{I} \subseteq 2^X$ and a monotone submodular function $f : 2^X \to \mathbb{R}_+ \cup \{0\}$ we wish to maximize, if the optimal solution B is (p^2+1)-stable, then local search with $(p,1)$-swaps exactly recovers it. If f is additive, recovery holds if B is p^2-stable.*

Proof. The basic idea is that we can contract the elements that belong to $A \cap B$ and then use the same charging argument from above. Using the notation from the proof of Theorem 13, for elements $a_i \in A \cap B$ the corresponding B_i block is just $\{a_i\}$. Now we can rename elements in $A \setminus B = \{a_1,\ldots,a_m\}$ with corresponding blocks $B_1,\ldots,B_m$ such that $B \setminus A = B_1 \cup \ldots \cup B_m$ and $|B_i| \leq p$. Rewriting the local search guarantee:

$$f_A(B \setminus A) \leq \sum_{i=1}^{m} \sum_{b \in B_i} f_A(b) \leq \sum_{i=1}^{m} |B_i| pf_{\{a_1,\ldots,a_{i-1}\}}(a_i) \leq p^2 \sum_{i=1}^{m} f_{\{a_1,\ldots,a_{i-1}\}}(a_i) \leq p^2 f(A \setminus B)$$

Since $f_A(B \setminus A) = f(B \cup A) - f(A) \geq f(B) - f(A)$, we can (p^2+1)-perturb the input (only the marginal of elements in $A \setminus B$) and get: $\tilde{f}(B) = f(B) \leq f(A) + p^2 f(A \setminus B) = \tilde{f}(A)$, hence contradicting the (p^2+1)-stability. In the case of additive f, $f_A(B \setminus A) = f(B \setminus A)$ and $\tilde{f}(B) = f(B) = f(B \setminus A) + f(B \cap A) \leq p^2 f(A \setminus B) + f(B \cap A) \leq f(A) + (p^2-1)f(A \setminus B) = \tilde{f}(A)$, where we p^2-perturbed the instance, hence contradicting the p^2-stability of the instance. ◀

5.5 Recovery for the intersection of Matroids

If the independence system $\mathcal{I}$ is the intersection of p matroids: $\mathcal{I} = \cap_{i=1}^{p} \mathcal{I}_i$, local search with $(p,1)$-swaps recovers $(p+1)$-stable optimal solutions (for proof, see full version of the paper).

▶ **Theorem 15.** *Given $(X, \mathcal{I})$, with $\mathcal{I} = \cap_{i=1}^{p} \mathcal{I}_i$ where each $\mathcal{I}_i$ is a matroid and f monotone submodular, such that the optimal solution is $(p+1)$-stable, Local Search exactly recovers it.*

Acknowledgements. The authors would also like to thank the anonymous reviewers for their useful comments.

References

1 Margareta Ackerman and Shai Ben-David. Clusterability: A theoretical study. In *Artificial Intelligence and Statistics*, pages 1–8, 2009.

2 Haris Angelidakis, Konstantin Makarychev, and Yury Makarychev. Algorithms for stable and perturbation-resilient problems. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 438–451, 2017.

3 Pranjal Awasthi, Avrim Blum, and Or Sheffet. Center-based clustering under perturbation stability. *Information Processing Letters*, 112(1):49–54, 2012.

4 Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54. ACM, 2006.

5 Maria-Florina Balcan, Avrim Blum, and Anupam Gupta. Approximate clustering without the approximation. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1068–1077. Society for Industrial and Applied Mathematics, 2009.

6 Maria-Florina Balcan and Mark Braverman. Nash equilibria in perturbation resilient games. *arXiv preprint arXiv:1008.1827*, 2010.

7 Maria-Florina Balcan, Nika Haghtalab, and Colin White. k-center clustering under perturbation resilience. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 68:1–68:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.68.

8 Maria Florina Balcan and Yingyu Liang. Clustering under perturbation resilience. In *International Colloquium on Automata, Languages, and Programming*, pages 63–74. Springer, 2012.

9 Shai Ben-David. Computational feasibility of clustering under clusterability assumptions. *arXiv preprint arXiv:1501.00437*, 2015.

10 Yonatan Bilu, Amit Daniely, Nati Linial, and Michael Saks. On the practically interesting instances of maxcut. *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 526–537, 2013.

11 Yonatan Bilu and Nathan Linial. Are stable instances easy? *Combinatorics, Probability and Computing*, 21(05):643–660, 2012.

12 Gruia Calinescu, Chandra Chekuri, Martin Pal, and Jan Vondrak. Maximizing a monotone submodular function subject to a matroid constraint. *SIAM Journal on Computing*, 40(6):1740–1766, 2011.

13 Wei Chen, Yajun Wang, and Siyu Yang. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 199–208. ACM, 2009.

14 Vincent Cohen-Addad, Philip N Klein, and Claire Mathieu. Local search yields approximation schemes for k-means and k-median in euclidean and minor-free metrics. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 353–364. IEEE, 2016.

15 Vincent Cohen-Addad and Claire Mathieu. Effectiveness of local search for geometric optimization. In *31st International Symposium on Computational Geometry (SoCG 2015)*, volume 34 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 329–343, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.SOCG.2015.329.

16 Vincent Cohen Addad and Chris Schwiegelshohn. On the local structure of stable clustering instances. *CoRR*, 2017.

17 Graham Cormode, Howard Karloff, and Anthony Wirth. Set cover algorithms for very large datasets. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 479–488. ACM, 2010.

18 Marek Cygan. Improved approximation for 3-dimensional matching via bounded pathwidth local search. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 509–518, 2013.

19 Amit Daniely, Nati Linial, and Michael Saks. Clustering is difficult only when it does not matter. *arXiv preprint arXiv:1205.4891*, 2012.

20 David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

21 Yuval Filmus and Justin Ward. The power of local search: Maximum coverage over a matroid. In *STACS'12 (29th Symposium on Theoretical Aspects of Computer Science)*, volume 14, pages 601–612. LIPIcs, 2012.

22 Marshall L Fisher, George L Nemhauser, and Laurence A Wolsey. An analysis of approximations for maximizing submodular set functions—ii. In *Polyhedral combinatorics*, pages 73–87. Springer, 1978.

23 Martin Fürer and Huiwen Yu. Approximating the k -set packing problem by local improvements. In *Combinatorial Optimization - Third International Symposium, ISCO 2014, Lisbon, Portugal, March 5-7, 2014, Revised Selected Papers*, pages 408–420, 2014.

24 David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 137–146. ACM, 2003.

25 Bernhard Korte and Dirk Hausmann. An analysis of the greedy heuristic for independence systems. *Annals of Discrete Mathematics*, 2:65–74, 1978.

26 Jon Lee, Maxim Sviridenko, and Jan Vondrák. Submodular maximization over multiple matroids via generalized exchange properties. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 244–257. Springer, 2009.

27 Benny Lehmann, Daniel Lehmann, and Noam Nisan. Combinatorial auctions with decreasing marginal utilities. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 18–28. ACM, 2001.

28 Jan Karel Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.

29 László Lovász. Submodular functions and convexity. In *Mathematical Programming The State of the Art*, pages 235–257. Springer, 1983.

30 Konstantin Makarychev, Yury Makarychev, and Aravindan Vijayaraghavan. Bilu-linial stable instances of max cut and minimum multiway cut. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'14, pages 890–906, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

31 Julián Mestre. Greedy in approximation algorithms. In *European Symposium on Algorithms*, pages 528–539. Springer, 2006.

32 Matúš Mihalák, Marcel Schöngens, Rastislav Šrámek, and Peter Widmayer. On the complexity of the metric tsp under stability considerations. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 382–393. Springer, 2011.

33 Elchanan Mossel and Sebastien Roch. On the submodularity of influence in social networks. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 128–134. ACM, 2007.

34 George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, 14(1):265–294, 1978.

35 Noam Nisan and Amir Ronen. Computationally feasible vcg mechanisms. *J. Artif. Intell. Res.(JAIR)*, 29:19–47, 2007.

36 Rafail Ostrovsky, Yuval Rabani, Leonard J Schulman, and Chaitanya Swamy. The effectiveness of lloyd-type methods for the k-means problem. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 165–176. IEEE, 2006.

37 César Rego, Dorabela Gamboa, Fred Glover, and Colin Osterman. Traveling salesman problem heuristics: leading methods, implementations and latest advances. *European Journal of Operational Research*, 211(3):427–441, 2011.

38 Joachim Reichel and Martin Skutella. Evolutionary algorithms and matroid optimization problems. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2007.

39 Lev Reyzin. Data stability in clustering: A closer look. In *International Conference on Algorithmic Learning Theory*, pages 184–198. Springer, 2012.

40 Maxim Sviridenko and Justin Ward. Large neighborhood local search for the maximum set packing problem. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*, pages 792–803, 2013.

41 Jan Vondrák. Optimal approximation for the submodular welfare problem in the value oracle model. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 67–74. ACM, 2008.

On the Complexity of Bounded Context Switching*

Peter Chini[1], Jonathan Kolberg[2], Andreas Krebs[†3], Roland Meyer[‡4], and Prakash Saivasan[5]

1 **TU Braunschweig, Germany**
p.chini@tu-bs.de
2 **TU Braunschweig, Germany**
j.kolberg@tu-bs.de
3 **Universität Tübingen, Germany**
krebs@informatik.uni-tuebingen.de
4 **TU Braunschweig, Germany**
roland.meyer@tu-bs.de
5 **TU Braunschweig, Germany**
p.saivasan@tu-bs.de

Abstract

Bounded context switching (BCS) is an under-approximate method for finding violations to safety properties in shared-memory concurrent programs. Technically, BCS is a reachability problem that is known to be NP-complete. Our contribution is a parameterized analysis of BCS.

The first result is an algorithm that solves BCS when parameterized by the number of context switches (cs) and the size of the memory (m) in $\mathcal{O}^*(m^{cs} \cdot 2^{cs})$. This is achieved by creating instances of the easier problem Shuff which we solve via fast subset convolution. We also present a lower bound for BCS of the form $m^{o(cs/\log(cs))}$, based on the exponential time hypothesis. Interestingly, the gap is closely related to a conjecture that has been open since FOCS'07. Further, we prove that BCS admits no polynomial kernel.

Next, we introduce a measure, called scheduling dimension, that captures the complexity of schedules. We study BCS parameterized by the scheduling dimension ($sdim$) and show that it can be solved in $\mathcal{O}^*((2m)^{4sdim}4^t)$, where t is the number of threads. We consider variants of the problem for which we obtain (matching) upper and lower bounds.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Shared memory concurrency, safety verification, fixed-parameter tractability, exponential time hypothesis, bounded context switching

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.27

1 Introduction

Concurrent programs where several threads interact through a shared memory can be found essentially everywhere where performance matters, in particular in critical infrastructure like operating systems and libraries. The asynchronous nature of the communication makes these

* A full version of the paper is available at https://arxiv.org/abs/1609.09728.
† Supported by the DFG Emmy Noether program (KR4042/2).
‡ A part of the work was carried out when the author was at Aalto University.

programs prone to programming errors. As a result, substantial effort has been devoted to developing automatic verification tools. The current trend for shared memory is bug-hunting: Algorithms that look for misbehavior in an under-approximation of the computations.

The most prominent method in the under-approximate verification of shared-memory concurrent programs is bounded context switching (BCS) [48]. A context switch occurs when a thread leaves the processor for another thread to be scheduled. The idea of BCS is to limit the number of times the threads may switch the processor. Effectively this limits the communication that can occur between the threads. (Note that there is no bound on the running time of each thread.) Bounded context switching has received considerable attention [37, 4, 3, 1, 38, 39, 2, 47] for at least two reasons. First, the under-approximation has been demonstrated to be useful in numerous experiments, in the sense that synchronization bugs show up in few context switches [46]. Second, compared to other verification methods, BCS is algorithmically appealing, with the complexity dropping from PSPACE to NP in the case of Boolean programs.

The hardness of verification problems, also the NP-hardness of BCS, is in sharp contrast to the success that verification tools see on industrial instances. This discrepancy between the worst-case behavior and efficiency in practice has also been observed in other areas within algorithmics. The response was a line of research that refines the classical worst-case complexity. Rather than only considering problems where the instance-size determines the running time, so-called parameterized problems identify further parameters that give information about the structure of the input or the shape of solutions. The complexity class of interest consists of the so-called fixed-parameter tractable problems. A problem is fixed-parameter tractable if the parameter that has been identified is indeed responsible for the non-polynomial running time or, phrased differently, the running time is $f(k)p(n)$ where k is the parameter, n is the size of the input, f is a computable function, and p is a polynomial.

Within fixed-parameter tractability, the recent trend is a fine-grained analysis to understand the precise functions f that are needed to solve a problem. From an algorithmic point of view, an exponential dependence on k, at best linear so that $f(k) = 2^k$, is particularly attractive. There are, however, problems where algorithms running in $2^{o(k \log(k))}$ are unlikely to exist. As common in algorithmics, unconditional lower bounds are hard to achieve, and none are known that separate 2^k and $2^{k \log(k)}$. Instead, one works with the so-called exponential time hypothesis (ETH): After decades of attempts, n-variable 3-SAT is not believed to admit an algorithm of running time $2^{o(n)}$. To derive a lower bound for a problem, one now shows a reduction from n-variable 3-SAT to the problem such that a running time in $2^{o(k \log(k))}$ means ETH breaks.

The contribution of our work is a fine-grained complexity analysis of the bounded context switching under-approximation. We propose algorithms as well as matching lower bounds in the spectrum 2^k to k^k. This work is not merely motivated by explaining why verification works in practice. Verification tasks have also been shown to be hard to parallelize. Due to the memory demand, the current trend in parallel verification is lock-free data structures [6]. So far, GPUs have not received much attention. With an algorithm of running time $2^k p(n)$, and for moderate k, say 12, one could run in parallel 4096 threads each solving a problem of polynomial effort.

When parameterized only by the context switches, BCS is quickly seen to be W[1]-hard and hence does not admit an FPT-algorithm. Since it is often the case that shared memory communication is via signaling (flags), the memory requirements are not high. We additionally parameterize by the memory. Our study can be divided into two parts.

We first give a parameterization of BCS (in the context switches and the size of the memory) that is *global* in the sense that all threads share a budget of cs many context switches. For the upper bound, we show that the problem can be solved in $\mathcal{O}^*(m^{cs}2^{cs})$. We first enumerate the sequences of memory states at which the threads could switch context, and there are m^{cs} such sequences where m is the size of the memory. For a given such sequence, we check a problem called Shuff: Do the threads have computations that justify the sequence (and lead to their accepting state)? Here, we use fast subset convolution to solve Shuff in $\mathcal{O}^*(2^{cs})$. Note that Shuff is a problem that may be interesting in its own right. It is an under-approximation that still leaves much freedom for the local computations of the threads. Indeed, related ideas have been used in testing [33, 12, 24, 31].

For the lower bound, the finding is that the global parameterization of BCS is closely related to *Subgraph Isomorphism* (SGI). Whereas the reduction is not surprising, the relationship is, with SGI being one of the problems whose fine-grained complexity is not fully understood. Subgraph isomorphism can be solved in $\mathcal{O}^*(n^k)$ where k is the number of edges in the graph that is to be embedded. The only lower bound, however, is $n^{o(k/\log k)}$, and has, to the best of our knowledge, not been improved since FOCS'07 [44, 45]. However, the belief is that the $\log k$-gap in the exponent can be closed. We show how to reduce SGI to the global version of BCS, and obtain an $m^{o(cs/\log cs)}$ lower bound. Phrased differently, BCS is harder than SGI but admits the same upper bound. So once Marx' conjecture is proven, we obtain a matching bound. If we proved a lower upper bound, we had disproven Marx' conjecture.

Our second contribution is a study of BCS where the parameterization is *local* in the sense that every thread is given a budget of context switches. Here, our focus is on the scheduling. We associate with computations so-called scheduling graphs that show how the threads take turns. We define the scheduling dimension, a measure on scheduling graphs (shown to be closely related to carving width) that captures the complexity of a schedule. Our main finding is a fixed-point algorithm that solves the local variant of BCS exponential only in the scheduling dimension and the number of threads. We study variants where only the budget of context switches is given, the graph is given, and where we assume round robin as a schedule. Verification under round robin has received quite some attention [10, 46, 40]. In that setting, we show that we get rid of the exponential dependence on the number of threads and obtain an $\mathcal{O}^*(m^{4cs})$ upper bound. We complement this by a matching lower bound.

The following table summarizes our results and highlights the main findings in gray.

Problem	Upper Bound	Lower Bound
Shuff	$\mathcal{O}^*(2^k)$	$(2-\varepsilon)^k$
BCS	$\mathcal{O}^*(m^{cs}2^{cs})$	$m^{o(cs/\log cs)}$, no poly. kernel
BCS-L-RR	$\mathcal{O}^*(m^{4cs})$	$2^{o(cs\log(m))}$
BCS-L-FIX	$\mathcal{O}^*((2m)^{4sdim})$	$2^{o(sdim\log(m))}$
BCS-L	$\mathcal{O}^*((2m)^{4sdim}4^t)$	$2^{o(sdim\log(m))}$

The organization is by expressiveness, measured in terms of the amount of computations that an analysis explores. Considering shuffle membership Shuff as an under-approximate analysis in its own right, Shuff is less expressive than the globally parameterized BCS. BCS is less expressive than round robin BCS-L-RR, which is an instance of fixing the scheduling graph BCS-L-FIX. The most liberal parameterization is via the scheduling dimension BCS-L. In the paper, we present algorithms for the case where the threads are finite state. Our results also hold for more general classes of programs, notably recursive ones. The only condition that we require is that the chosen automaton model for the threads has a polynomial time decision procedure for checking non-emptiness when intersected with a regular language.

There have been previous efforts in studying fixed-parameter tractable algorithms for automata and verification-related problems. In [21], the authors introduced the notion of conflict serializability under TSO and gave an FPT-algorithm for checking serializability. In [24], the authors studied the complexity of predicting atomicity violation on concurrent systems and showed that no FPT solution is possible for the same. In [18], various model checking problems for synchronized executions on parallel components were considered and proven to be intractable. Parameterized complexity analyses for different problems on finite automata were given in [25, 26, 50].

Verification of concurrent systems has received considerable attention. The parameterized verification was studied in [20, 22, 29, 34, 38]. Concurrent shared-memory systems with a fixed number of threads were also studied in [2, 3, 5].

2 Preliminaries

We define the bounded context switching problem of interest [48] and recall the basics on fixed-parameter tractability following [19, 27].

Bounded Context Switching. We study the safety verification problem for shared-memory concurrent programs. To obtain precise complexity results, it is common to assume both the number of threads and the data domain to be finite. Safety properties partition the states of a program into *unsafe* and *safe* states. Hence, checking safety amounts to checking whether no unsafe state is reachable. In the following, we develop a language-theoretic formulation of the reachability problem that will form the basis of our study.

We model the shared memory as a (non-deterministic) finite automaton of the form $M = (Q, \Sigma, \delta_M, q_0, q_f)$. The states Q correspond to the data domain, the set of values that the memory can be in. The initial state $q_0 \in Q$ is the value that the computation starts from. The final state $q_f \in Q$ reflects the reachability problem. The alphabet Σ models the set of operations. Operations have the effect of changing the memory valuation, formalized by the transition relation $\delta_M \subseteq Q \times \Sigma \times Q$. We generalize the transition relation to words $u \in \Sigma^*$. The set of sequences of operations that lead from a state q to another state q' is the language $L(M(q, q')) := \{u \in \Sigma^* \mid q' \in \delta_M(q, u)\}$. The language of M is $L(M) := L(M(q_0, q_f))$. The size of M, denoted $|M|$, is the number of states.

We also model the threads operating on the shared memory M as finite automata $A_{id} = (P, \Sigma \times \{id\}, \delta_A, p_0, p_f)$. Note that they use the alphabet Σ of the shared memory, indexed by the name of the thread. The index will play a role when we define the notion of context switches below. The automaton A_{id} is nothing but the control flow graph of the thread id. Its language is the set of sequences of operations that the thread could potentially execute to reach the final state. As the thread language does not take into account the effect of the operations on the shared memory, not all these sequences will be feasible. Indeed, the thread may issue a command $write(x, 1)$ followed by $read(x, 0)$, which the automaton for the shared memory will reject. The computations of A that are actually feasible on the shared memory are given by the intersection $L(M) \cap L(A_{id})$. Here, we silently assume the intersection to project away the second component of the thread alphabet.

A concurrent program consists of multiple threads A_1 to A_t that mutually influence each other by accessing the same memory M. We mimic this influence by interleaving the thread languages, formalized with the shuffle operator $\sqcup\!\sqcup$. Consider languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ over disjoint alphabets $\Sigma_1 \cap \Sigma_2 = \emptyset$. The shuffle of the languages contains all words over the union of the alphabets where the corresponding projections $(- \downarrow -)$ belong to the operand languages, $L_1 \sqcup\!\sqcup L_2 := \{u \in (\Sigma_1 \cup \Sigma_2)^* \mid u \downarrow \Sigma_i \in L_i \cup \{\varepsilon\}, i = 1, 2\}$.

With these definitions in place, a *shared-memory concurrent program (SMCP)* is a tuple $S = (\Sigma, M, (A_i)_{i\in[1..t]})$. Its language is $L(S) := L(M) \cap (\ \text{⧢}_{i\in[1..t]} L(A_i)\ )$. The safety verification problem induced by the program is to decide whether $L(S)$ is non-empty. Note that we use $[1..t]$ to identify the set $\{1, \ldots, t\}$.

We formalize the notion of context switching. Every word in the shuffle of the thread languages, $u \in \text{⧢}_{i\in[1..t]} L(A_i)$, has a unique decomposition into maximal infixes that are generated by the same thread. Formally, $u = u_1 \ldots u_{cs+1}$ so that there is a function $\varphi : [1..cs+1] \to [1..t]$ satisfying $u_i \in (\Sigma \times \{\varphi(i)\})^+$ and $\varphi(i) \neq \varphi(i+1)$ for all $i \in [1..cs]$. We refer to the u_i as contexts and to the thread changes between u_i to u_{i+1} as *context switches*. So u has $cs+1$ contexts and cs context switches. Let $\text{Context}(\Sigma, t, cs)$ denote the set of words (over Σ with t threads) that have at most cs-many context switches. The *bounded context switching* under-approximation limits the safety verification task to this language.

Bounded Context Switching (BCS)
Input: An SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ and a bound $cs \in \mathbb{N}$.
Question: Is $L(S) \cap \text{Context}(\Sigma, t, cs) \neq \emptyset$?

Fixed Parameter Tractability. BCS is NP-complete, even for unary alphabets [23]. Our goal is to understand which instances can be solved efficiently and, in turn, what makes an instance hard. Parameterized complexity addresses these questions.

A *parameterized problem* L is a subset of $\Sigma^* \times \mathbb{N}$. The problem is *fixed-parameter tractable (FPT)* if there is a deterministic algorithm that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, decides $(x, k) \in L$ in time $f(k) \cdot |x|^{O(1)}$. Here, f is a computable function that only depends on the parameter k. It is common to denote the running time by $\mathcal{O}^*(f(k))$ and suppress the polynomial part.

While many parameterizations of NP-hard problems were proven to be fixed-parameter tractable, there are problems that are unlikely to be FPT. A famous example that we shall use is k-Clique, the problem of finding a clique of size k in a given graph. k-Clique is complete for the complexity class W[1], and W[1]-hard problems are believed to lie outside FPT.

A theory of relative hardness needs an appropriate notion of reduction. Given parameterized problems $L, L' \subseteq \Sigma^* \times \mathbb{N}$, we say that L is *reducible* to L' via a *parameterized reduction*, denoted by $L \leq^{fpt} L'$, if there is an algorithm that transforms an input (x, k) to an input (x', k') in time $g(k) \cdot n^{O(1)}$ so that $(x, k) \in L$ if and only if $(x', k') \in L'$. Here, g is a computable function and k' is computed by a function only dependent on k.

For BCS, a first result is that a parameterization by the number of context switches and additionally by the number of threads, denoted by $\text{BCS}(cs, t)$, is not sufficient for FPT: The problem is W[1]-hard. It remains in W[1] if we only parameterize by the context switches.

▶ **Proposition 1.** $\text{BCS}(cs)$ *and* $\text{BCS}(cs, t)$ *are both* W[1]*-complete.*

The running time of an FPT-algorithm is dominated by f. The goal of *fine-grained complexity theory* is to give upper and lower bounds on this non-polynomial function. For lower bounds, the problem that turned out to be hard is n-variable 3-SAT. The *Exponential Time Hypothesis* (ETH) is that n-variable 3-SAT does not admit a $2^{o(n)}$-time algorithm [36]. We will prove a number of lower bounds that hold, provided ETH is true.

In the remainder of the paper, we consider parameterizations of BCS that are FPT. Our contribution is a fine-grained complexity analysis.

3 Global Parameterization

Besides the number of context switches cs, we now consider the size m of the memory as a parameter of BCS. This parameterization is practically relevant and, as we will show, algorithmically appealing. Concerning the relevance, note that communication over the shared memory is often implemented in terms of flags. Hence, when limiting the size of the memory we still explore a large part of the computations.

Upper Bounds. The idea of our algorithm is to decompose BCS into exponentially many instances of the easier problem shuffle membership (Shuff) defined below. Then we solve Shuff with fast subset convolution. To state the result, let the given instance of BCS be $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ with bound cs. To each automaton A_i, our algorithm will associate another automaton B_i of size polynomial in A_i. Let $b = \max_{i\in[1..t]} |B_i|$. Moreover, let $\mathsf{Shuff}(b, k, t) = \mathcal{O}(2^k \cdot t \cdot k \cdot (b^2 + k \cdot bc(k)))$ be the complexity of solving the shuffle problem. The factor $bc(k)$ appears as we need to multiply k-bit integers (see below). The currently best known running time is $bc(k) = k \log k \cdot 2^{\mathcal{O}(\log^* k)}$ [32, 35].

▶ **Theorem 2.** BCS *can be solved in* $\mathcal{O}(m^{cs+1} \cdot \mathsf{Shuff}(b, cs+1, t) + t \cdot m^3 \cdot b^3)$.

We decompose BCS along *interface sequences*. Such an interface sequence is a word $\sigma = (q_1, q_1') \dots (q_k, q_k')$ over pairs of states of the memory automaton M. The length is k. An interface sequence is *valid* if q_1 is the initial state of the memory automaton, q_k' the final state, and $q_i' = q_{i+1}$ for $i \in [1..k-1]$. Consider a word $u \in L(S)$ with contexts $u = u_1 \dots u_m$. An interface sequence $\sigma = (q_0, q_1)(q_1, q_2) \dots (q_{m-1}, q_m)$ is *induced* by u, if there is an accepting run of M on u such that for all $i \in [1..m]$, q_i is the state reached by M upon reading $u_1 \dots u_i$. Note that we only consider the states that occur upon context switches. Moreover, induced sequences are valid by definition. Finally, note that a word with cs-many context switches induces an interface sequence of length precisely $cs + 1$. We define $IIF(S) \subseteq (Q \times Q)^*$ to be the *language of all induced interface sequences*.

Induced interface sequences witness non-emptiness of $L(S)$: $L(S) \neq \emptyset$ iff $IIF(S) \neq \emptyset$. Since the number of context switches is bounded by cs, we can thus iterate over all sequences in $(Q \times Q)^{\leq cs+1}$ and test each of them for being an induced interface sequence, i.e. an element of $IIF(S)$. Since induced sequences are valid, there are at most m^{cs+1} sequences to test.

Before turning to this test, we do a preprocessing step that removes the dependence on the memory automaton M. To this end, we define the *interface language* $IF(A_{id})$ of a thread. It makes visible the state changes on the shared memory that the contexts of this thread may induce. Formally, the interface language consists of all interface sequences $(q_1, q_1') \dots (q_k, q_k')$ so that $L(A_{id}) \cap (\ L(M(q_1, q_1')) \dots L(M(q_k, q_k'))\ ) \neq \emptyset$. These sequences do not have to be valid as the thread may be interrupted by others. Below, we rely on the fact that $IF(A_{id})$ is again a regular language, a representation of which is easy to compute.

▶ **Lemma 3.**

(i) *We have* $IIF(S) = \text{Ш}_{i\in[1..t]} IF(A_i) \cap \{\sigma \in (Q \times Q)^* \mid \sigma \text{ valid}\}$.

(ii) *One can compute in time* $\mathcal{O}(|A_{id}|^3 \cdot |M|^3)$ *an automaton* B_{id} *with* $L(B_{id}) = IF(A_{id})$.

It remains to check whether a valid sequence $\sigma \in (Q \times Q)^{cs+1}$ is included in the shuffle $\text{Ш}_{i\in[1..t]} L(B_i)$. This means we address the following problem:

Shuffle Membership (Shuff)
Input: NFAs $(B_i)_{i\in[1..t]}$ over the alphabet Γ, an integer k, and a word $w \in \Gamma^k$.
Question: Is w in $\text{Ш}_{i\in[1..t]}L(B_i)$?

We obtain the following upper bound, with b and $bc(k)$ as defined above.

▶ **Theorem 4.** Shuff *can be solved in time* $\mathcal{O}(2^k \cdot t \cdot k \cdot (b^2 + k \cdot bc(k)))$.

Our algorithm is based on *fast subset convolution* [7], an algebraic technique for summing up partitions of a given set. Typically, fast subset convolution is applied to graph problems: Björklund et al. [7] used it to present the first $\mathcal{O}^*(2^k)$-time algorithm for the Steiner Tree problem with k terminals and bounded edge weights. Cygan et al. incorporated a generalized version as a subprocedure in applications of their *Cut & Count* technique [17]. Variants of Dominating Set parameterized by treewidth were solved by van Rooij et al. in [49] using fast subset convolution. We are not aware of an automata-theoretic application.

Let $f, g : \mathcal{P}(B) \to \mathbb{Z}$ be two functions from the powerset of a k-element set B to the ring of integers. The *convolution* of f and g is the function $f * g : \mathcal{P}(B) \to \mathbb{Z}$ that maps a subset $S \subseteq B$ to the sum $\sum_{U\subseteq S} f(U)g(S \setminus U)$. Note that the convolution is associative. There is a close connection to partitions. For $t \in \mathbb{N}$, a *t-partition* of a set S is a tuple $(U_1, \ldots, U_t)$ of subsets of S such that $U_1 \cup \cdots \cup U_t = S$ and $U_i \cap U_j = \emptyset$ for all $i \neq j$. Now it is easy to see that the convolution of t functions $f_i : \mathcal{P}(B) \to \mathbb{Z}, i \in [1..t]$, sums up all t-partitions of S:

$$(f_1 * \cdots * f_t)(S) = \sum_{\substack{(U_1,\ldots,U_t)\\ \text{is a } t\text{-parition of } S}} f_1(U_1) \cdots f_t(U_t) \ .$$

To apply the convolution, we give a characterization of Shuff in terms of partitions. Let $((B_i)_{i\in[1..t]}, k, w)$ be an instance of Shuff. The following observation is crucial. The word w lies in the shuffle of the $L(B_i)$ if and only if there are non-overlapping, possibly empty (scattered) subwords $w_1, \ldots, w_t$ of w that decompose w and that satisfy $w_i \in L(B_i) \cup \{\varepsilon\}$ for all $i \in [1..t]$. By scattered, we mean that the subwords do not have to form an infix of w. Such a decomposition induces a t-partition $(U_1, \ldots, U_t)$ of the set of positions $\text{Pos} = \{1, \ldots, k\}$ of w, where each U_i holds exactly the positions of w_i. In turn, given a t-partition $(U_1, \ldots, U_t)$ of Pos, we can derive a decomposition of w by setting $w_i = w[U_i]$ for all $i \in [1..t]$. Here, $w[U_i]$ is the projection of w to the positions in U_i. Hence, w lies in the shuffle if and only if there is a t-partition $(U_1, \ldots, U_t)$ of Pos such that $w[U_i] \in L(B_i) \cup \{\varepsilon\}$ for all $i \in [1..t]$.

To express the language membership in $L(B_i)$ in terms of functions, we employ the characteristic functions $f_i : \mathcal{P}(\text{Pos}) \to \mathbb{Z}$ that map a set S to 1 if $w[S] \in L(B_i) \cup \{\varepsilon\}$, and to 0 otherwise. By the above formula, it follows that $(f_1 * \cdots * f_t)(\text{Pos}) > 0$ if and only if there is a t-partition $(U_1, \ldots, U_t)$ of Pos such that $f_i(U_i) = 1$ for $i \in [1..t]$. Altogether, we have proven the following lemma:

▶ **Lemma 5.** *The word* $w \in \Gamma^k$ *is in* $\text{Ш}_{i\in[1..t]}L(B_i)$ *if and only if* $(f_1 * \cdots * f_t)(\text{Pos}) > 0$.

Our algorithm for Shuff computes the characteristic functions f_i and $t-1$ convolutions to obtain $f_1 * \cdots * f_t$. Then it evaluates the convolution at the set Pos. Computing and storing a value $f_i(S)$ for a subset $S \subseteq \text{Pos}$ takes time $\mathcal{O}(k \cdot b^2)$ since we have to test membership of a word of length at most k in B_i. Hence, computing all f_i takes time $\mathcal{O}(2^k \cdot t \cdot k \cdot b^2)$. Due to Björklund et al. [7], we can compute the convolution of two functions $f, g : \mathcal{P}(\text{Pos}) \to \mathbb{Z}$ in $\mathcal{O}(2^k \cdot k^2)$ multiplications in $\mathbb{Z}$. Furthermore, if the ranges of f and g are bounded by C, we have to perform these operations on $\mathcal{O}(k \log C)$-bit integers [7]. Since the characteristic functions f_i have ranges bounded by a constant, we only need to compute with $\mathcal{O}(k)$-bit

integers. Hence, the $t-1$ convolutions can be carried out in time $\mathcal{O}(2^k \cdot k^2 \cdot (t-1) \cdot bc(k))$. Altogether, this proves Theorem 4.

Lower Bound for Bounded Context Switching. We prove a lower bound for the NP-hard BCS by reducing the subgraph isomorphism problem (SGI) to it. The result is such that it also applies to BCS(cs) and BCS(cs, m). We explain why the result is non-trivial.

In fine-grained complexity, lower bounds for W[1]-hard problems are often obtained by reductions from k-Clique. Chen et al. [13] have shown that k-Clique cannot be solved in time $f(k)n^{o(k)}$ for any computable function f, unless ETH fails. To transport the lower bound to a problem of interest, one has to construct a parameterized reduction that blows up the parameter only linearly. In the case of BCS, this fails. We face a well-known problem which was observed for reductions using edge-selection gadgets [45, 16]: A reduction from k-Clique would need to select a clique candidate of size k and check whether every two vertices of the candidate share an edge. This needs $\mathcal{O}(k^2)$ communications between the chosen vertices, which translates to $\mathcal{O}(k^2)$ context switches. Hence, we only obtain $n^{o(\sqrt{k})}$ as a lower bound.

To overcome this, we follow Marx [45] and give a reduction from SGI. This problem takes as input two graphs G and H and asks whether G is isomorphic to a subgraph of H. This means that there is an injective map $\varphi : V(G) \to V(H)$ such that for each edge (u, v) in G, the pair $(\varphi(u), \varphi(v))$ is an edge in H. We use $V(G)$ to denote the vertices and $E(G)$ to denote the edges of a graph G. Marx has shown that SGI cannot be solved in time $f(k)n^{o(k/\log k)}$, where k is the number of edges of G, unless ETH fails. In our reduction, the number of edges is mapped linearly to the number of context switches.

▶ **Theorem 6.** *Assuming* ETH*, there is no f s.t.* BCS *can be solved in $f(cs)n^{o(cs/\log(cs))}$.*

Roughly, the idea is this: The alphabet $V(G) \times V(H)$ describes how the vertices of G are mapped to vertices of H. Now we can use the memory M to output all possible injective maps from $V(G)$ to $V(H)$. There is one thread A_i for each edge of G. Its task is to verify that the edges of G get mapped to edges of H.

Note that Theorem 6 implies a lower bound for the FPT-problem BCS(cs, m). It cannot be solved in time $m^{o(cs/\log(cs))}$, unless ETH fails.

Lower Bound for Shuffle Membership. We prove it unlikely that Shuff can be solved in $\mathcal{O}^*((2-\delta)^k)$ time, for a $\delta > 0$. Hence, the $\mathcal{O}^*(2^k)$-time algorithm above may be optimal. We base our lower bound on a reduction from Set Cover. An instance consists of a family of sets $(S_i)_{i\in[1..m]}$ over a universe $U = \bigcup_{i\in[1..m]} S_i$, and an integer $t \in \mathbb{N}$. The problem asks for t sets $S_{i_1}, \ldots, S_{i_t}$ from the family such that $U = \bigcup_{j\in[1..t]} S_{i_j}$.

We are interested in a parameterization of the problem by the size n of the universe. It was shown that this parameterization admits an $\mathcal{O}^*(2^n)$-time algorithm [28]. But so far, no $\mathcal{O}^*((2-\varepsilon)^n)$-time algorithm was found, for an $\varepsilon > 0$. Actually, the authors of [15] conjecture that the existence of such an algorithm would contradict the *Strong Exponential Time Hypothesis* (SETH) [36, 11]. This is the assumption that n-variable SAT cannot be solved in $\mathcal{O}^*((2-\varepsilon)^n)$ time, for an $\varepsilon > 0$ (SETH implies ETH). By now, there is a list of lower bounds based on Set Cover [8, 15]. We add Shuff to this list.

▶ **Proposition 7.** *If* Shuff *can be solved in time $\mathcal{O}^*((2-\delta)^k)$ for a $\delta > 0$, then* Set Cover *can be solved in time $\mathcal{O}^*((2-\varepsilon)^n)$ for an $\varepsilon > 0$.*

Lower Bound on the Size of the Kernel. Kernelization is a preprocessing technique for parameterized problems that transforms a given instance to an equivalent instance of size

bounded by a function in the parameter. It is well-known that any FPT-problem admits a kernelization and any kernelization yields an FPT-algorithm [16]. The search for small problem kernels is ongoing research. A survey can be found in [42].

There is also the opposite approach, disproving the existence of a kernel of polynomial size [9, 30]. Such a result indicates hardness of the problem at hand, and hence serves as a lower bound. Technically, the existence of a polynomial kernel is linked to the inclusion $\mathsf{NP} \subseteq \mathsf{coNP/poly}$. The latter is unlikely as it would cause a collapse of the polynomial hierarchy to the third level [51]. Based on this approach, we show that $\mathsf{BCS}(cs, m)$ does not admit a kernel of polynomial size. We introduce the needed notions, following [16].

A *kernelization* for a parameterized problem Q is an algorithm that, given an instance (I, k), returns an equivalent instance (I', k') in polynomial time such that $|I'| + k' \leq g(k)$ for some computable function g. If g is a polynomial, Q is said to admit a *polynomial kernel*.

We also need *polynomial equivalence relations*. These are equivalence relations on Σ^*, with Σ some alphabet, such that: (1) There is an algorithm that, given $x, y \in \Sigma^*$, decides whether $(x, y) \in \mathcal{R}$ in time polynomial in $|x| + |y|$. (2) For every n, $\mathcal{R}$ restricted to $\Sigma^{\leq n}$ has at most polynomially (in n) many equivalence classes.

To relate parameterized and unparameterized problems, we employ *cross-compositions*. Consider a language $L \subseteq \Sigma^*$ and a parameterized language $Q \subseteq \Sigma^* \times \mathbb{N}$. Then L *cross-composes* into Q if there is a polynomial equivalence relation $\mathcal{R}$ and an algorithm $\mathcal{A}$, referred to as the *cross-composition*, with: $\mathcal{A}$ takes as input a sequence $x_1, \ldots, x_t \in \Sigma^*$ of strings that are equivalent with respect to $\mathcal{R}$, runs in time polynomial in $\Sigma_{i=1}^{t} |x_i|$, and outputs an instance (y, k) of Q such that $k \leq p(\max_{i \in [1..t]} |x_i| + \log(t))$ for a polynomial p. Moreover, $(y, k) \in Q$ if and only if there is an $i \in [1..t]$ such that $x_i \in L$. Cross-compositions are the key to lower bounds for kernels:

▶ **Theorem 8** ([16]). *Assume that an* NP*-hard language cross-composes into a parameterized language Q. Then Q does not admit a polynomial kernel, unless* $\mathsf{NP} \subseteq \mathsf{coNP/poly}$.

To show that $\mathsf{BCS}(cs, m)$ does not admit a polynomial kernel, we cross-compose 3-SAT into $\mathsf{BCS}(cs, m)$. Then Theorem 8 yields the following:

▶ **Theorem 9.** $\mathsf{BCS}(cs, m)$ *does not admit a polynomial kernel, unless* $\mathsf{NP} \subseteq \mathsf{coNP/poly}$.

Proof Idea. For the cross-composition, we first need a polynomial equivalence relation $\mathcal{R}$. Assume some standard encoding of 3-SAT-instances over a finite alphabet Γ. We let two encodings φ, ψ be equivalent with respect to $\mathcal{R}$ if both are proper 3-SAT-instances and have the same number of clauses and variables.

Let $\varphi_1, \ldots, \varphi_t$ be instances of 3-SAT that are equivalent with respect to $\mathcal{R}$. Then each φ_i has exactly ℓ clauses and k variables. We can assume that the set of variables is $\{x_1, \ldots, x_k\}$. To handle the evaluation of these, we introduce the NFAs $A_i, i \in [1..k]$, each storing the value of x_i. We further construct an automaton B that picks one out of the t formulas φ_j. Automaton B tries to satisfy φ_j by iterating through the ℓ clauses. To satisfy a clause, B chooses one out of the three variables and requests the corresponding value.

The request by B is synchronized with the memory M. After every such request, M either ensures that the sent variable x_i actually has the requested value or stops the computation. This is achieved by a synchronization with the corresponding variable automaton A_i, which keeps the value of x_i. The number of context switches lies in $\mathcal{O}(\ell)$ and the size of the memory in $\mathcal{O}(k)$. Hence, all conditions for a cross-composition are met. ◀

4 Local Parameterization

In the previous section, we considered a parameterization of BCS that was global in the sense that the threads shared the number of context switches. We now study a parameterization that is *local* in that every thread is given a budget of context switches.

We would like to have a measure for the amount of communication between processes and consider only those computations in which heavily interacting processes are scheduled adjacent to each other. The idea relates to [43], where it is observed that a majority of concurrency bugs already occur between a few interacting processes.

Given a word $u \in \text{⧢}_{i\in[1..t]} L(A_i)$, we associate with it a graph that reflects the order in which the threads take turns. This *scheduling graph* of u is the directed multigraph $G(u) = (V, E)$ with one node per thread that participates in u, $V \subseteq [1..t]$, and edge weights $E : V \times V \to \mathbb{N}$ defined as follows. Value $E(i, j)$ is the number of times the context switches from thread i to thread j in u. Formally, this is the number of different decompositions $u = u_1.a.b.u_2$ of u so that a is in the alphabet of A_i and b is in the alphabet of A_j. Note that $E(i, i) = 0$ for all $i \in [1..t]$. In the following we refer to directed multigraphs simply as graphs.

In the scheduling graph, the degree of a node corresponds to the number of times the thread has the processor. The *degree* of a node n in $G = (V, E)$ is the maximum over the outdegree and the indegree, $deg(n) = max\{indeg(n), outdeg(n)\}$. As usual, the outdegree of a node n is the number of edges leaving the node, $outdeg(n) = \sum_{n'\in V} E(n, n')$, the indegree is defined similarly. To see the correspondence, observe that a scheduling graph can have three kinds of nodes. The *initial node* is the only node where the indegree equals the outdegree minus 1, and the thread has the processor outdegree many times. For the *final node*, the outdegree equals the indegree minus 1, and the thread computes for indegree many contexts. For all other (usual) nodes, indegree and outdegree coincide. Any scheduling graph either has one initial, one final, and only usual nodes or, if the computation starts and ends in the same thread, only consists of usual nodes. The degree of the graph is the maximum among the node degrees, $deg(G) = max\{deg(n) \mid n \in V\}$.

Our goal is to measure the complexity of schedules. Intuitively, a schedule is simple if the threads take turns following some pattern, say round robin where they are scheduled in a cyclic way. To formalize the idea of scheduling patterns, we iteratively contract scheduling graphs to a single node and measure the degrees of the intermediary graphs. If always the same threads follow each other, we will be able to merge the nodes of such neighboring threads without increasing the degree of the resulting graph. This discussion leads to a notion of scheduling dimension that we define in the following paragraph. In the full version of the paper [14], we elaborate on the relation to an established measure: The carving-width.

Given a graph $G = (V, E)$, two nodes $n_1, n_2 \in V$, and $n \notin V$, we define the operation of *contracting* n_1 *and* n_2 into the fresh node n by adding up the incoming and outgoing edges. Formally, the graph $G[n_1, n_2 \mapsto n] = (V', E')$ contains the vertices $V' = (V \setminus \{n_1, n_2\}) \cup \{n\}$ and has the edge weights $E'(n', n) = E(n', n_1) + E(n', n_2)$, $E'(n, n') = E(n_1, n') + E(n_2, n')$, and $E'(m, m') = E(m, m')$ for all other nodes. Using iterated contraction, we can reduce a graph to only one node. Formally, a *contraction process* of G is a sequence $\pi = G_1, \ldots, G_{|V|}$ of graphs, where $G_1 = G$, $G_{k+1} = G_k[n_1, n_2 \mapsto n]$ for some $n_1, n_2 \in V(G_k)$ and $n \notin V(G_k)$, $k \in [1.. |V| - 1]$, and $G_{|V|}$ consists of a single node. The degree of a contraction process is the maximum of the degrees of the graphs in that process, $deg(\pi) = max\{deg(G_i) \mid i \in [1.. |V|]\}$. The *scheduling dimension* of G is $sdim(G) = min\{deg(\pi) \mid \pi \text{ a contraction process of } G\}$.

We study the complexity of BCS when parameterized by the scheduling dimension. To this end, we define the language of all words where the scheduling dimension is bounded by the parameter $sdim \in \mathbb{N}$: $SDL(\Sigma, t, sdim) = \{u \in (\Sigma \times [1..t])^* \mid sdim(G(u)) \leq sdim\}$.

Bounded Context Switching – Local Parameterization (BCS-L)
Input: $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ and bound $sdim \in \mathbb{N}$ on the scheduling dimension.
Question: Is $L(S) \cap SDL(\Sigma, t, sdim) \neq \emptyset$?

▶ **Theorem 10.** BCS-L *can be solved in time* $\mathcal{O}^*((2m)^{4sdim}4^t)$.

We present a fixed-point iteration that mimics the definition of contraction processes by iteratively joining the interface sequences of neighboring threads. Towards the definition of a suitable composition operation, let the *product* of two interface sequences σ and τ be $\sigma \otimes \tau = \bigcup_{\rho\in\sigma \text{ш} \tau} \rho\downarrow$. Language $\rho\downarrow$ consists of all interface sequences ρ' obtained by summarizing subsequences in ρ. Summarizing $(r_1, r'_1)\dots(r_n, r'_n)$ where $r'_1 = r_2$ up to $r'_{n-1} = r_n$ means to contract the sequence to (r_1, r'_n). We write $\sigma \otimes^k \tau$ for the variant of the product that only returns interface sequences of length at most $k \geq 1$, $(\sigma \otimes \tau) \cap (Q \times Q)^{\leq k}$.

Our algorithm computes a fixed point over the powerset lattice (ordered by inclusion) $\mathcal{P}(\,(Q\times Q)^{\leq sdim} \times \mathcal{P}([1..t])\,)$. The elements are *generalized interface sequences*, pairs consisting of an interface sequence together with the set of threads that has been used to construct it. We generalize $\otimes^k$ to this domain. For the definition, consider (σ_1, T_1) and (σ_2, T_2). If the sets of threads are not disjoint, $T_1 \cap T_2 \neq \emptyset$, the sequences cannot be merged and we obtain $(\sigma_1, T_1) \otimes (\sigma_2, T_2) = \emptyset$. If the sets are disjoint, we define $(\sigma_1, T_1) \otimes^k (\sigma_2, T_2) = (\sigma_1 \otimes^k \sigma_2) \times \{T_1 \cup T_2\}$. The fixed-point iteration is given by $L_1 = \bigcup_{i\in[1..t]} IF(A_i) \times \{\{i\}\}$ and $L_{i+1} = L_i \cup (L_i \otimes^{sdim} L_i)$. The following lemma states that it solves BCS-L. We elaborate on the complexity in the full version of the paper [14].

▶ **Lemma 11.** BCS-L *holds iff the least fixed point contains* $((q_{init}, q_{final}), T)$ *for some* T.

Problem BCS-L can be generalized and can be restricted in natural ways. We discuss both options and show that variants of the above algorithm still apply.

Let BCS-L-ANY be the variant of BCS-L where each thread is given a budget of running cs times, but where we do not make any assumption on the scheduling. Still, the scheduling dimension is bounded by $t \cdot cs$. The above algorithm solves BCS-L-ANY in time $\mathcal{O}^*((2m)^{4t\cdot cs}4^t)$.

Fixing the Scheduling Graph. We consider BCS-L-FIX, a variant of BCS-L where we fix a scheduling graph together with a contraction process of degree bounded by *sdim*. We are interested in finding an accepting computation that switches contexts as depicted by the fixed graph. Formally, BCS-L-FIX takes as input an SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$, a scheduling graph G, and a contraction process π of G of degree at most *sdim*. The task is to find a word $u \in L(S)$ such that $G(u) = G$. Our main observation is that a variant of the above algorithm applies and yields a running time polynomial in t.

▶ **Theorem 12.** BCS-L-FIX *can be solved in time* $\mathcal{O}^*((2m)^{4sdim})$.

Fixing the scheduling graph $G = (V, E)$ and contraction process π has two crucial implications on the above algorithm. First, we need to contract interface sequences according to the structure of G. To this end, we introduce a new product. Secondly, instead of a fixed point we can now compute the required products iteratively along π. Hence, we do not have to maintain the set of threads in the domain but can compute on $\mathcal{P}((Q \times Q)^{\leq sdim})$.

Towards obtaining the algorithm, we first describe the new product that summarizes interface sequences along the graph structure. Let σ and τ be interface sequences. Further, let $\rho \in \sigma \text{ш} \tau$. We call a position in ρ an *out-contraction* if it is of the form $(q, q')(p, p')$ so that (q, q') belongs to σ, (p, p') belongs to τ, and $q' = p$. Similarly, we define *in-contractions*. These are positions where a pair of states of τ is followed by a pair of σ. The *directed product*

of σ and τ is defined as: $\sigma \odot_{(i,j)} \tau = \bigcup_{\rho \in \sigma \amalg \tau} \rho \downarrow_{(i,j)}$. The language $\rho \downarrow_{(i,j)}$ contains all interface sequences ρ' obtained by summarizing subsequences of ρ, in total containing exactly i out-contractions and j in-contractions. Note that for $\sigma \in (Q \times Q)^n$ and $\tau \in (Q \times Q)^k$, the directed product contracts at $i + j$ positions and yields: $\sigma \odot_{(i,j)} \tau \subseteq (Q \times Q)^{n+k-(i+j)}$.

Now we describe the iteration. First, we may assume that $V = [1..t]$. Otherwise, the non-participating threads in S can be deleted. We distinguishes two cases.

In the first case, we assume that G has a designated initial vertex v_0 and final vertex v_f. Let $\pi = G_1, \ldots, G_t$. The iteration starts by assigning to each $v \in V$ the set $S_v = IF(A_v) \cap (Q \times Q)^{deg(v)}$. For S_{v_0}, we further require the first component of the first pair occurring in an interface sequence to be q_{init}. Similarly, for S_{v_f} we require that the second component of the last pair is q_{final}.

Now we iterate along π: For each contraction $G_{j+1} = G_j[n_1, n_2 \mapsto n]$, we compute $S_n = (S_{n_1} \odot_{(i,k)} S_{n_2})$, where $i = E(n_1, n_2)$ and $k = E(n_2, n_1)$. Then $S_n \subseteq (Q \times Q)^{deg(n)}$, where $deg(n)$ is the degree of n in G_{j+1}. Let $V(G_t) = \{w\}$. Then the algorithm terminates after S_w has been computed.

For the second case, suppose that no initial vertex is given. This means that initial and final vertex coincide. Then we iterate through all vertices in V, designate any to be initial (and final), and run the above algorithm. The correctness is shown in the following lemma.

▶ **Lemma 13.** BCS-L-FIX *holds iff* $(q_{init}, q_{final}) \in S_w$.

Round Robin. We consider an application of BCS-L-FIX. We define BCS-L-RR to be the round-robin version of BCS-L. Again, each thread is given cs contexts, but now we schedule the threads in a fixed order: First thread A_1 has the processor, then A_2, followed by A_3 up to A_t. For a new round, the processor is given back to A_1. The computation ends in A_t.

▶ **Proposition 14.** BCS-L-RR *can be solved in time* $\mathcal{O}^*(m^{4cs})$.

The problem BCS-L-RR can be understood as fixing the scheduling graph to a cycle where every node i is connected to $i + 1$ by an edge of weight cs for $i \in [1..t-1]$ and the nodes t and 1 are connected by an edge of weight $cs - 1$. We can easily describe a contraction process: Contract the vertices 1 and 2, then the result with vertex 3 and up to t. We refer to this as π. Then we have $deg(\pi) = cs$. Hence, we have constructed an instance of BCS-L-FIX.

An application of the algorithm for BCS-L-FIX takes time at most $\mathcal{O}^*(m^{4cs})$ in this case: Let $G_{j+1} = G_j[n_1, n_2 \mapsto n]$ be a contraction in π with $j < t - 1$. Note that $S_{n_1}, S_{n_2} \subseteq (Q \times Q)^{cs}$. We have $E(n_1, n_2) = cs$ and $E(n_2, n_1) = 0$. Hence, the corresponding set S_n is given by $(S_{n_1} \odot_{(cs,0)} S_{n_2}) \subseteq (Q \times Q)^{cs}$. The directed product $\sigma \odot_{(cs,0)} \tau$ can be computed in linear time: Any sequence ρ' in $\sigma \odot_{(cs,0)} \tau$ is obtained from a sequence $\rho \in \sigma \amalg \tau$ by summarizing cs many out-contractions. Since σ and τ both have length cs, ρ has to be the sequence where pairs of states of σ and τ alternatively take turns. Hence, $\sigma \odot_{(cs,0)} \tau$ either only consists of ρ' and it is a linear-time procedure to find it, or is empty. For the last contraction $G_t = G_{t-1}[n'_1, n'_2 \mapsto n']$ we have $S_{n'} = (S_{n'_1} \odot_{(cs,cs-1)} S_{n'_2})$. Similar to $\sigma \odot_{(cs,0)} \tau$, one can compute $\sigma \odot_{(cs,cs-1)} \tau$ in linear time. This avoids the cost of the product, the factor 2^{4cs}, in the complexity estimation.

Lower Bound for Round Robin. We prove the optimality of the algorithm for BCS-L-RR by giving a reduction from k × k Clique. This variant of the classical clique problem asks for a clique of size k in a graph whose vertices are the elements of a $k \times k$ matrix. Furthermore, the clique must contain exactly one vertex from each row. The problem was introduced as a part

of the framework in [41]. It was shown that the brute-force approach is optimal: k × k Clique cannot be solved in $2^{o(k \log k)}$ time, unless ETH fails. We transport this to BCS-L-RR.

▶ **Lemma 15.** *Assuming* ETH, BCS-L-RR *cannot be solved in time* $2^{o(cs \log(m))}$.

5 Discussion

Our main motivation was to find bugs in shared-memory concurrent programs. We restricted our analysis to under-approximations and considered behaviors that are bounded in the number of context switches, the memory size, or the scheduling. While this is enough to find bugs, there are cases where we need to check correctness of a program. We shortly outline an FPT upper bound, as well as a matching lower bound for the problem.

The reachability problem on a shared-memory concurrent program in full generality is PSPACE-complete. However, in real-world scenarios, it is often the case that only few (a fixed number of) threads execute in parallel with unbounded interaction. Thus, a first attempt is to parameterize the system by the number of threads t. But this yields a hardness result. Indeed, the problem with t as a parameter is hard for any level of the W-hierarchy.

We suggest a parameterization by the number of threads t and by a, the maximal size of the thread automata A_{id}. We obtain an FPT-algorithm by constructing a product automaton. The complexity is $\mathcal{O}^*(a^t)$. However, there is not much hope for improvement: By a reduction from k × k Clique, we can show that the algorithm is indeed optimal.

References

1 M. F. Atig. Global model checking of ordered multi-pushdown systems. In *FSTTCS*, volume 8 of *LIPIcs*, pages 216–227. Schloss Dagstuhl, 2010.

2 M. F. Atig, A. Bouajjani, K. N. Kumar, and P. Saivasan. On bounded reachability analysis of shared memory systems. In *FSTTCS*, volume 29 of *LIPIcs*, pages 611–623. Schloss Dagstuhl, 2014.

3 M. F. Atig, A. Bouajjani, and T. Touili. Analyzing asynchronous programs with preemption. In *FSTTCS*, volume 2 of *LIPIcs*, pages 37–48. Schloss Dagstuhl, 2008.

4 M. F. Atig, A. Bouajjani, and T. Touili. On the reachability analysis of acyclic networks of pushdown systems. In *CONCUR*, volume 5201 of *LNCS*, pages 356–371. Springer, 2008.

5 M.F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *LMCS*, 7(4), 2011.

6 J. Barnat, L. Brim, I. Cerná, P. Moravec, P. Rockai, and O. Simecek. Divine - A tool for distributed verification. In *CAV*, volume 4144 of *LNCS*, pages 278–281. Springer, 2006.

7 A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets Möbius: fast subset convolution. In *STOC*, pages 67–74. ACM, 2007.

8 A. Björklund, P. Kaski, and L. Kowalik. Constrained multilinear detection and generalized graph motifs. *Algorithmica*, 74(2):947–967, 2016.

9 H. L. Bodlaender, R. G. Downey, M. R. Fellows, and D. Hermelin. On problems without polynomial kernels. *JCSS*, 75(8):423–434, 2009.

10 A. Bouajjani, M. Emmi, and G. Parlato. On sequentializing concurrent programs. In *SAS*, volume 6887 of *LNCS*, pages 129–145. Springer, 2011.

11 C. Calabro, R. Impagliazzo, and R. Paturi. The complexity of satisfiability of small depth circuits. In *IWPEC*, volume 5917 of *LNCS*, pages 75–85. Springer, 2009.

12 J.F. Cantin, M.H. Lipasti, and J.E. Smith. The complexity of verifying memory coherence and consistency. *TPDS*, 16(7):663–671, 2005.

13 J. Chen, X. Huang, I. A. Kanj, and G. Xia. Strong computational lower bounds via parameterized complexity. *JCSS*, 72(8):1346–1367, 2006.

14 P. Chini, J. Kolberg, A. Krebs, R. Meyer, and P. Saivasan. On the complexity of bounded context switching. *CoRR*, abs/1609.09728, 2017. URL: `https://arxiv.org/abs/1609.09728`.

15 M. Cygan, H. Dell, D. Lokshtanov, D. Marx, J. Nederlof, Y. Okamoto, R. Paturi, S. Saurabh, and M. Wahlström. On problems as hard as CNF-SAT. *ACM TALG*, 12(3):41:1–41:24, 2016.

16 M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized algorithms*. Springer, 2015.

17 M. Cygan, J. Nederlof, M. Pilipczuk, M. Pilipczuk, J. M. M. van Rooij, and J. O. Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *FOCS*, pages 150–159. IEEE, 2011.

18 S. Demri, F. Laroussinie, and P. Schnoebelen. A parametric analysis of the state explosion problem in model checking. In *STACS*, volume 2285 of *LNCS*, pages 620–631. Springer, 2002.

19 R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.

20 A. Durand-Gasselin, J. Esparza, P. Ganty, and R. Majumdar. Model checking parameterized asynchronous shared-memory systems. In *CAV*, volume 9206 of *LNCS*, pages 67–84. Springer, 2015.

21 C. Enea and A. Farzan. On atomicity in presence of non-atomic writes. In *TACAS*, volume 9636 of *LNCS*, pages 497–514. Springer, 2016.

22 J. Esparza, P. Ganty, and R. Majumdar. Parameterized verification of asynchronous shared-memory systems. In *CAV*, volume 8044 of *LNCS*, pages 124–140. Springer, 2013.

23 J. Esparza, P. Ganty, and T. Poch. Pattern-based verification for multithreaded programs. *ACM TOPLAS*, 36(3):9:1–9:29, 2014.

24 A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *TACAS*, volume 5505 of *LNCS*, pages 155–169. Springer, 2009.

25 H. Fernau, P. Heggernes, and Y. Villanger. A multi-parameter analysis of hard problems on deterministic finite automata. *JCSS*, 81(4):747–765, 2015.

26 H. Fernau and A. Krebs. Problems on finite automata and the exponential time hypothesis. In *CIAA*, volume 9705 of *LNCS*, pages 89–100. Springer, 2016.

27 J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006.

28 F. V. Fomin, D. Kratsch, and G. J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In *WG*, volume 3353 of *LNCS*, pages 245–256. Springer, 2004.

29 M. Fortin, A. Muscholl, and I. Walukiewicz. On parametrized verification of asynchronous, shared-memory pushdown systems. *CoRR*, abs/1606.08707, 2016.

30 L. Fortnow and R. Santhanam. Infeasibility of instance compression and succinct PCPs for NP. *JCSS*, 77(1):91–106, 2011.

31 F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. Memory-model-aware testing: A unified complexity analysis. *ACM TECS*, 14(4):63:1–63:25, 2015.

32 M. Fürer. Faster integer multiplication. *SICOMP*, 39(3):979–1005, 2009.

33 P. B. Gibbons and E. Korach. Testing shared memories. *SICOMP*, 26(4):1208–1244, 1997.

34 M. Hague. Parameterised pushdown systems with non-atomic writes. In *FSTTCS*, volume 13 of *LIPIcs*, pages 457–468. Schloss Dagstuhl, 2011.

35 D. Harvey, J. van der Hoeven, and G. Lecerf. Even faster integer multiplication. *Journal of Complexity*, 36:1–30, 2016.

36 R. Impagliazzo and R. Paturi. On the complexity of k-sat. *JCSS*, 62(2):367–375, 2001.

37 S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170. IEEE, 2007.

38 S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV*, volume 6174 of *LNCS*, pages 629–644. Springer, 2010.

39 S. La Torre and M. Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *CONCUR*, volume 6901 of *LNCS*, pages 203–218. Springer, 2011.

40 A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV*, volume 5123 of *LNCS*, pages 37–51. Springer, 2008.

41 D. Lokshtanov, D. Marx, and S. Saurabh. Slightly superexponential parameterized problems. In *SODA*, pages 760–776. SIAM, 2011.

42 D. Lokshtanov, N. Misra, and S. Saurabh. Kernelization – preprocessing with a guarantee. In *The Multivariate Algorithmic Revolution and Beyond - Essays Dedicated to Michael R. Fellows on the Occasion of His 60th Birthday*, volume 7370 of *LNCS*, pages 129–161. Springer, 2012.

43 S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339. ACM, 2008.

44 D. Marx. Can you beat treewidth? In *FOCS*, pages 169–179. IEEE, 2007.

45 D. Marx. Can you beat treewidth? *TOC*, 6(1):85–112, 2010.

46 M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455. ACM, 2007.

47 H. Ponce de León, F. Furbach, K. Heljanko, and R. Meyer. Portability analysis for axiomatic memory models. PORTHOS: one tool for all models. *To appear at SAS*, 2017.

48 S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

49 J. M. M. van Rooij, H. L. Bodlaender, and P. Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In *ESA*, volume 5757 of *LNCS*, pages 566–577. Springer, 2009.

50 T. Wareham. The parameterized complexity of intersection and composition operations on sets of finite-state automata. In *CIAA*, volume 2088 of *LNCS*, pages 302–310. Springer, 2000.

51 C. K. Yap. Some consequences of non-uniform conditions on uniform classes. *TCS*, 26:287–300, 1983.

Improved Approximate Rips Filtrations with Shifted Integer Lattices*

Aruni Choudhary[1], Michael Kerber[†2], and Sharath Raghvendra[‡3]

1 **Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
aruni.choudhary@mpi-inf.mpg.de
2 **Graz University of Technology, Graz, Austria**
kerber@tugraz.at
3 **Virginia Tech, Blacksburg, VA, USA**
sharathr@vt.edu

Abstract

Rips complexes are important structures for analyzing topological features of metric spaces. Unfortunately, generating these complexes constitutes an expensive task because of a combinatorial explosion in the complex size. For n points in $\mathbb{R}^d$, we present a scheme to construct a $3\sqrt{2}$-approximation of the multi-scale filtration of the L_∞-Rips complex, which extends to a $O(d^{0.25})$-approximation of the Rips filtration for the Euclidean case. The k-skeleton of the resulting approximation has a total size of $n2^{O(d \log k)}$. The scheme is based on the integer lattice and on the barycentric subdivision of the d-cube.

1998 ACM Subject Classification F.2.2 Geometrical problems and computations

Keywords and phrases Persistent homology, Rips filtrations, Approximation algorithms, Topological Data Analysis

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.28

1 Introduction

Persistent homology [4, 10, 11] is a technique to analyze of data sets using topological invariants. The idea is to build a multi-scale representation of the data set and to track its homological changes across the scales.

A standard construction for the important case of point clouds in Euclidean space is the *Vietoris-Rips complex* (or just *Rips complex*): for a scale parameter $\alpha \geq 0$, it is the collection of all subsets of points with diameter at most α. When α increases from 0 to ∞, the Rips complexes form a *filtration*, an increasing sequence of nested simplicial complexes whose homological changes can be computed and represented in terms of a *barcode*.

The computational drawback of Rips complexes is their sheer size: the k-skeleton of a Rips complex (that is, only subsets of size $\leq k+1$ are considered) for n points consists of $\Theta(n^{k+1})$ simplices because every $(k+1)$-subset joins the complex for a sufficiently large scale parameter. This size bound turns barcode computations for large point clouds infeasible even for low-dimensional homological features[1]. This poses the question of what we can say about the barcode of the Rips filtration without explicitly constructing all of its simplices.

* A longer version of the paper is available on arXiv [7].
† MK is supported by the Austrian Science Fund (FWF) grant number P 29984-N35.
‡ SR acknowledges support of NSF CRII grant CCF-1464276.
1 An exception are point clouds in $\mathbb{R}^2$ and $\mathbb{R}^3$, for which *alpha complexes* [10] are an efficient alternative.

We address this question using approximation techniques. Barcodes form a metric space: two barcodes are close if the same homological features occur on roughly the same range of scales (see Section 2 for the precise definition). The first approximation scheme by Sheehy [16] constructs a $(1+\varepsilon)$-approximation of the k-skeleton of the Rips filtration using only $n(\frac{1}{\varepsilon})^{O(\lambda k)}$ simplices for arbitrary finite metric spaces, where λ is the doubling dimension of the metric. Further approximation techniques for Rips complexes [9] and the closely related *Čech complexes* [1, 5, 13] have been derived subsequently, all with comparable size bounds. More recently, we constructed an approximation scheme for Rips complexes in Euclidean space that yields a worse approximation factor of $O(d)$, but uses only $n2^{O(d \log k)}$ simplices [8], where d is the ambient dimension of the point set.

Contributions. We present a $3\sqrt{2}$-approximation for the Rips filtration of n points in $\mathbb{R}^d$ in the L_∞-norm , whose k-skeleton has size $n2^{O(d \log k)}$. This translates to a $O(d^{0.25})$-approximation of the Rips filtration in the Euclidean metric and hence improves the asymptotic approximation quality of our previous approach [8] with the same size bound.

On a high level, our approach follows a straightforward approximation scheme: given a scaled and appropriately shifted integer grid on $\mathbb{R}^d$, we identify those grid points that are close to the input points and build an approximation complex using these grid points. The challenge lies in how to connect these grid points to a simplicial complex such that close-by grid points are connected, while avoiding too many connections to keep the size small. Our approach first selects a set of *active faces* in the cubical complex defined over the grid, and defines the approximation complex using the barycentric subdivision of this cubical complex.

We also describe an output-sensitive algorithm to compute our approximation. By randomizing the aforementioned shifts of the grids, we obtain a worst-case running time of $n2^{O(d)} \log \Delta + 2^{O(d)} M$, where Δ is *spread* of the point set (that is, the ratio of the diameter to the closest distance of two points) and M is the size of the approximation.

Additionally, this paper makes the following technical contributions:

- We follow the standard approach of defining a sequence of approximation complexes and establishing an *interleaving* between the Rips filtration and the approximation. We realize our interleaving using *chain maps* connecting a Rips complex at scale α to an approximation complex at scale $c\alpha$, and vice versa, with $c \geq 1$ being the approximation factor. Previous approaches [8, 9, 16] used *simplicial maps* for the interleaving, which induce an elementary form of chain maps and are therefore more restrictive.
  The explicit construction of such maps can be a non-trivial task. The novelty of our approach is that we avoid this construction by the usage of *acyclic carriers* [15]. In short, carriers are maps that assign subcomplexes to subcomplexes under some mild extra conditions. While they are more flexible, they still certify the existence of suitable chain maps, as we exemplify in Section 4. We believe that this technique is of general interest for the construction of approximations of cell complexes.
- We exploit a simple trick that we call *scale balancing* to improve the quality of approximation schemes. In short, if the aforementioned interleaving maps from and to the Rips filtration do not increase the scale parameter by the same amount, one can simply multiply the scale parameter of the approximation by a constant. Concretely, given maps

  $$\phi_\alpha : \mathcal{R}_\alpha \to \mathcal{X}_\alpha \qquad \psi_\alpha : \mathcal{X}_\alpha \to \mathcal{R}_{c\alpha}$$

  interleaving the Rips complex $\mathcal{R}_\alpha$ and the approximation complex $\mathcal{X}_\alpha$, we can define $\mathcal{X}'_\alpha := \mathcal{X}_{\alpha/\sqrt{c}}$ and obtain maps

  $$\phi'_\alpha : \mathcal{R}_\alpha \to \mathcal{X}'_{\sqrt{c}\alpha} \qquad \psi_\alpha : \mathcal{X}'_\alpha \to \mathcal{R}_{\sqrt{c}\alpha}$$

which improves the interleaving from c to $\sqrt{c}$. While it has been observed that the same trick can be used for improving the worst-case distance between Rips and Čech filtrations[2], our work seems to be the first to make use of it in the context of approximations.

Our technique can be combined with dimension reduction techniques in the same way as in [8] (see Theorems 19, 21, and 22 therein), with improved logarithmic factors. We omit the technical details in this paper. Also, we point out that the complexity bounds for size and computation time are for the entire approximation scheme and not for a single scale as in [8]. However, similar techniques as the ones exposed in Section 5 can be used to improve the results of [8] to hold for the entire approximation as well[3].

Outline. We start the presentation by discussing the relevant topological concepts in Section 2. Then, we present few results about grid lattices in Section 3. Building on these ideas, the approximation scheme is presented in Section 4. Computational aspects of the approximation scheme are discussed in Section 5. We conclude in Section 6. Many of the proofs are detailed in the arXiv version of our paper [7].

2 Background

We review the essential topological concepts needed; see [2, 6, 10, 15] for more details.

Simplicial complexes. A *simplicial complex* K on a finite set of elements S is a collection of subsets $\{\sigma \subseteq S\}$ called *simplices* such that each subset $\tau \subset \sigma$ is also in K. The dimension of a simplex $\sigma \in K$ is $k := |\sigma| - 1$, in which case σ is called a *k-simplex*. A simplex τ is a *subsimplex* of σ if $\tau \subseteq \sigma$. We remark that, commonly a subsimplex is called a 'face' of a simplex, but we reserve the word 'face' for a different structure. For the same reason, we do not introduce the common notation of of 'vertices' and 'edges' of simplicial complexes, but rather refer to 0- and 1-simplices throughout. The *k-skeleton* of K consists of all simplices of K whose dimension is at most k. For instance, the 1-skeleton of K is a graph defined by its 0-simplices and 1-simplices.

Given a point set $P \subset \mathbb{R}^d$ and a real number $\alpha \geq 0$, the *(Vietoris-)Rips* complex on P at scale α consists of all simplices $\sigma = (p_0, \ldots, p_k) \subseteq P$ such that $diam(\sigma) \leq \alpha$, where $diam$ denotes the diameter. In this work, we write $\mathcal{R}_\alpha$ for the Rips complex at scale α with the Euclidean metric, and $\mathcal{R}^\infty_\alpha$ when using the metric of the L_∞-norm. In either way, a Rips complex is an example of a *flag complex*, which means that whenever a set $\{p_0, \ldots, p_k\} \subseteq P$ has the property that every 1-simplex $\{p_i, p_j\}$ is in the complex, then the k-simplex $\{p_0, \ldots, p_k\}$ is also in the complex.

A simplicial complex K' is a *subcomplex* of K if $K' \subseteq K$. For instance, $\mathcal{R}_\alpha$ is a subcomplex of $\mathcal{R}_{\alpha'}$ for $0 \leq \alpha \leq \alpha'$. Let L be a simplicial complex. Let $\hat{\varphi}$ be a map which assigns to each vertex of K, a vertex of L. A map $\varphi : K \to L$ is called a *simplicial map* induced by $\hat{\varphi}$, if for every simplex $\{p_0, \ldots, p_k\}$ in K, the set $\{\hat{\varphi}(p_0), \ldots, \hat{\varphi}(p_k)\}$ is a simplex of L. For K' a subcomplex of K, the inclusion map $inc : K' \to K$ is an example of a simplicial map. A simplicial map $K \to L$ is completely determined by its action on the 0-simplices of K.

[2] Ulrich Bauer, private communication
[3] An extended version of [8] containing these improvements is currently under submission.

Chain complexes. A *chain complex* $\mathcal{C}_* = (\mathcal{C}_p, \partial_p)$ with $p \in \mathbb{N}$ is a collection of abelian groups $\mathcal{C}_p$ and homomorphisms $\partial_p : \mathcal{C}_p \to \mathcal{C}_{p-1}$ such that $\partial_{p-1} \circ \partial_p = 0$. A simplicial complex K gives rise to a chain complex $\mathcal{C}_*(K)$ by fixing a base field $\mathcal{F}$, defining $\mathcal{C}_p$ as the set of formal linear combinations of p-simplices in K over $\mathcal{F}$, and ∂_p as the linear operator that assigns to each simplex the (oriented) sum of its sub-simplices of codimension one[4].

A *chain map* $\phi : \mathcal{C}_* \to D_*$ between chain complexes $\mathcal{C}_* = (\mathcal{C}_p, \partial_p)$ and $D_* = (D_p, \partial'_p)$ is a collection of group homomorphisms $\phi_p : \mathcal{C}_p \to D_p$ such that $\phi_{p-1} \circ \partial_p = \partial'_p \circ \phi_p$. For example, a simplicial map φ between simplicial complexes induces a chain map $\bar{\varphi}$ between the corresponding chain complexes. This construction is *functorial*, meaning that for φ the identity function on a simplicial complex K, $\bar{\varphi}$ is the identity function on $\mathcal{C}_*(K)$, and for composable simplicial maps φ, φ', we have that $\overline{\varphi \circ \varphi'} = \bar{\varphi} \circ \bar{\varphi'}$.

Homology and carriers. The *p-th homology group* $H_p(\mathcal{C}_*)$ of a chain complex is defined as $\ker \partial_p / \operatorname{im} \partial_{p+1}$. The p-th homology group of a simplicial complex K, $H_p(K)$, is the p-th homology group of its induced chain complex. In either case $H_p(\mathcal{C}_*)$ is a $\mathcal{F}$-vector space because we have chosen our base ring $\mathcal{F}$ as a field. Intuitively, when the chain complex is generated from a simplicial complex, the dimension of the p-th homology group counts the number of p-dimensional holes in the complex (except for $p = 0$, where it counts the number of connected components). We write $H(\mathcal{C}_*)$ for the direct sum of all $H_p(\mathcal{C}_*)$ for $p \geq 0$.

A chain map $\phi : \mathcal{C}_* \to D_*$ induces a linear map $\phi^* : H(\mathcal{C}_*) \to H(D_*)$ between the homology groups. Again, this construction is functorial, meaning that it maps identity maps to identity maps, and it is compatible with compositions.

We call a simplicial complex K *acyclic*, if K is connected and all homology groups $H_p(K)$ with $p \geq 1$ are trivial. For simplicial complexes K and L, an *acyclic carrier* Φ is a map that assigns to each simplex σ in K, a non-empty subcomplex $\Phi(\sigma) \subseteq L$ such that $\Phi(\sigma)$ is acyclic, and whenever τ is a subsimplex of σ, then $\Phi(\tau) \subseteq \Phi(\sigma)$. We say that a chain $c \in \mathcal{C}_p(K)$ is *carried* by a subcomplex K', if c takes value 0 except for p-simplices in K'. A chain map $\phi : \mathcal{C}_*(K) \to \mathcal{C}_*(L)$ is *carried by* Φ, if for each simplex $\sigma \in K$, $\phi(\sigma)$ is carried by $\Phi(\sigma)$. We state the *acyclic carrier theorem* [15]:

▶ **Theorem 1.** *Let $\Phi : K \to L$ be an acyclic carrier.*
- *There exists a chain map $\phi : \mathcal{C}_*(K) \to \mathcal{C}_*(L)$ such that ϕ is carried by Φ.*
- *If two chain maps $\phi_1, \phi_2 : \mathcal{C}_*(K) \to \mathcal{C}_*(L)$ are both carried by Φ, then $\phi_1^* = \phi_2^*$.*

Filtrations and towers. Let $I \subseteq \mathbb{R}$ be a set of real values which we refer to as *scales*. A *filtration* is a collection of simplicial complexes $(K_\alpha)_{\alpha \in I}$ such that $K_\alpha \subseteq K'_\alpha$ for all $\alpha \leq \alpha' \in I$. For instance, $(\mathcal{R}_\alpha)_{\alpha \geq 0}$ is a filtration which we call the *Rips filtration*. A *(simplicial) tower* is a sequence $(K_\alpha)_{\alpha \in J}$ of simplicial complexes with J being a discrete set (for instance $J = \{2^k \mid k \in \mathbb{Z}\}$), together with simplicial maps $\varphi_\alpha : K_\alpha \to K_{\alpha'}$ between complexes at consecutive scales. For instance, the Rips filtration can be turned into a tower by restricting to a discrete range of scales, and using the inclusion maps as φ. The approximation constructed in this paper will be another example of a tower.

We say that a simplex σ is *included* in the tower at scale α', if σ is not the image of $\varphi_\alpha : K_\alpha \to K_{\alpha'}$, where α is the scale preceding α' in the tower. The *size* of a tower is the number of simplices included over all scales. If a tower arises from a filtration, its size is simply the size of the largest complex in the filtration (or infinite, if no such complex exists).

[4] To avoid thinking about orientations, it is often assumed that $\mathcal{F} = \mathbb{Z}_2$ is the field with two elements.

However, this is not true for in general for simplicial towers, since simplices can collapse in the tower and the size of the complex at a given scale may not take into account the collapsed simplices which were included at earlier scales in the tower.

Barcodes and Interleavings. A collection of vector spaces $(V_\alpha)_{\alpha\in I}$ connected with linear maps $\lambda_{\alpha_1,\alpha_2} : V_{\alpha_1} \to V_{\alpha_2}$ is called a *persistence module*, if $\lambda_{\alpha,\alpha}$ is the identity on V_α and $\lambda_{\alpha_2,\alpha_3} \circ \lambda_{\alpha_1,\alpha_2} = \lambda_{\alpha_1,\alpha_3}$ for all $\alpha_1 \le \alpha_2 \le \alpha_3 \in I$ for the index set I.

We generate persistence modules using the previous concepts. Given a simplicial tower $(K_\alpha)_{\alpha\in I}$, we generate a sequence of chain complexes $(\mathcal{C}_*(K_\alpha))_{\alpha\in I}$. By functoriality, the simplicial maps φ of the tower give rise to chain maps $\overline{\varphi}$ between these chain complexes. Using functoriality of homology, we obtain a sequence $(H(K_\alpha))_{\alpha\in I}$ of vector spaces with linear maps $\overline{\varphi}^*$, forming a persistence module. The same construction can be applied to filtrations.

Persistence modules admit a decomposition into a collection of intervals of the form $[\alpha, \beta]$ (with $\alpha, \beta \in I$), called the *barcode*, subject to certain tameness conditions. The barcode of a persistence module characterizes the module uniquely up to isomorphism. If the persistence module is generated by a simplicial complex, an interval $[\alpha, \beta]$ in the barcode corresponds to a homological feature (a "hole") that comes into existence at complex K_α and persists until it disappears at K_β.

Two persistence modules $(V_\alpha)_{\alpha\in I}$ and $(W_\alpha)_{\alpha\in I}$ with linear maps $\lambda_{\cdot,\cdot}$ and $\mu_{\cdot,\cdot}$ are said to be *weakly (multiplicatively) c-interleaved* with $c \ge 1$, if there exist linear maps $\gamma_\alpha : V_\alpha \to W_{c\alpha}$ and $\delta_\alpha : W_\alpha \to V_{c\alpha}$, called *interleaving maps*, such that the diagram

$$\begin{array}{ccccccccc} \cdots & \longrightarrow & V_{\alpha c} & & \xrightarrow{\lambda} & & V_{\alpha c^3} & \longrightarrow & \cdots \\ & & & \overset{\delta}{\nearrow} & & \overset{\gamma}{\searrow} & & \overset{\delta}{\nearrow} & \\ \cdots & \longrightarrow & W_\alpha & & \xrightarrow{\mu} & & W_{\alpha c^2} & \longrightarrow & \cdots \end{array} \tag{1}$$

commutes for all $\alpha \in I$, that is, $\mu = \gamma \circ \delta$ and $\lambda = \delta \circ \gamma$ (we have skipped the subscripts of the maps for readability). In such a case, the barcodes of the two modules are $3c$-approximations of each other in the sense of [6]. We say that two towers are *c-approximations* of each other, if their persistence modules that are c-approximations. Under the more stringent conditions of *strong interleaving*, the approximation ratio can be improved. See [7] for more details.

3 Grids and cubes

Let $I := \{\lambda 2^s \mid s \in \mathbb{Z}\}$ with $\lambda > 0$ be a discrete set of scales. For a scale $\alpha_s := \lambda 2^s$, we inductively define a grid G_s on scale α_s which is a scaled and translated (shifted) version of the integer lattice: for $s = 0$, G_s is simply $\lambda\mathbb{Z}^d$, the scaled integer grid. For $s \ge 0$, we choose an arbitrary $O \in G_s$ and define

$$G_{s+1} = 2(G_s - O) + O + \frac{\alpha_s}{2}(\pm 1, \ldots, \pm 1) \tag{2}$$

where the signs of the components of the last vector are chosen uniformly at random (and the choice is independent for each s). For $s \le 0$, we define

$$G_{s-1} = \frac{1}{2}(G_s - O) + O + \frac{\alpha_{s-1}}{2}(\pm 1, \ldots, \pm 1). \tag{3}$$

It is then easy to check that 2 and 3 are consistent at $s = 0$. A simple instance of the above construction is the sequence of lattices with $G_s := \alpha_s\mathbb{Z}^d$ for even s, and $G_s := \alpha_s\mathbb{Z}^d + \frac{\alpha_{s-1}}{2}(1, \ldots, 1)$ for odd s.

We motivate the shifting next. For a finite point set $Q \subset \mathbb{R}^d$ and $x \in Q$, the *Voronoi region* $Vor_Q(x) \subset \mathbb{R}^d$ is the (closed) set of points in $\mathbb{R}^d$ that have x as one of its closest points in Q. If $Q = G_s$, it is easy to see that the Voronoi region of any grid point x is a cube of side length α_s centered at x. The shifting of the grids ensures that each $x \in G_s$ lies in the Voronoi region of a unique $y \in G_{s+1}$. By an elementary calculation, we show a stronger statement; for shorter notation, we write $Vor_s(x)$ instead of $Vor_{G_s}(x)$.

▶ **Lemma 2.** *Let $x \in G_s, y \in G_{s+1}$ such that $x \in Vor_{s+1}(y)$. Then, $Vor_s(x) \subset Vor_{s+1}(y)$.*

Cubical complexes. The integer grid $\mathbb{Z}^d$ naturally defines a *cubical complex*, where each element is an axis-aligned, k-cube with $0 \leq k \leq d$. Let $\square$ denote the set of all integer translates of faces of the unit cube $[0,1]^d$, considered as a convex polytope in $\mathbb{R}^d$. We call the elements of $\square$ *faces*. Each face has a dimension k; the 0-faces, or *vertices* are exactly the points in $\mathbb{Z}^d$. The *facets* of a k-face f are the $(k-1)$-faces contained in f. We call a pair of facets of f *opposite* if they are disjoint. Obviously, these concepts carry over to scaled and translated versions of $\mathbb{Z}^d$, so we define $\square_s$ as the cubical complex defined by G_s.

We define a map $g_s : \square_s \to \square_{s+1}$ as follows: for vertices, we assign to $x \in G_s$ the (unique) vertex $y \in G_{s+1}$ such that $x \in Vor_{s+1}(y)$ (cf. Lemma 2). For a k-face f of $\square_s$ with vertices $(p_1, \ldots, p_{2^k})$ in G_s, we set $g_s(f)$ to be the convex hull of $\{g_s(p_1), \ldots, g_s(p_{2^k})\}$; the next lemma shows that this is indeed a well-defined map (see [7]).

▶ **Lemma 3.** *$\{g_s(p_1), \ldots, g_s(p_{2^k})\}$ are the vertices of a face e of G_{s+1}. Moreover, if e_1, e_2 are any two opposite facets of e, then there exists a pair of opposite facets f_1, f_2 of f such that $g_s(f_1) = e_1$ and $g_s(f_2) = e_2$.*

Barycentric subdivision. A *flag* in $\square_s$ is a set of faces $\{f_0, \ldots, f_k\}$ of $\square_S$ such that $f_0 \subseteq \ldots \subseteq f_k$. The *barycentric subdivision* sd_s of $\square_s$ is the (infinite) simplicial complex whose simplices are the flags of $\square_s$; in particular, the 0-simplices of sd_s are the faces of $\square_s$. An equivalent geometric description of sd_s can be obtained by defining the 0-simplices as the barycenters of the faces in sd_s, and introducing a k-simplex between $(k+1)$ barycenters if the corresponding faces form a flag. It is easy to see that sd_s is a flag complex. Given a face f in $\square_s$, we write $sd(f)$ for the subcomplex of sd_s consisting of all flags that are formed only by faces contained in f.

4 Approximation scheme

We define our approximation complex at scale α_s as a finite subcomplex of sd_s. To simplify the subsequent analysis, we define the approximation in a slightly generalized form.

Barycentric spans. For a fixed s, let V denote a non-empty subset of G_s. We say that a face $f \in \square_s$ is *spanned* by V if $f \cap V \neq \varnothing$ and is not contained in any facet of f. Trivially, the vertices of $\square_s$ spanned by V are precisely the points in V. We point out that the set of spanned faces is *not* closed under taking sub-faces; for instance, if V consists of two antipodal points of a d-cube, the only faces spanned by V are the d-cube and the two vertices.

The *barycentric span* of V is the subcomplex of sd_s defined by all flags $\{f_0, \ldots, f_k\}$ such that all f_i are spanned by V. This is indeed a subcomplex of sd_s because it is closed under taking subsets. Moreover, for a face $f \in \square_k$, we define the *f-local barycentric span* of V as the set of all flags $\{f_0, \ldots, f_k\}$ in the barycentric span such that $f_i \subseteq f$ for all i. This is a subcomplex both of $sd(f)$ and of the barycentric span of V and is a flag complex.

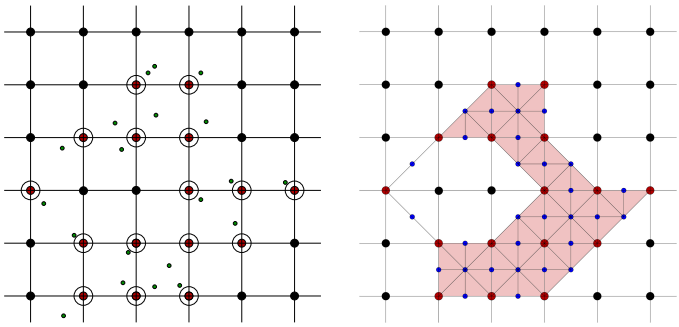


Figure 1 The left figure shows a two-dimensional grid, along with its cubical complex. The green points (small dots) denote the points in P and the red vertices (encircled) are the active vertices. The figure on the right shows the generated simplicial complex. The blue vertices (small dots) are the barycenters of the active faces.

▶ **Lemma 4.** *For each face f, the f-local barycentric span of V is either empty or acyclic.*

Furthermore, if $W \subseteq V$, it is easy to see that faces spanned by W are also spanned by V. Consequently, the barycentric span of W is a subcomplex of the barycentric span of V.

Approximation complex. We denote by $P \subset \mathbb{R}^d$ a finite set of points. For each point $p \in P$, we let $a_s(p)$ denote the grid point in G_s that is closest to p (we assume for simplicity that this closest point is unique). We define the *active vertices of* G_s, V_s, as $a_s(P)$, that is, the set of grid points that are closest to some point in P. The next statement is a direct application of the triangle inequality; let diam_∞ denote the diameter in the L_∞-norm.

▶ **Lemma 5.** *Let $Q \subseteq P$ be such that $\mathrm{diam}_\infty(Q) \leq \alpha_s$. Then, the set $a_s(Q)$ is contained in a face of $\square_s$. Equivalently, for a simplex $\sigma = (p_0, \ldots, p_k) \in \mathcal{R}^\infty_{\alpha_s}$ on P, the set of active vertices $\{a_s(p_0), \ldots, a_s(p_k)\}$ is contained in a face of $\square_s$.*

Vice versa, we define a map $b_s : V_s \to P$ by mapping an active vertex to its closest point in P (again, assuming for simplicity that the assignment is unique). The map b_s is a section of a_s, that is, $a_s \circ b_s$ is the identity on V_s.

Recall that the map $g_s : \square_s \to \square_{s+1}$ from Section 3 maps grid points of G_s to grid points of G_{s+1}. With Lemma 2, it follows at once:

▶ **Lemma 6.** *For all $x \in V_s$, $g_s(x) = (a_{s+1} \circ b_s)(x)$.*

We now define our approximation tower: for scale α_s, we define $\mathcal{X}_{\alpha_s}$ as the barycentric span of the active vertices $V_s \subset G_s$. See Figure 1 for an illustration. To simplify notations, we call the faces of $\square_s$ spanned by V_s *active faces*, and simplices of $\mathcal{X}_{\alpha_s}$ *active flags*.

To complete the construction, we need to define simplicial maps $\mathcal{X}_{\alpha_s} \to \mathcal{X}_{\alpha_{s+1}}$. First, we show:

▶ **Lemma 7.** *Let f be an active face of $\square_s$. Then, $g_s(f)$ is an active face of $\square_{s+1}$.*

Proof. From Lemma 3, $e := g_s(f)$ is a face of G_{s+1}. If e is a vertex, it is active, because f contains at least one active vertex v, and $g_s(v) = e$ in this case. If e is not a vertex, we assume for a contradiction that it is not active. Then, it contains a facet e_1 that contains all active vertices in e. Let e_2 denote the opposite facet. By Lemma 3, f contains opposite facets f_1, f_2 such that $g_s(f_1) = e_1$ and $g_s(f_2) = e_2$. Since f is active, both f_1 and f_2 contain active vertices, in particular, f_2 contains an active vertex v. But then, the active vertex $g_s(v)$ must lie in e_2, contracting the fact that e_1 contains all active vertices of e. ◀

Recall that a simplex $\sigma \in \mathcal{X}_{\alpha_s}$ is a flag $f_0 \subseteq \ldots \subseteq f_k$ of active faces in $\square_s$. We set $\tilde{g}(\sigma)$ as the flag $g(f_0) \subseteq \ldots \subseteq g(f_k)$, which consists of active faces in $\square_{s+1}$ by Lemma 7, and hence is a simplex in $\mathcal{X}_{\alpha_{s+1}}$. It follows that $\tilde{g} : \mathcal{X}_{\alpha_s} \to \mathcal{X}_{\alpha_{s+1}}$ is a simplicial map. This finishes our construction of the simplicial tower $(\mathcal{X}_{\lambda 2^s})_{s\in\mathbb{Z}}$, with simplicial maps $\tilde{g} : \mathcal{X}_{\lambda 2^s} \to \mathcal{X}_{\lambda 2^{s+1}}$.

4.1 Interleaving

To relate our tower with the L_∞-Rips filtration, we start by defining two acyclic carriers. We write $\alpha := \alpha_s = \lambda 2^s$ to simplify notations.

- $C_1 : \mathcal{R}^\infty_\alpha \to \mathcal{X}_\alpha$: let $\sigma = (p_0, \ldots, p_k)$ be any simplex of $\mathcal{R}^\infty_\alpha$. We set $C_1(\sigma)$ as the barycentric span of $U := \{a_s(p_0), \ldots, a_s(p_k)\}$, which is a subcomplex of $\mathcal{X}_\alpha$. U lies in a face f of $\square_s$ by Lemma 5 hence $C_1(\sigma)$ is also the f-local barycentric span of U. Using Lemma 4, $C_1(\sigma)$ is acyclic.
- $C_2 : \mathcal{X}_\alpha \to \mathcal{R}^\infty_{2\alpha}$: let σ be any flag $e_0 \subseteq \ldots \subseteq e_k$ of $\mathcal{X}_\alpha$. Let $\{q_0, \ldots, q_m\}$ be the set of active vertices of e_k. We set $C_2(\sigma) := \{b_s(q_0), \ldots, b_s(q_m)\}$. With a simple triangle inequality, we see that $C_2(\sigma)$ is a simplex in $\mathcal{R}^\infty_{2\alpha}$, hence it is acyclic.

Using the Acyclic Carrier Theorem (Theorem 1), there exist chain maps $c_1 : \mathcal{C}_*(\mathcal{R}^\infty_\alpha) \to \mathcal{C}_*(\mathcal{X}_\alpha)$ and $c_2 : \mathcal{C}_*(\mathcal{X}_\alpha) \to \mathcal{C}_*(\mathcal{R}^\infty_{2\alpha})$, which are carried by C_1 and C_2, respectively. Aggregating the chain maps, we have the following diagram:

$$\begin{array}{ccccccc} \cdots & \longrightarrow & \mathcal{C}_*(\mathcal{R}^\infty_{2\alpha}) & \xrightarrow{inc} & \mathcal{C}_*(\mathcal{R}^\infty_{4\alpha}) & \longrightarrow & \cdots \\ & & \uparrow{\scriptstyle c_2} \quad \downarrow{\scriptstyle c_1} & & \uparrow{\scriptstyle c_2} & & \\ \cdots & \longrightarrow & \mathcal{C}_*(\mathcal{X}_\alpha) & \xrightarrow{\tilde{g}} & \mathcal{C}_*(\mathcal{X}_{2\alpha}) & \longrightarrow & \cdots \end{array} \tag{4}$$

where *inc* corresponds to the inclusion chain map and $\tilde{g}$ denotes the chain map for the corresponding simplicial maps (we removed indices for readability). The chain complexes give rise to a diagram of the corresponding homology groups, connected by the induced linear maps $c_1^*, c_2^*, inc^*, \tilde{g}^*$.

▶ **Lemma 8.** *$inc^* = c_2^* \circ c_1^*$ and $\tilde{g}^* = c_1^* \circ c_2^*$. In particular, the persistence modules $(H(\mathcal{X}_{2^s}))_{s\in\mathbb{Z}}$ and $(H(\mathcal{R}^\infty_\alpha))_{\alpha\geq 0}$ are weakly 2-interleaved.*

Proof. To prove the claim, we consider both triangles separately. We show that the chain maps $\tilde{g}$ and $c_1 \circ c_2$ are carried by a common acyclic carrier. Then we show the same statement for *inc* and $c_2 \circ c_1$. The claim then follows from the Acyclic Carrier Theorem.

- *Lower triangle*: The map $C_1 \circ C_2 : \mathcal{X}_\alpha \to \mathcal{X}_{2\alpha}$ is an acyclic carrier, because $C_2(\sigma)$ is a simplex for any simplex $\sigma \in \mathcal{X}_\alpha$. Clearly, $C_1 \circ C_2$ carries the map $c_1 \circ c_2$. We show that it also carries $\tilde{g}$.
  Let σ be a flag $f_0 \subseteq \ldots \subseteq f_k$ in $\mathcal{X}_\alpha$ and let $V(f_i)$ denote the active vertices of f_i. Then, $C_1 \circ C_2(\sigma)$ is the barycentric span of $U := \{a_{s+1} \circ b_s(q) \mid q \in V(f_k)\} = \{g_s(q) \mid q \in V(f_k)\}$ (Lemma 6). On the other hand, $V(f_i) \subseteq V(f_k)$ and hence $g(V(f_i)) \subseteq U$. Then, $g(f_i)$ is spanned by U: indeed, since f_i is active, $g(f_i)$ is active and hence spanned by all active vertices, and it remains spanned if we remove all active vertices not in U, since they are not contained in f_i. It follows that the flag $g(f_0) \subseteq \ldots \subseteq g(f_k)$, which is equal to $\tilde{g}(\sigma)$, is in the barycentric span of U.
- *Upper triangle*: We define an acyclic carrier $D : \mathcal{R}^\infty_{2\alpha} \to \mathcal{R}^\infty_{4\alpha}$ which carries both *inc* and $c_2 \circ c_1$. Let $\sigma = (p_0, \ldots, p_k) \in \mathcal{R}^\infty_{2\alpha}$ be a simplex. The active vertices $U := \{a(p_0), \ldots, a(p_k)\} \subset G_{s+1}$ lie in a face f of $G_{2\alpha}$, using Lemma 5. We can assume that f is active, as otherwise, we pass to a facet of f that contains U. We set $D(\sigma)$ as the

simplex on the subset of points in P whose closest grid point in G_{s+1} lies in U. Using a simple application of triangle inequalities, $D(\sigma) \in \mathcal{R}^{\infty}_{4\alpha}$, so D is an acyclic carrier. The 0-simplices of σ are a subset of $D(\sigma)$, so D carries the map *inc*. We next show that D carries $c_2 \circ c_1$.

Let δ be a simplex in $\mathcal{X}_{2\alpha}$ for which the chain $c_1(\sigma)$ takes a non-zero value. Since $c_1(\sigma)$ is carried by $C_1(\sigma)$, $\delta \in C_1(\sigma)$ which is the barycentric span of $V(f)$. Furthermore, for any $\tau \in C_1(\sigma)$, $C_2(\tau)$ is of the form $\{b(q_0), \ldots, b(q_m)\}$ with $\{q_0, \ldots, q_m\} \in V(f)$. It follows that $C_2(\tau) \subseteq D(\sigma)$. In particular, since c_2 is carried by C_2, $c_2(c_1(\sigma)) \subseteq D(\sigma)$ as well. ◀

4.2 Scale balancing

We improve the approximation factor with a simple modification. Let $(A_{\lambda\gamma^k})_{k\in\mathbb{Z}}$ and $(B_{\lambda\gamma^k})_{k\in\mathbb{Z}}$ be two simplicial towers with simplicial maps f_3 and f_4 respectively, with $\lambda, \gamma > 0$. Assume that there exist interleaving linear maps f_1^*, f_2^* such that the diagram

$$\begin{array}{ccccccccc} \cdots & \longrightarrow & H(B_{\alpha\gamma}) & \xrightarrow{f_4^*} & H(B_{\alpha\gamma^2}) & \longrightarrow & \cdots & & \\ & \nearrow^{f_1^*} & \downarrow{\scriptstyle f_2^*} & \nearrow^{f_1^*} & & & & & \\ \cdots & \longrightarrow & H(A_\alpha) & \xrightarrow{f_3^*} & H(A_{\alpha\gamma}) & \longrightarrow & \cdots & & \end{array} \tag{5}$$

commutes for all scales $\alpha = \lambda\gamma^k$, which implies that the persistence modules are weakly γ-interleaved. Defining another tower $(A'_{\lambda\sqrt{\gamma}\gamma^k})_{k\in\mathbb{Z}}$ with $A'_\alpha := A_{\alpha/\sqrt{\gamma}}$, we obtain a diagram

$$\begin{array}{ccccccccc} \cdots & \longrightarrow & H(B_{\alpha\gamma}) & & \xrightarrow{f_4^*} & & H(B_{\alpha\gamma^2}) & \longrightarrow & \cdots \\ & \nearrow^{f_1^*} & & \searrow^{f_2^*} & & \nearrow^{f_1^*} & & & \\ \cdots & \longrightarrow & H(A'_{\alpha\sqrt{\gamma}}) & & \xrightarrow{f_3^*} & & H(A'_{\alpha\gamma\sqrt{\gamma}}) & \longrightarrow & \cdots \end{array} \tag{6}$$

which implies that the persistence modules are weakly $\sqrt{\gamma}$-interleaved. Therefore, scale balancing improves the interleaving ratio by only scaling the persistence module.

In our context, we improve the weak 2-interleaving of $(H(\mathcal{X}_{2^k\alpha}))_{k\in\mathbb{Z}}$ and $(H(\mathcal{R}^{\infty}_{\alpha}))_{\alpha\geq 0}$ to a weak $\sqrt{2}$-interleaving. Using the proximity results for persistence modules [6],

▶ **Theorem 9.** *The persistence module* $\left(H(\mathcal{X}_{2^k/\sqrt{2}})\right)_{k\in\mathbb{Z}}$ *is a* $3\sqrt{2}$*- approximation of the* L_∞*-Rips persistence module* $\left(H(\mathcal{R}^{\infty}_{\alpha})\right)_{\alpha\geq 0}$.

For any pair of points $p, p' \in \mathbb{R}^d$, it holds that $\|p - p'\|_2 \leq \|p - p'\|_\infty \leq \sqrt{d}\,\|p - p'\|_2$ which implies that the L_2- and the L_∞-Rips complexes are strongly $\sqrt{d}$-interleaved. The scale balancing technique also works for strongly interleaved persistence modules and yields

▶ **Lemma 10.** $(H(\mathcal{R}_{\alpha/d^{0.25}}))_{\alpha\geq 0}$ *is strongly* $d^{0.25}$*-interleaved with* $(H(\mathcal{R}^{\infty}_{\alpha}))_{\alpha\geq 0}$.

Using Theorem 9, Lemma 10 and the fact that interleavings satisfy the triangle inequality [3, Theorem 3.3], we see that $(H(\mathcal{X}_{2^k/\sqrt{2}}))_{k\in\mathbb{Z}}$ is weakly $\sqrt{2}d^{0.25}$-interleaved with the scaled Rips module $(H(\mathcal{R}_{\alpha/d^{0.25}}))_{\alpha\geq 0}$. We can remove the scaling in the Rips filtration simply by multiplying both sides with $d^{0.25}$ and obtain our final approximation result.

▶ **Theorem 11.** *The persistence module* $\left(H(\mathcal{X}_{\frac{2^k \sqrt[4]{d}}{\sqrt{2}}})\right)_{k\in\mathbb{Z}}$ *is a* $3\sqrt{2}d^{0.25}$*-approximation of the Euclidean Rips persistence module* $\left(H_*(\mathcal{R}_\alpha)\right)_{\alpha\geq 0}$.

5 Size and computation

Set $n := |P|$ and let $CP(P)$ denote the closest pair distance of P. At scale $\alpha_0 := \frac{CP(P)}{3d}$ and lower, no d-cube of the cubical complex contains more than one active vertex, so the approximation complex consists of n isolated 0-simplices. At scale $\alpha_m := diam(P)$ and higher, points of P map to active vertices of a common face by Lemma 5, so the generated complex is acyclic using Lemma 4. We inspect the range of scales $[\alpha_0, \alpha_m]$ to construct the tower, since the barcode is explicitly known for scales outside this range. The total number of scales is $\lceil \log_2 \alpha_m/\alpha_0 \rceil = \lceil \log_2 \Delta + \log_2 3d \rceil = O(\log \Delta + \log d)$.

5.1 Size of the tower

Recall that the size of a tower is the number of simplices that do not have a preimage. We start by considering the case of 0-simplices.

▶ **Lemma 12.** *The number of* 0*-simplices included in the tower is at most* $n2^{O(d)}$.

The proof can be summarized as follows: 0-simplices in the tower correspond to active faces. Active vertices are only added at the lowest scale, hence they account for n inclusions. Active faces of higher dimensions have at least one active vertex on their boundary. We charge the inclusion of such a face to one point in P that is "close" to the face. In this way, we show that every point in P is charged at most $2^{O(d)}$ times. See [7] for further details.

The next lemma follows from a simple combinatorial counting argument for the number of flags in a d-dimensional cube (see [7]).

▶ **Lemma 13.** *Each* 0*-simplex of* $\mathcal{X}_\alpha$ *has at most* $2^{O(d \log k)}$ *incident* k*-simplices.*

▶ **Theorem 14.** *The* k*-skeleton of the tower has size at most* $n2^{O(d \log k)}$.

Proof. Let $\sigma = f_0 \subseteq \ldots \subseteq f_k$ be a flag included at some scale α. The crucial insight is that this can only happen if at least one face f_i in the flag is included in the tower at the same scale. Indeed, if each f_i has a preimage e_i on the previous scale, then $e_0 \subseteq \ldots \subseteq e_k$ is a flag on the previous scale which maps to σ under $\tilde{g}$.

We charge the inclusion of the flag to the inclusion of f_i. By Lemma 13, the 0-simplex f_i of $\mathcal{X}$ is charged at most $\sum_{i=1}^{k} 2^{O(d \log i)} = 2^{O(d \log k)}$ times in this way, and by Lemma 12, there are at most $n2^{O(d)}$ 0-simplices that can be charged. ◀

5.2 Computing the tower

Recall from the construction of the grids that G_{s+1} is built from G_s using an arbitrary translation vector $(\pm 1, \ldots, \pm 1) \in \mathbb{Z}^d$. In our algorithm, we pick the components of this translation vector uniformly at random, and independently for each scale.

Recall the cubical map $g_s : \square_s \to \square_{s+1}$ from Section 3. For a fixed s, we denote by $g^{(j)} : \square_s \to \square_{s+j}$ the j-fold composition of g, that is $g^{(j)} = g_{s+j-1} \circ g_{s+j-2} \circ \ldots \circ g_s$.

▶ **Lemma 15.** *For a* k*-face* f *of* $\square_s$*, let* Y *be the minimal integer* j *such that* $g^{(j)}(f)$ *is a vertex. Then* $E[Y] \leq 3 \log k$.

The proof idea is as follows. A k-face has a non-zero length in k coordinate direction. In order to map to a point, $g^{(j)}(f)$ has to "collapse" all these dimensions. For a fixed direction x_i, such a collapse happens for $g(f)$ if the random translation moves a grid point in the x_i-range of the face, which happens for exactly half of the translations (depending on the sign

at the i-position of the translation vector). The number of steps for which the x_i-direction is not collapsed is thus equivalent to the number of flips of a fair coin until heads shows for the first time, which is 2 in expectation. The entire k-face is collapsed to a point if k coins flipped simultaneously all have shown heads at least once. This takes at most $3 \log k$ steps in expectation. See [7] for details.

As a consequence of the lemma, the expected "lifetime" of k-simplices in our tower with $k > 0$ is rather short: given a flag $e_0 \subseteq \ldots \subseteq e_\ell$, the face e_ℓ will be mapped to a vertex after $O(\log d)$ steps, and so will be all its sub-faces, turning the flag into a vertex. It follows that the total number of k-simplices in the tower is upper bounded by $n2^{O(d \log k)}$ as well.

Algorithm description. We first specify what it means to "compute" the tower. We make use of the fact that a simplicial map between simplicial complexes can be written as a composition of simplex inclusions and contractions of 0-simplices [9, 12]. That is, when passing from a scale α_s to α_{s+1}, it suffices to specify which pairs of 0-simplices in $\mathcal{X}_{\alpha_s}$ are mapped to the same image under $\tilde{g}$ and which simplices in $\mathcal{X}_{\alpha_{s+1}}$ are included.

The input is a set of n points $P \subset \mathbb{R}^d$. The output is a list of *events*, where each event is of one of the three following types: a *scale event* defines a real value α and signals that all upcoming events happen at scale α (until the next scale event). An *inclusion event* introduces a new simplex, specified by the list of 0-simplices on its boundary (we assume that every 0-simplex is identified by an integer). A *contraction event* is a pair of 0-simplices (i, j) and signifies that i and j are identified as the same from that scale.

In a first step, we calculate the range of scales that we are interested in. We compute a 2-approximation of $diam(P)$ by taking any point $p \in P$ and calculating $max_{q \in P} \|p - q\|$. Then we compute $CP(P)$ using a randomized algorithm in $n2^{O(d)}$ expected time [14].

Next, we proceed scale-by-scale and construct the list of events accordingly. On the lowest scale, we simply compute the active vertices by point location for P in a cubical grid, and enlist n inclusion events (this is the only step where the input points are considered in the algorithm). We use an auxiliary container S and maintain the invariant that whenever a new scale is considered, S consists of all simplices of the previous scale, sorted by dimension. In S, for each 0-simplex, we store an id and a coordinate representation of the active face to which it corresponds. Every ℓ-simplex with $\ell > 0$ is stored just as a list of integers, denoting its boundary 0-simplices. We initialize S with the n 0-simplices at the lowest scale.

Let $\alpha < \alpha'$ be any two consecutive scales with $\square, \square'$ the respective cubical complexes and $\mathcal{X}, \mathcal{X}'$ the approximation complexes, with $\tilde{g} : \mathcal{X} \to \mathcal{X}'$ being the simplicial map connecting them. Suppose we have already constructed all events at scale α. We enlist the scale event for α'. Then, we enlist the contraction events. For that, we iterate through the 0-simplices of $\mathcal{X}$ and compute their value under g, using point location in a cubical grid. We store the results in a list S' (which contains the simplices of $\mathcal{X}'$). If for a 0-simplex j, $g(j)$ is found to be equal to $g(i)$ for a previously considered 0-simplex, we choose the minimal such i and enlist a contraction event for i and j.

We turn to the inclusion events and start with the case of 0-simplices. Every 0-simplex is an active face at scale α' and must contain an active vertex, which is also a 0-simplex of $\mathcal{X}'$. We iterate through the elements in S'. For each active vertex v encountered, we go over all faces of the cubical complex $\square'$ that contain v as vertex and check whether they are active. For every active face encountered that is not in S' yet, we add it to S' and enlist an inclusion event of a new 0-simplex. At termination, all 0-simplices of $\mathcal{X}'$ have been detected.

Next, we iterate over the simplices of S of dimension ≥ 1 and compute their image under $\tilde{g}$, and store the result in S'. To find the simplices of dimension ≥ 1 included at $\mathcal{X}'$,

we exploit our previous insight that they contain at least one 0-simplex that is included at the same scale (see the proof of Theorem 14). Hence, we iterate over the 0-simplices included in $\mathcal{X}'$ and proceed inductively in dimension. Let v be the current 0-simplex under consideration; assume that we have found all $(p-1)$-simplices in $\mathcal{X}'$ that contain v. Each such $(p-1)$-simplex σ is a flag in $\square'$. We iterate over all faces e that extend σ to a flag of length $p+1$. If e is active, we found a p-simplex in $\mathcal{X}'$. If this simplex is not in S' yet, we add it and enlist an inclusion event for it. We also enqueue the simplex in our inductive procedure, to look for $(p+1)$-simplices in the next iteration. At the end of the procedure, we have detected all simplices in $\mathcal{X}'$ without preimage, and S' contains all simplices of $\mathcal{X}'$. We set $S \leftarrow S'$ and proceed to the next scale. This ends the description of the algorithm.

▶ **Theorem 16.** *To compute the k-skeleton, the algorithm takes time $\left(n2^{O(d)} \log \Delta + 2^{O(d)} M\right)$ time in expectation and M space, where M is the size of the tower. In particular, the expected time is bounded by $\left(n2^{O(d)} \log \Delta + n2^{O(d \log k)}\right)$ and the space is bounded by $n2^{O(d \log k)}$.*

The first summand of the time bound comes from the fact that on each scale, the number of 0-simplices of $\mathcal{X}$ is bounded by $n3^d$, and we employ local searches in the cubical complex to find 0-simplices included in $\mathcal{X}'$. This local search only causes an overhead of $O(2^d)$ per active vertex. The second summand arises because we find the higher-dimensional simplices of $\mathcal{X}'$ inductively and can therefore charge the cost for this search to the number of simplices encountered. Finally, computing the image of $\tilde{g}$ for all simplices in $\mathcal{X}$ can be bounded in expectation by $O(2^{O(d)}M)$, because the total size of all $\mathcal{X}$ in the algorithm is bounded by $O(\log dM)$ (see the remark after Lemma 15). More details are in [7].

6 Conclusion

We gave an approximation scheme for the Rips filtration, with improved approximation ratio, size and computational complexity than previous approaches for the case of high-dimensional point clouds. Moreover, we introduced the technique of using acyclic carriers to prove interleaving results. We point out that, while the proof of the interleaving in Section 4.1 is still technically challenging, it greatly simplifies by the usage of acyclic carriers; defining the interleaving chain maps explicitly significantly blows up the analysis. There is also no benefit in knowing the interleaving maps because they are only required for the analysis, not for the computation.

Our tower is connected by simplicial maps; there are (implemented) algorithms to compute the barcode of such towers [9, 12]. It is also quite easy to adapt our tower construction to a streaming setting [12], where the output list of events is passed to an output stream instead of being stored in memory.

An interesting question is whether persistence can be computed efficiently for more general chain maps, which would allow more freedom in building approximation schemes.

References

1 M. Botnan and G. Spreemann. Approximating Persistent Homology in Euclidean space through collapses. *Applied Algebra in Engineering, Communication and Computing*, 26(1-2):73–101, 2015.

2 P. Bubenik, V. de Silva, and J. Scott. Metrics for Generalized Persistence Modules. *Foundations of Computational Mathematics*, 15(6):1501–1531, 2015.

3 P. Bubenik and J.A. Scott. Categorification of Persistent Homology. *Discrete & Computational Geometry*, 51(3):600–627, 2014.

4 G. Carlsson. Topology and Data. *Bulletin of the American Mathematical Society*, 46:255–308, 2009.

5 N.J. Cavanna, M. Jahanseir, and D. Sheehy. A Geometric Perspective on Sparse Filtrations. In *Proceedings of the 27th Canadian Conference on Computational Geometry (CCCG 2015)*.

6 F. Chazal, D. Cohen-Steiner, M. Glisse, L.J. Guibas, and S.Y. Oudot. Proximity of Persistence Modules and their Diagrams. In *Proceedings of the ACM Symposium on Computational Geometry (SoCG 2009)*, pages 237–246, 2009.

7 A. Choudhary, M. Kerber, and S. Raghvendra. Improved approximate rips filtrations with shifted integer lattices. *CoRR*, abs/1706.07399, 2016. URL: `https://arxiv.org/abs/1706.07399`.

8 A. Choudhary, M. Kerber, and S. Raghvendra. Polynomial-sized topological approximations using the Permutahedron. In *Proceedings of the 32nd International Symposium on Computational Geometry (SoCG 2016)*, pages 31:1–31:16, 2016.

9 T.K. Dey, F. Fan, and Y. Wang. Computing Topological Persistence for Simplicial maps. In *Proceedings of the ACM Symposium on Computational Geometry (SoCG 2014)*, pages 345–354, 2014.

10 H. Edelsbrunner and J. Harer. *Computational Topology - An Introduction*. American Mathematical Society, 2010.

11 H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological Persistence and Simplification. *Discrete & Computational Geometry*, 28(4):511–533, 2002.

12 M. Kerber and H. Schreiber. Barcodes of Towers and a Streaming Algorithm for Persistent Homology. In *Proceedings of the 33rd International Symposium on Computational Geometry (SoCG)*, pages 57:1–57:15, 2017.

13 M. Kerber and R. Sharathkumar. Approximate Čech Complex in Low and High Dimensions. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 666–676, 2013.

14 S. Khuller and Y. Matias. A Simple Randomized Sieve Algorithm for the Closest-Pair Problem. *Information and Computation*, 118(1):34–37, 1995.

15 J.R. Munkres. *Elements of algebraic topology*. Westview Press, 1984.

16 D. Sheehy. Linear-size Approximations to the Vietoris-Rips Filtration. *Discrete & Computational Geometry*, 49(4):778–796, 2013.

The Sparse Awakens: Streaming Algorithms for Matching Size Estimation in Sparse Graphs

Graham Cormode[*1], **Hossein Jowhari**[†2], **Morteza Monemizadeh**[‡3], **and S. Muthukrishnan**[4]

1 **University of Warwick, UK**
g.cormode@warwick.ac.uk
2 **University of Warwick, UK**
H.Jowhari@warwick.ac.uk
3 **Amazon, Palo Alto, CA, USA**
mmorteza@amazon.com
4 **Rutgers University, Piscataway, NJ, USA**
muthu@cs.rutgers.edu

Abstract

Estimating the size of the maximum matching is a canonical problem in graph analysis, and one that has attracted extensive study over a range of different computational models. We present improved streaming algorithms for approximating the size of maximum matching with sparse (bounded arboricity) graphs.

- *(Insert-Only Streams)* We present a one-pass algorithm that takes $O(\alpha \log n)$ space and approximates the size of the maximum matching in graphs with arboricity α within a factor of $O(\alpha)$. This improves significantly upon the state-of-the-art $\tilde{O}(\alpha n^{2/3})$-space streaming algorithms, and is the first poly-logarithmic space algorithm for this problem.
- *(Dynamic Streams)* Given a dynamic graph stream (i.e., inserts and deletes) of edges of an underlying α-bounded arboricity graph, we present an one-pass algorithm that uses space $\tilde{O}(\alpha^{10/3} n^{2/3})$ and returns an $O(\alpha)$-estimator for the size of the maximum matching on the condition that the number edge deletions in the stream is bounded by $O(\alpha n)$. For this class of inputs, our algorithm improves the state-of-the-art $\tilde{O}(\alpha n^{4/5})$-space algorithms, where the $\tilde{O}(.)$ notation hides logarithmic in n dependencies.

In contrast to prior work, our results take more advantage of the streaming access to the input and characterize the matching size based on the ordering of the edges in the stream in addition to the degree distributions and structural properties of the sparse graphs.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases streaming algorithms, matching size

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.29

1 Introduction

In this paper, we address a core graph analysis question of finding the size of a maximum matching, using space asymptotically smaller than even the number of nodes. Graphs naturally capture relationships between entities, whether entities of the same type (simple

* Supported in part by European Research Council grant ERC-2014-CoG 647557, The Alan Turing Institute under the EPSRC grant EP/N510129/1, and a Royal Society Wolfson Research Merit Award.
† Supported by European Research Council grant ERC-2014-CoG 647557.
‡ Work was done when the author was at Rutgers University, Piscataway, NJ, USA.

graphs), of two types (bipartite graphs), or other combinations of types (encoded via multigraphs and hypergraphs). In modern applications, it is not uncommon to encounter graphs with many millions or billions of nodes (capturing the huge number of entities that can interact), and billions to trillions of edges (enumerating the vast number of possible interactions). This has led to significant interest in addressing traditional graph analysis problems in novel computational models: external memory, parallel and streaming models.

Problems related to (maximum) matchings in graph have a long history in Computer Science. They arise in many contexts, from choosing which advertisements to display to online users [34], to characterizing properties of chemical compounds [42]. Stable matchings have a suite of applications, from assigning students to universities, to arranging organ donations [41]. These have been addressed in a variety of different computational models, from the traditional RAM model, to more recent sublinear (property testing [38]) and external memory/parallel (e.g. MapReduce [25]) models. Matching has also been studied for a number of classes of input graph, including general graphs, bipartite graphs, weighted graphs, and those with some sparsity structure.

Our work focuses on the streaming case, where each edge is seen once only, and we are restricted to space sublinear in the size of the graph (ie., the number of vertices). This captures the scenario when the number of edges is overwhelmingly large, such as when analyzing connections between a massive number of individuals in a communication or social network. Now the objective is to find (approximately) the *size* of the matching. That is, while we cannot hope to retrieve a full description of the matching in sublinear space, we can hope to estimate how big the matching is. Even here, results for general graphs are either weak or make assumptions about the input or the stream order. In this work, we seek to improve the guarantees by restricting to graphs that have some measure of sparsity – bounded arboricity, or bounded degree. This aligns with reality, where most massive graphs have asymptotically fewer than $\Theta(n^2)$ edges. For example, in graphs that arise in the context of social networks, most nodes have a degree that is less than a few hundred, as people can only maintain this number of active connections (Dunbar's number), although a few nodes ("celebrities") have very high (in-)degree in the multi-millions.

Estimating the matching size for graphs in the streaming model has been the subject of some study in the algorithms and data analysis community in recent years. Kapralov, Khanna, and Sudan [21] developed a streaming algorithm which computes an estimate of matching size for general graphs within a factor of $O(\text{polylog}(n))$ in the *random-order* streaming model using $O(\text{polylog}(n))$ space. In the random-order model, the input stream is assumed to be chosen uniformly at random from the set of all possible permutations of the edges. Esfandiari *et al.* [15] were the first to study streaming algorithms for estimating the size of matching in bounded arboricity graphs in the *adversarial-order* streaming model, where the algorithm is required to provide a good approximation for any ordering of edges. Graph arboricity is a measure to quantify the density of a given graph. A graph $G(V, E)$ has arboricity α if the set E of its edges can be partitioned into at most α forests. Since a forest on n nodes has at most $n-1$ edges, a graph with arboricity α can have at most $\alpha(n-1)$ edges. Indeed, by a result of Nash-Williams [36, 37] this holds for any subgraph of a α-bounded arboricity graph G. Formally, the Nash-Williams Theorem [36, 37] states that

$$\alpha = \max_{U \subseteq V} \frac{|E(U)|}{(|U|-1)},$$

where $|U|$ and $|E(U)|$ are the number of nodes and edges in the subgraph induced by the nodes U, respectively. Several important families of graphs have constant arboricity. Examples

include planar graphs (that have arboricity $\alpha = 3$), bounded genus graphs, bounded treewidth graphs, and more generally, graphs that exclude a fixed minor.[1]

The important observation in [15] is that the size of matching in bounded arboricity graphs can be approximately characterized by the number of high degree vertices (vertices with degree above a fixed threshold) and the number of so-called *shallow edges* (edges with both low degree endpoints). This characterization allows for estimation of the matching size in sublinear space by taking samples from the vertices and edges of the graph. The work of [15] implements the characterization in $\tilde{O}(\alpha n^{2/3})$ space and gives a $O(\alpha)$ approximation of the matching size. Subsequent works [6, 30] consider alternative characterizations and improve upon the approximation factor however they do not result in major space improvements.

1.1 Our Contributions

We present major improvements in the space usage of streaming algorithms for sparse graphs (α-bounded Arboricity Graphs). Our main result is a polylog space algorithm that beats the n^{ε} space bound of prior algorithms. More precisely, we show:

▶ **Theorem 1.** *Let $G(V, E)$ be a graph with arboricity bounded by α. Let S be an (adversarial order) insertion-only stream of the edges of the underlying graph G. Let M^* be the size of the maximum matching of G (or S interchangeably). Then, there is a randomized 1-pass streaming algorithm that outputs a $(22.5\alpha + 6)(1 + \varepsilon)$-approximation to M^* with probability at least $1 - \delta$ and takes $O(\frac{\alpha}{\varepsilon^2} \log n)$ space.*

This result is notable, since it is the first demonstration that the polynomial space cost can be beaten for matching size estimation, and shows a that polylogarithmic space is sufficient for a constant factor approximation. Subsequent to our initial statement of results [10], McGregor and Vorotnikova have provided a new analysis of our algorithm to improve the constants in the approximation factor to achieve a $(\alpha + 2)(1 + \varepsilon)$ factor [32].

For the case of dynamic streams (i.e, streams of inserts and deletes of edges), we design a different algorithm using $\tilde{O}(\alpha^{10/3} n^{2/3})$ space which improves the $\tilde{O}(\alpha n^{4/5})$-space dynamic (insertion/deletion) streaming algorithms of [6, 7]. The following theorem states this result (proved in Section 3.3).

▶ **Theorem 2.** *Let $G(V, E)$ be a graph with the arboricity bounded by α. Let M^* be the size of the maximum matching of G. Let S be a dynamic stream of edge insertions and deletions of the underlying graph G of length at most $O(\alpha n)$. Let*

$$\beta = \mu \frac{(2\mu)}{(\mu - 2\alpha + 1) + 1} \text{ where } \mu > 2\alpha.$$

Then, there exists a streaming algorithm that takes $O(\frac{\beta^{4/3}(n\alpha)^{2/3}}{\varepsilon^{4/3}})$ space in expectation and outputs a $(1 + \varepsilon)\beta$ approximation of M^* with probability at least 0.86.

Quite recently Assadi *et al.* [3] gave an $\Omega(\frac{n^{1/2}}{\alpha^{2.5}})$ space lower bound for getting a c-approximation of the matching size in dynamic graph streams with arboricity bounded by α. We obtain our $n^{2/3}$ bound by first defining an algorithm for insert-only streams with as $n^{1/2}$ behavior, which suggests that this could also be feasible in the dynamic setting.

Our algorithms for bounded arboricity graphs are based on two novel *streaming-friendly* characterizations of the maximum maching size. The first characterization is a modification

[1] For any H-minor-free graph, the arboricity is $O(h\sqrt{h})$ where h is the number of vertices of H. [24]

Table 1 Known results for estimating the size of a maximum matching in data streams.

Reference	Graph class	Stream	Approx. Factor[2]	Space Bound[3]
[21]	General	Random Order	$O(\text{polylog}(n))$	$O(\text{polylog}(n))$
[15]	Arboricity $\leq \alpha$	Insert-Only	$5\alpha + 9$	$\tilde{O}(\alpha n^{2/3})$
[30]	Arboricity $\leq \alpha$	Insert-Only	$\alpha + 2$	$\tilde{O}(\alpha n^{2/3})$
[6, 7]	Arboricity $\leq \alpha$	Insert/Delete	$O(\alpha)$	$\tilde{O}(\alpha n^{4/5})$
This paper	Arboricity $\leq \alpha$	Insert-Only	$(2\alpha + 1)(2\alpha + 2)$	$\tilde{O}(\alpha^{2.5}\sqrt{n})$
This paper	Arboricity $\leq \alpha$	Insert/Delete	$(2\alpha + 1)(2\alpha + 2)$	$\tilde{O}(\alpha^{10/3} n^{2/3})$
This paper	Arboricity $\leq \alpha$	Insert-Only	$22.5\alpha + 6$	$\tilde{O}(\alpha \log n)$

of the characterization in [15] which approximates the size of the maximum matching by $h_\mu + s_\mu$ where h_μ is defined as the number of high degree vertices (vertices with degree more than a threshold μ) and s_μ is the number of shallow edges (edges with low degree endpoints). While h_μ can be easily approximated by sampling the vertices and checking if they are high degree or not, approximating s_μ in sublinear space is a challenge because in one pass we cannot determine if a sampled edge is shallow or not. The work of [15] resolves this issue by sampling the edges at a high rate and manages to implement their characterization in $\tilde{O}(\alpha n^{2/3})$ space for adversarial insert-only streams.

To bring the space usage down to $\tilde{O}(\alpha^{2.5} n^{1/2})$ (for insert-only streams), we modify the formulation of the above characterization. We still need to approximate h_μ but instead of s_μ we approximate n_μ the number of non-isolated vertices in the induced subgraph G_μ defined over the low degree vertices. Note that s_μ is the number of edges in G_μ. This subtle change of definition turns out to be immensely helpful. Similar to h_μ we only need to sample the nodes and check if their degrees are below a certain threshold or not. However we carry the additional constraint that we have to avoid counting the nodes in G_μ that are isolated (have only high degree nodes as neighbors). To satisfy this additional constraint, our algorithm stores the neighbors of the sampled vertices along with a counter for each that maintains their degree in the rest of the stream. Although we only obtain a lower bound on the degree of the neighbors, as it turns out the lower bound information on the degree is still useful because we can ensure the number of false positives that contribute to our estimate is within a certain limit. As a result, we can approximate $h_\mu + n_\mu$ using $\tilde{O}(\alpha n^{1/2})$ space which gives a $(2\alpha + 1)(2\alpha + 2)$ approximation of the maximum matching size after choosing appropriate values for μ and other parameters. This characterization is of particular importance, as it can be adapted to work under edge deletions as well as long as the number of deletions is bounded by $O(\alpha n)$. Details of the characterization and the associated algorithms are given in Lemma 4 and Section 3.2.

To obtain a polylog(n) space algorithm (and prove the claim of Theorem 1), we give a totally new characterization. This characterization, unlike the previous ones that only depend on the parameters of the graph, also takes the ordering of the edges in the stream into account. Roughly speaking, we characterize the size of a maximum matching by the number of edges in the stream that have few neighbor edges in the rest of the stream. To understand the connection with maximum matching, consider the following simplistic special

[2] In some entries, a $(1 + \varepsilon)$ multiplicative factor has been suppressed for concision.
[3] The $\tilde{O}(.)$ notation hides logarithmic in n dependencies.

case. Suppose the input graph G is a forest composed of k disjoint stars. Observe that the maximum matching on this graph is just to pick one edge from each star. We relate this to a combinatorial characterization that arises from the sequence of edges in the stream: no matter how we order the edges of G in the stream, from each star there is exactly one edge that has no neighboring edges in the remainder of the stream (in other words, the last edge of the star in the stream). Our characterization generalizes this idea to graphs with arboricity bounded by α by counting the γ-*good edges*, *i.e.* edges that have at most $\gamma = 6\alpha$ neighbors in the remainder of the stream. We prove this characterization gives an $O(\alpha)$ approximation of the maximum matching size. More important, a nice feature of this characterization is that it can be implemented in polylog(n) space if one allows a $1+\varepsilon$ approximation. The implementation adapts an idea from the well-known L_0 sampling algorithm. It runs $O(\log n)$ parallel threads each sampling the stream at a different rate. At the end, a thread "wins" that has sampled roughly $\Theta(\log n)$ elements from the γ-good edges (samples the edges with a rate of $\frac{\log n}{k}$ where k is the number of γ-good edges). The threads that under-sample will end up with few edges or nothing while the ones that have oversampled will keep too many γ-good edges and will be terminated as soon as they hit a space threshold as a result. Table 1 summarizes the known and new results for estimating the size of a maximum matching.

1.2 Related Work

We discuss the most relevant work to matching size estimation earlier in the introduction. Here, we give an overview of related work by providing the context of matching and streaming algorithms in general, before focusing in on the most related works at the intersection. The problem of computing the maximum matching of G has been extensively studied in the classical offline model, where we assume we have enough space to store all vertices and edges of a graph $G = (V, E)$. The classical result in this model is the algorithm due to Micali and Vazirani [35] with running time $O(m\sqrt{n})$, where $n = |V|$ and $m = |E|$. Recent work has given improved results for sparse bipartite graphs [27]. A matching of size within $(1-\varepsilon)$ factor of a maximum cardinality matching can be found in $O(m/\varepsilon)$ time [19, 35]. Recently, Duan and Pettie [12] developed a $(1-\varepsilon)$-approximate maximum weighted matching algorithm in time $O(m/\varepsilon)$.

The model of streaming data analysis has received a similar level of scrutiny. A survey by McGregor [31] gives an overview of results in the graph streaming model. Many fundamental questions have been tackled: counting the number of occurrences of specific small subgraphs such as triangles [33]; estimating properties of neighborhoods [8]; and using 'sketch' techniques to track local and global properties of graphs like connectivity [2].

The question of *finding* an approximation to the maximum cardinality matching has been extensively studied in the streaming model. An $O(n)$-space greedy algorithm trivially obtains a maximal matching, which is a 2-approximation for the maximum cardinality matching [16]. A natural question is whether one can beat the approximation factor of the greedy algorithm with $O(n\,\text{polylog}(n))$ space. Recently, it was shown that obtaining an approximation factor better than $\frac{e}{e-1} \simeq 1.58$ in one pass requires $n^{1+\Omega(1/\log\log n)}$ space [17, 20], even in bipartite graphs and in the *vertex-arrival* model, where the vertices arrive in the stream together with their incident edges. This setting has also been studied in the context of *online algorithms*, where each arriving vertex has to be either matched or discarded irrevocably upon arrival. Seminal work due to Karp, Vazirani and Vazirani [22] gives an online algorithm with $\frac{e}{e-1}$ approximation factor in this model.

Closing the gap between the upper bound of 2 and the lower bound of $\frac{e}{e-1}$ remains one of the most appealing open problems in the graph streaming area (see [39]). The

factor of 2 can be improved on if one either considers the random-order model or allows for two passes [23]. By allowing even more passes, the approximation factor can be improved to multiplicative $(1-\epsilon)$-approximation via finding and applying augmenting paths with successive passes [28, 29, 13, 1].

Another line of research [16, 28, 43, 14, 11] has explored the question of approximating the maximum-weight matching in one pass and $O(n\,\mathrm{polylog}(n))$ space. The latest result is that a $(2+\varepsilon)$ approximation factor is possible using an $O(n \log n)$ space deterministic algorithm, essentially meeting the unweighted matching case [40]. These results are for the insert-only case. Where deletions are allowed (the dynamic, or turnstile case), the problem is harder: $\Omega(n^{2-3\epsilon})$ space is needed to provide an $O(n^{\epsilon})$ approximation [5]; and $\Omega(n/\alpha^2)$ to provide an $O(\alpha)$ approximation [4]. However, our focus is on finding the size of the maximum matching without materializing it, and so our aim is for sublinear space algorithms.

2 Preliminaries and Notations

Let $G(V,E)$ be an undirected unweighted graph with $n=|V|$ vertices and $m=|E|$ edges. For a vertex $v \in V$, let $\deg_G(v)$ denote the degree of vertex v in G. A *matching* M of G is a set of pairwise non-adjacent edges, i.e., no two edges share a common vertex. Edges in M are called *matched* edges; the other edges are called *unmatched*. A *maximum matching* of graph $G(V,E)$ is a matching of maximum size. Throughout the paper, when we fix a maximum matching of $G(V,E)$, we denote it by M^*. A matching M of G is *maximal* if it is not a proper subset of any other matching in graph G. Abusing the notation, we sometimes use M^* and M for the size of the maximum and maximal matching, respectively. It is well-known (see for example [26]) that the size of a maximal matching is at least half of the size of a maximum matching, i.e., $M \geq M^*/2$. Thus, we say a maximal matching is a 2-approximation of a maximum matching of G. It is known [26] that the simple greedy algorithm, where we include each new edge if neither of its endpoints are already matched, returns a maximal matching.

3 Algorithms for Bounded Arboricity Graphs

Throughout this section, h_μ denotes the number of vertices in graph $G=(V,E)$ that have degree above μ. Let $G_\mu=(L,F)$ be the induced subgraph of G where $L=\{v|\deg_G(v)\leq\mu\}$ and $(u,v)\in F\subseteq E$ when u and v are both in L. Note that G_μ might have isolated vertices. In the following we let M_μ denote the size of maximum matching in G_μ.

3.1 Characterization lemmas

▶ **Lemma 3** ([15]). *For a α-bounded arboricity graph $G(V,E)$ and $\mu > 2\alpha$, we have $h_\mu \leq \frac{2\mu}{\mu-2\alpha+1}M^*$.*

▶ **Lemma 4.** *For a α-bounded arboricity graph $G(V,E)$ and $\mu > 2\alpha$, we have*

$$M^* \leq h_\mu + M_\mu \leq \left(\frac{2\mu}{\mu-2\alpha+1}+1\right)M^* \ .$$

Proof. The lower bound is easy to see: every edge of a maximum matching either has an endpoint with degree more than μ or both of its endpoints are vertices with degree at most μ. The number of matched edges of the first type are bounded by h_μ whereas the number of matched edges of the second type are bounded by M_μ.

To prove the upper bound, we use the fact $M_\mu \leq M^*$ and Lemma 3. ◀

▶ **Definition 5.** Let $S = (e_1, \ldots, e_m)$ be a sequence of edges. We say the edge $e_i = (u, v)$ is γ-*good* with respect to S if $\max\{d_i(u), d_i(v)\} \leq \gamma$ where $d_i(x)$ is defined as $|\{e_j | j > i, e_j = (x, w)\}|$, i.e. the number of edges incident on x that appear after the i-th edge in the stream. We write $E_\gamma(S)$ as the set of γ-good edges in S, and usually drop (S) in context.

To illustrate the power of this definition, we first consider the case of trees. Trees are a good test case for understanding matchings, since they can have widely varying matching sizes: from 1 (a star graph on n nodes) to $O(n)$ (a path of length n or a binary tree on n nodes). In fact the following lemma suggests that counting the number of 1-good edges gives a 2 factor approximation to the matching size on trees. (Due to the space limitations we have deferred the proof of this lemma to the full version of this paper.)

▶ **Lemma 6.** *For trees we have* $M^* \leq |E_1| \leq 2M^*$.

Our main result on γ-goodness is for general graphs with α-bounded arboricity.

▶ **Lemma 7.** *Let* $\mu > 2\alpha$ *be a (large enough) integer, and let* E_γ *be the set of* γ*-good edges in an edge stream for a graph with arboricity at most* α*. We have:*

$$\left(\frac{1}{2} - \frac{\alpha}{\mu + 1}\right) M^* \;\leq\; |E_\gamma| \;\leq\; \left(\frac{5}{4}\gamma + 2\right) M^*,$$

where $\gamma = \max\{\mu - 1, \frac{4\alpha(\mu+1)}{\mu+1-2\alpha}\}$*. In particular for* $\mu = 6\alpha - 1$*, we have*

$$M^* \;\leq\; 3|E_{6\alpha}| \;\leq\; (22.5\alpha + 6)M^*$$

Proof. First we prove the lower bound on $|E_\gamma|$. In particular we show a relation involving the number of edges where both endpoints have low degree. Define $h_\mu = |\{v | v \in V, \deg_G(v) > \mu\}|$, and $s_\mu = |\{e = (u, v) | e \in E, \deg_G(u) \leq \mu, \deg_G(v) \leq \mu\}|$, i.e. the number of edges in the graph G_μ. Then:

$$\left(\frac{1}{2} - \frac{\alpha}{\mu + 1}\right) h_\mu + s_\mu \;\leq\; |E_\gamma|.$$

The claim in the lemma follows from the relatively loose bound that $M^* \leq h_\mu + s_\mu$. Let H be the set of vertices in the graph with degree above μ and let $L = V \setminus H$. Recall that $h_\mu = |H|$. Let H_0 be the vertices in H that have no neighbor in L, and let $H_1 = H \setminus H_0$. First we notice that $|H_1| \geq (1 - \frac{2\alpha}{\mu})|H|$. To see this, let E' be the edges with at least one endpoint in H_0. By definition, every node in H_0 has degree at least $\mu + 1$, so we have $|E'| \geq \frac{\mu+1}{2}|H_0|$. At the same time, the total number of edges in the subgraph induced by the nodes H is at most $\alpha(|H| - 1)$, using the arboricity assumption. Therefore,

$$\alpha(|H| - 1) \geq |E'| \geq \tfrac{1}{2}(\mu + 1)|H_0|$$

It follows that $|H_0| \leq \frac{2\alpha}{\mu+1}(|H| - 1)$ which further implies that

$$|H_1| \geq (1 - \frac{2\alpha}{\mu + 1})|H| = (1 - \frac{2\alpha}{\mu + 1})h_\mu. \tag{1}$$

Now let $\deg_H(v)$ be the degree of v in the subgraph induced by H. We have $\sum_{v \in H_1} \deg_H(v) \leq 2\alpha|H|$, again using the arboricity bound and the fact that summing over degrees counts each edge at most twice. Therefore, taking the average over nodes in H_1,

$$\overline{\deg_H}(v) \leq \frac{2\alpha}{1 - \frac{2\alpha}{\mu+1}}$$

for $v \in H_1$. Consequently, at least half of the vertices in H_1 have their $\deg_H$ bounded by $\frac{4\alpha(\mu+1)}{\mu+1-2\alpha}$ (via the Markov inequality). Let H_1' be those vertices. For each $v \in H_1'$ we find a γ-good edge. Let $e^* = (v, u)$ be the last edge in the stream where $u \in L$. Then, there cannot be too many edges that neighbor (v, u) and come after it in the stream: the total number of edges that share an endpoint with e^* in the rest of the stream is bounded by $\max\{\mu - 1, \frac{4\alpha(\mu+1)}{\mu+1-2\alpha}\}$. Consequently, for

$$\gamma = \max\{\mu - 1, \frac{4\alpha(\mu+1)}{\mu+1-2\alpha}\},$$

we have $|E_\gamma| \geq (\frac{1}{2} - \frac{\alpha}{\mu+1})h_\mu$, based on the set of $|H_1|/2$ edges connected to the vertices in H_1' and using (1). For $\gamma \geq \mu$, E_γ also contains the disjoint set of edges from $L \times L$, which are all guaranteed to be γ-good since both their endpoints have degree bounded by μ. Therefore, as claimed,

$$|E_\gamma| \geq s_\mu + \left(\frac{1}{2} - \frac{\alpha}{\mu+1}\right) h_\mu.$$

To prove the upper bound on $|E_\gamma|$, we notice that the subgraph containing only the edges in E_γ has degree at most $\gamma + 1$. Such a graph has a matching size of at least $\frac{4|E_\gamma|}{5(\gamma+1)+3}$ [18]. It follows that $|E_\gamma| \leq \frac{5\gamma+8}{4} M^*$. This finishes the proof of the lemma. ◀

3.2 $\tilde{O}(\sqrt{n})$ space algorithm for insert-only streams

In this section, we present Algorithm 1 to estimate $M_\mu + h_\mu$ and prove the following theorem.

▶ **Theorem 8.** *Let $G(V, E)$ be a graph with the arboricity bounded by α. Let S be an (adversarial order) insertion-only stream of the edges of the underlying graph G. Let*

$$\beta = \mu((2\mu)/(\mu - 2\alpha + 1) + 1) \text{ where } \mu > 2\alpha.$$

Then, there exists a streaming algorithm (Algorithm 1) that processes S, takes $O(\frac{\beta\sqrt{\alpha n}}{\varepsilon} \log n)$ space in expectation and outputs a $(1+\varepsilon)\beta$ approximation of M^* with probability at least 0.86, where M^* is a maximum matching of G.

For each w in $\Gamma(T)$ (the set of neighbors of nodes in T), the algorithm maintains $l(w)$, the number of occurrences of w observed since the first time a neighbor of w was added to T. Note that in this algorithm, $l(w)$ is a lower bound on the degree of w. For the output, T_1 is the subset of nodes in T whose degree is bounded by μ and additionally for each node in T_1, there is a neighbor w whose observed degree ($d(w)$ or $l(w)$) is at most μ. Meanwhile, T_2 is the set of "high degree" nodes in T.

▶ **Lemma 9.** *Let $\varepsilon \in (0, 1)$ and $\beta = \mu(\frac{2\mu}{\mu-2\alpha+1} + 1)$. With probability at least $1 - e^{\frac{-\varepsilon^2 M^* p}{4\beta^2}}$, Algorithm 1 outputs s where*

$$(1-\varepsilon)M^* \leq s \leq (1+\varepsilon)\beta M^*.$$

Proof. First we prove the following bounds on $\mathsf{E}(s)$.

$$M_\mu + h_\mu \leq \mathsf{E}(s) \leq \mu(M_\mu + h_\mu).$$

Let L be the set of vertices in G that have degree at most μ and let G_L be the induced graph on L. Let $H = V \setminus L$. Note that G_L might have isolated vertices. Let N be the non-isolated

Algorithm 1: Estimate-$M_\mu + h_\mu$

Initialization: Each node is sampled to set T with probability p (determined below).

Stream Processing:
forall *edges* $e = (u, v)$ *in the stream* **do**
  if $u \in T$ *or* $v \in T$ **then**
    store e in H;
    if $u \in T$ **then** increment $d(u)$ **else** increment $l(u)$;
    if $v \in T$ **then** increment $d(v)$ **else** increment $l(v)$;

Post Processing:
Let $T_1 = \{v \in T | d(v) \leq \mu, \exists w \in \Gamma(v) : d(w) + l(w) \leq \mu\}$
Let $T_2 = \{v \in T | d(v) > \mu\}$
return $s = (|T_1| + |T_2|)/p$

Algorithm 2: Estimate-M^*

Initialization: Let $\varepsilon \in (0, 1)$ and $t = \lceil \frac{\beta\sqrt{8nc}}{\varepsilon} \rceil$ where β is as defined in Lemma 9.
Stream Processing: Do the following tasks in parallel:
(1) Greedily keep a maximal matching of size at most $r \leq t$ (and terminate this task if this size bound is exceeded).
(2) Run the Estimate-$(M_\mu + h_\mu)$ procedure (Algorithm 1) with $p \geq \frac{8}{\lambda^2 t}$ where $\lambda = \frac{\varepsilon}{\beta}$.

Post processing: If $r < t$ then output $2r$ as the estimate for M^*, otherwise output the result of the Estimate-$(M_\mu + h_\mu)$ procedure.

vertices in G_L. It is clear that if the algorithm samples $v \in N$, v will be in T_1. Likewise, if it samples a vertex $w \in H$, w will be in T_2. Given the fact that $|H| = h_\mu$ and $|N| \geq M_\mu$, this proves the lower bound on $\mathsf{E}(s)$.

The expectation may be above M_μ, as the algorithm may pick an isolated vertex in G_L (a vertex that is *only* connected to the high-degree vertices) and include it in T_1 because one of its high-degree neighbours w was identified as low degree, i.e., $w \in \Gamma(T)$ and $l(w) \leq \mu$ but $w \in H$. Let $u \in H$ and let $U = \{a_1, \ldots, a_\mu\}$ be the last μ neighbours of u according to the ordering of the edges in the stream. The algorithm can only identify u as low degree when it picks a sample from U and no samples from $\Gamma(u) \setminus U$. This restricts the number of *unwanted* isolated vertices to at most μh_μ. Together with the fact that $|N| \leq \mu M_\mu$, it establishes the upper bound on $\mathsf{E}(s)$. Now using a Chernoff bound,

$$\begin{aligned}\Pr\left[|s - \mathsf{E}(s)| \geq \lambda \mathsf{E}(s)\right] &= \Pr\left[|s.p - \mathsf{E}(s.p)| \geq \lambda \mathsf{E}(s.p)\right] \\ &\leq \exp(-\lambda^2 (M_\mu + h_\mu)p/4) \leq \exp(-\lambda^2 M^* p/4).\end{aligned}$$

Therefore with probability at least $1 - e^{\frac{-\lambda^2 M^* p}{4}}$,

$$(M_\mu + h_\mu) - \lambda\mu(h_\mu + M_\mu) \leq s \leq \mu(1 + \lambda)(M_\mu + h_\mu) \tag{2}$$

Setting $\lambda = \frac{\varepsilon}{\beta}$ and combining with Lemma 4, we derive the statement of the lemma. ◀

Proof of Theorem 8. Suppose $M^* < t$. Clearly the size of the maximal matching r obtained by the first task will be less than t. In this case, $M^* \leq 2r \leq 2M^*$. Now suppose $M^* \geq t$.

By Lemma 4, we will have $M_\mu + h_\mu \geq t$ and hence by Lemma 9, with probability at least $1-e^{-2} \geq 0.86$, the output of the algorithm will be within the promised bounds. The expected space of the algorithm is $O((t+pn\alpha)\log n)$. Setting $t = \beta\sqrt{8n\alpha}/\varepsilon$ to balance the space costs, the space complexity of the algorithm will be $O(\frac{\beta\sqrt{\alpha n}}{\varepsilon}\log n)$ as claimed. ◀

3.3 $O(n^{2/3})$ space algorithm for insertion/deletion streams

Algorithms 1 and 2 form the basis of our solution in the more general case where the stream contains deletions of edges as well. In the case of Algorithm 1, the algorithm has to maintain the induced subgraph on T and the edges of the cut $(T, \Gamma(T))$. However if we allow an arbitrary number of insertions and deletions, the size of the cut $(T, \Gamma(T))$ can grow as large as $O(n)$ even when $|T| = 1$. This is because each node at some intermediate point could become high degree and then lose its neighbours because of the subsequent deletion of edges. Therefore here in order to limit the space usage of the algorithm, we make the assumptions that number of deletions is bounded by $O(\alpha n)$. Since the processed graph has arboricity at most α this forces the number of insertions to be $O(\alpha n)$ as well. Under this assumption, if we pick a random vertex, still, in expectation the number of neighbours is bounded by $O(\alpha)$.

Another complication arises from the fact that, with edge deletions, a vertex added to $\Gamma(T)$ might become isolated at some point. In this case, we discard it from $\Gamma(T)$. Additionally for each vertex in $T \cup \Gamma(T)$, the counters $d(v)$ (or $l(v)$ depending on if it belongs to T or $\Gamma(T)$) can be maintained as before. The space complexity of the algorithm remains $O(pn\alpha \log n)$ in expectation as long as the arboricity factor remains within $O(\alpha)$ in the intermediate graphs. In the case of Algorithm 2, we need to keep a maximal matching of size $O(t)$. This can be done in $O(t^2)$ space using a randomized algorithm [7]. Setting t at $(\frac{8\beta n\alpha}{\varepsilon^2})^{1/3}$ to rebalance the space costs, we obtain the result of Theorem 2.

3.4 The polylog space algorithm for insert-only streams

In this section we present our polylog space algorithm by presenting an algorithm for estimating $|E_\gamma|$ within a $(1+\varepsilon)$ factor. Our algorithm is similar in spirit to the well-known L_0 sampling strategy [9]. We first describe it in terms of running $O(\log n)$ parallel threads each sampling the stream at a different rate. At the end, a thread "wins" that has sampled roughly $\Theta(\log n)$ elements from $|E_\gamma|$ (samples the edges with a rate of $\frac{\log n}{|E_\gamma|}$). The threads that under-sample will end up with few edges or nothing while the ones that have oversampled will keep too many elements of E_γ and will be aborted as result. Finally, we mention how a suitable implementation can reduce the space dependency to $O(\alpha \log n)$ (treating ϵ as constant).

First we give a simple subroutine (Algorithm 3) that is the building block of the algorithm. Given γ and an edge e in the stream, it simply counts up the number of subsequent edges that are incident on either endpoint of e, and consequently determines whether e is in E_γ. Our main algorithm (Algorithm 4) samples edges, and applies this subroutine to them. Multiple sampling rates p_i are used in parallel; however, if at any point the number of sampled edges in a level exceeds a threshold τ, the level is "terminated", and no further samples are taken at this level. This ensures that the space used remains bounded.

▶ **Lemma 10.** *With high probability, Algorithm 4 outputs a $1 \pm O(\varepsilon)$ approximation of $|E_\gamma|$ where γ is defined according to Lemma 7.*

Proof. Consider the sets of active γ-good tests at each level at the conclusion of the algorithm, X_i. First we observe that if $|X_0| \leq \tau$ then $X_0 = E_\gamma$ and the algorithm makes no error.

Algorithm 3: The γ-good test

Initialization: given the edge $e = (u, v)$ in the stream, let $r(u) = 0$ and $r(v) = 0$.
forall *subsequent edges* $e' = (t, w)$ **do**
  if $u \in \{t, w\}$ **then** increment $r(u)$;
  if $v \in \{t, w\}$ **then** increment $r(v)$;
  if $\max\{r(u), r(v)\} > \gamma$ **then** terminate and report NOT γ-good;

Algorithm 4: An algorithm for approximating $|E_\gamma|$

Initialization: $\forall i. X_i = \emptyset$ ▷ X_i represents the current set of sampled γ-good edges.

Stream Processing:
forall *levels* $i \in \{0, 1, \ldots, \lfloor \lfloor \log_{1+\varepsilon} n^2 \rfloor \}$ *in parallel* **do**
  forall *edges* e **do**
    Feed e to the active γ-good tests and update X_i
    With probability $p_i = \frac{1}{(1+\varepsilon)^i}$ add e to X_i and start a γ-good test for e.
    Let $|X_i|$ be the number of active γ-good tests within this level.
    if $|X_i| > \tau = \frac{64\gamma^2 \log n}{\alpha\varepsilon^2}$ **then** terminate level i;

Post processing:
if $|X_0| \le \tau$ **then**
  return $|X_0|$
else ▷ $|X_0| > \tau$
  let j be smallest integer s.t. $|X_j| \le \frac{8 \log n(1+\varepsilon)}{\varepsilon^2}$ and j-th level was not terminated;
  if *there is no such* j **then return** FAIL **else return** $\frac{|X_j|}{p_j}$;

In case $|X_i| > \tau$, we claim that $|E_\gamma| > \frac{\alpha}{2\gamma^2}\tau$. To prove this, let t be the time step where $|X_i|$ exceeds τ (i.e. when this level is terminated) and let $G_t = (V, E^{(t)})$ be the graph where $E^{(t)} = \{e_1, \ldots, e_t\}$. Clearly $M^*(G) \ge M^*(G_t)$ because the size of the matching only increases as new edges arrive. Abusing the notation, let $E_\gamma(G_t)$ denote the set of γ-good edges at time t. By Lemma 7 and definition of γ, we have

$$\tau < |E_\gamma(G_t)| \le \left(\frac{5}{4}\gamma + 2\right) M^*(G_t) \le 4\gamma M^*(G) \le \frac{2(\mu+1)}{\mu - 2\alpha + 1} 4\gamma |E_\gamma| \le \left(\frac{\gamma}{2\alpha}\right) 4\gamma |E_\gamma| \tag{3}$$

This proves the claim that $|E_\gamma| > \frac{\alpha}{2\gamma^2}\tau$ when $|X_i| > \tau$. Let $\tau' = \frac{8 \log n}{\varepsilon^2}$ and let i^* be the integer such that

$$(1+\varepsilon)^{i^*-1}\tau' \le |E_\gamma| \le (1+\varepsilon)^{i^*}\tau'.$$

Assuming the i^*-th level does not terminate before the end, we have $\frac{\tau'}{(1+\varepsilon)} \le \mathsf{E}[|X_{i^*}|] \le \tau'$. By a Chernoff bound, for each i we have (again assuming we do not terminate the corresponding level)

$$\Pr\left[\left||X_i| - \mathsf{E}(|X_i|)\right| \ge \varepsilon \mathsf{E}(|X_i|)\right] \le \exp\left(-\frac{\varepsilon^2 p_i |E_\gamma|}{4}\right).$$

$$\text{So, } \Pr\left[\left||X_{i^*}| - \mathsf{E}(|X_{i^*}|)\right| \ge \varepsilon \mathsf{E}(|X_{i^*}|)\right] \le \exp\left(-\frac{\varepsilon^2 |E_\gamma|}{2(1+\varepsilon)^{i^*}}\right) \le \exp\left(\frac{2\log n}{1+\varepsilon}\right) \le O(n^{-1}).$$

As a result, with high probability $|X_{i^*}| \le \frac{8 \log n(1+\varepsilon)}{\varepsilon^2}$. Moreover for all $i < i^* - 1$, the corresponding levels either terminate prematurely or in the end we will have $|X_i| >$

$\frac{8\log n(1+\varepsilon)}{\varepsilon^2}$ with high probability. Consequently $j \in \{i^*, i^*-1\}$. It remains to prove that runs corresponding to i^* and i^*-1 will survive until the end with high probability. We prove this for i^*. The case of i^*-1 is similar.

Consider a fixed time t in the stream and let $X_{i^*}^{(t)}$ be the set of sampled γ-good edges at time t corresponding to the i^*-th level. Note that $X_{i^*}^{(t)}$ contains the a subset of γ-good edges with respect to the stream $S_t = (e_1, \ldots, e_t)$. From the definition of i^* and Inequality (3) we have

$$\mathsf{E}[|X_{i^*}^{(t)}|] = \frac{|E_\gamma(G_t)|}{(1+\varepsilon)^{i^*}} \leq \frac{2\gamma^2|E_\gamma|}{\alpha(1+\varepsilon)^{i^*}} \leq \frac{2\gamma^2\tau'}{\alpha}.$$

By the Chernoff inequality for $\delta \geq 1$,

$$\Pr\left[|X_{i^*}^{(t)}| \geq (1+\delta)E(|X_{i^*}^{(t)}|)\right] \leq \exp\left(\frac{-\delta}{3}E(|X_{i^*}^{(t)}|)\right).$$

From $\delta = \frac{\tau}{E(|X_{i^*}^{(t)}|)} - 1 = \frac{\tau(1+\varepsilon)^{i^*}}{|E_\gamma(G_t)|} - 1$, we get

$$\Pr\left[|X_{i^*}^{(t)}| \geq \tau\right] \leq \exp\left(\frac{-\tau}{3} + \frac{|E_\gamma(G_t)|}{(1+\varepsilon)^{i^*}}\right) \leq \exp\left(\frac{-\tau}{3} + \frac{2\gamma^2\tau'}{\alpha}\right)$$

For $\tau \geq \frac{8\gamma^2\tau'}{\alpha}$, the term inside the exponent is smaller than $-2\log n$. It also satisfies $\delta \geq 1$. After applying the union bound, for all t the size of $X_{i^*}^{(t)}$ is bounded by $\tau = \frac{64\gamma^2\log n}{\alpha\varepsilon^2}$ with high probability. This finishes the proof of the lemma. ◀

Next, putting everything together, we prove Theorem 1.

Proof of Theorem 1. The theorem follows from Lemmas 7 and 10 and taking $\gamma = \mu+1 = 6\alpha$. Observe that the space cost of Algorithm 4 can be bounded: we have $\log_{1+\varepsilon} n^2$ levels where each level runs at most τ concurrent γ-good tests otherwise it will be terminated. Each γ-good test keeps an edge and two counters and as result it occupies $O(1)$ space. Consequently the space usage of the algorithm is bounded by $O(\tau \log_{1+\varepsilon} n)$. Using the fact that $\tau = O(\frac{\alpha}{\varepsilon^2}\log n)$ for $\gamma = 6\alpha$, we obtain a space bound of $O(\frac{\alpha}{\varepsilon^2}\log^2 n)$.

A simple implementation optimization is not to run multiple guesses of p in parallel, but instead to begin with $i = 1$ and $p_1 = 1$. Whenever $|X_i| > \tau$, then we increment i and uniformly sample elements from X_i into X_{i+1} with probability $\frac{1}{1+\epsilon}$. It is immediate that the resulting X_{i+1} corresponds to a sample of the γ-good edges in the stream so far with a sampling probability of $p_i = \frac{1}{(1+\epsilon)^i}$, by the principle of deferred decisions. Consequently, the space bound is reduced to $O(\frac{\alpha}{\epsilon^2}\log n)$. ◀

Acknowledgements. We thank Andrew McGregor and Jelani Nelson for some helpful conversations. We also thank the anonymous reviewers for their careful reading of the paper and helpful comments.

References

1 K. J. Ahn, S. Guha, and A. McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 459–467, 2012.

2 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *ACM Principles of Database Systems*, pages 5–14, 2012. doi:10.1145/2213556.2213560.

3 Sepehr Assadi, Sanjeev Khanna, and Yang Li. On estimating maximum matching size in graph streams. In *Proceedings of the Twenty-Eigth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, January 2017*, 2017. URL: `http://www.seas.upenn.edu/~sassadi/pages/streaming_matching-size_2017.html`.

4 Sepehr Assadi, Sanjeev Khanna, and Yang Li. On estimating maximum matching size in graph streams. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1723–1742, 2017. `doi:10.1137/1.9781611974782.113`.

5 Sepehr Assadi, Sanjeev Khanna, Yang Li, and Grigory Yaroslavtsev. Maximum matchings in dynamic graph streams and the simultaneous communication model. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1345–1364, 2016. `doi:10.1137/1.9781611974331.ch93`.

6 M. Bury and C. Schwiegelshohn. Sublinear estimation of weighted matchings in dynamic data streams. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA)*, pages 263–274, 2015.

7 R. Chitnis, G. Cormode, H. Esfandiari, M.T. Hajiaghayi, A. McGregor, M. Monemizadeh, and S. Vorotnikova. Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1326–1344, 2016.

8 Edith Cohen. All-distances sketches, revisited: HIP estimators for massive graphs analysis. In *ACM Principles of Database Systems*, pages 88–99, 2014. `doi:10.1145/2594538.2594546`.

9 Graham Cormode and Donatella Firmani. On unifying the space of ℓ_0-sampling algorithms. In *Algorithm Engineering and Experiments*, 2013.

10 Graham Cormode, Hossein Jowhari, Morteza Monemizadeh, and S. Muthukrishnan. The sparse awakens: Streaming algorithms for matching size estimation in sparse graphs. Technical Report 1608.03118, ArXiv, 2016. URL: `http://arxiv.org/abs/1608.03118`.

11 M. Crouch and D. S. Stubbs. Improved streaming algorithms for weighted matching, via unweighted matching. In *Proceedings of the 17th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, pages 96–104, 2014.

12 R. Duan and S. Pettie. Linear-time approximation for maximum weight matchings. *Journal of the ACM*, 61(1):1–23, 2014.

13 S. Eggert, L. Kliemann, P. Munstermann, and A. Srivastav. Bipartite graph matchings in the semi-streaming model. *Algorithmica*, 63(1-2):490–508, 2012.

14 Leah Epstein, Asaf Levin, Julián Mestre, and Danny Segev. Improved approximation guarantees for weighted matching in the semi-streaming model. *SIAM J. Discrete Math.*, 25(3):1251–1265, 2011. `doi:10.1137/100801901`.

15 H. Esfandiari, M.T. Hajiaghyi, V. Liaghat, M. Monemizadeh, and K. Onak. Streaming algorithms for estimating the matching size in planar graphs and beyond. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2015.

16 J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2):207–216, 2005.

17 Ashish Goel, Michael Kapralov, and Sanjeev Khanna. On the communication and streaming complexity of maximum bipartite matching. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 468–485, 2012.

18 Yijie Han. Matching for graphs of bounded degree. In *Frontiers in Algorithmics, Second Annual International Workshop, FAW 2008, Changsha, China, June 19-21, 2008, Proceedings*, pages 171–173, 2008. `doi:10.1007/978-3-540-69311-6_19`.

19 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973. `doi:10.1137/0202019`.

20 M. Kapralov. Better bounds for matchings in the streaming model. *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1679–1697, 2013.

21 M. Kapralov, S. Khanna, and M. Sudan. Approximating matching size from random streams. *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 734–751, 2014.

22 R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite matching. *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 352–358, 1990.

23 C. Konrad, F. Magniez, and C. Mathieu. Maximum matching in semi-streaming with few passes. In *Proceedings of the 11th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, pages 231–242, 2012.

24 Alexandr V. Kostochka. Lower bound of the hadwiger number of graphs by their average degree. *Combinatorica*, 4(4):307–316, 1984.

25 Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: A method for solving graph problems in mapreduce. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'11, pages 85–94. ACM, 2011. doi:10.1145/1989493.1989505.

26 L. Lovasz and M.D. Plummer. Matching theory. In *North-Holland, Amsterdam-New York*, 1986.

27 Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 253–262, 2013. doi:10.1109/FOCS.2013.35.

28 A. McGregor. Finding graph matchings in data streams. In *Proceedings of the 8th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, pages 170–181, 2005.

29 A. McGregor. Graph mining on streams. In *Encyclopedia of Database Systems*, pages 1271–1275. Springer, 2009.

30 A. McGregor and S. Vorotnikova. Planar matching in streams revisited. In *Proceedings of the 19th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, 2016.

31 Andrew McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014. doi:10.1145/2627692.2627694.

32 Andrew McGregor and Sofya Vorotnikova. A note on logarithmic space stream algorithms for matchings in low arboricity graphs. Technical Report 1612.02531, ArXiv, 2016.

33 Andrew McGregor, Sofya Vorotnikova, and Hoa T. Vu. Better algorithms for counting triangles in data streams. In *ACM Principles of Database Systems*, pages 401–411, 2016. doi:10.1145/2902251.2902283.

34 Aranyak Mehta, Amin Saberi, Umesh V. Vazirani, and Vijay V. Vazirani. Adwords and generalized online matching. *J. ACM*, 54(5), 2007.

35 S. Micali and V. V. Vazirani. An $o(\sqrt{|V||e|})$ algorithm for finding maximum matching in general graphs. *Proceedings of the 21st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 17–27, 1980.

36 C. St. J. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 36(1):445–450, 1961.

37 C. St. J. A. Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, 39(1):12, 1964.

38 Huy Nguyen and Krzysztof Onak. Constant-time approximation algorithms via local improvements. In *IEEE Conference on Foundations of Computer Science*, 2008.

39 List of open problems in sublinear algorithms: Problem 60. http://sublinear.info/60.

40 Ami Paz and Gregory Schwartzman. A $(2 + \epsilon)$-approximation for maximum weight matching in the semi-streaming model. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2153–2161, 2017. doi:10.1137/1.9781611974782.140.

41 A. E. Roth and M. A. O. Sotomayor. *Two-sided matching: A study in game-theoretic modeling and analysis*. Cambridge University Press, 1990.

42 Nenad Trinajstic, Douglas J. Klein, and Milan Randic. On some solved and unsolved problems of chemical graph theory. *International Journal of Quantum Chemistry*, 30(S20):699–742, 1986.

43 Mariano Zelke. Weighted matching in the semi-streaming model. *Algorithmica*, 62(1-2):1–20, 2012.

Improving TSP Tours Using Dynamic Programming over Tree Decompositions*†

Marek Cygan[1], Łukasz Kowalik[2], and Arkadiusz Socała[3]

1 **University of Warsaw, Poland**
cygan@mimuw.edu.pl
2 **University of Warsaw, Poland**
a.socala@mimuw.edu.pl
3 **University of Warsaw, Poland**
kowalik@mimuw.edu.pl

Abstract

Given a traveling salesman problem (TSP) tour H in graph G a k-move is an operation which removes k edges from H, and adds k edges of G so that a new tour H' is formed. The popular k-OPT heuristic for TSP finds a local optimum by starting from an arbitrary tour H and then improving it by a sequence of k-moves.

Until 2016, the only known algorithm to find an improving k-move for a given tour was the naive solution in time $O(n^k)$. At ICALP'16 de Berg, Buchin, Jansen and Woeginger showed an $O(n^{\lfloor 2/3k \rfloor + 1})$-time algorithm.

We show an algorithm which runs in $O(n^{(1/4+\epsilon_k)k})$ time, where $\lim_{k\to\infty} \epsilon_k = 0$. It improves over the state of the art for every $k \geq 5$. For the most practically relevant case $k = 5$ we provide a slightly refined algorithm running in $O(n^{3.4})$ time. We also show that for the $k = 4$ case, improving over the $O(n^3)$-time algorithm of de Berg et al. would be a major breakthrough: an $O(n^{3-\epsilon})$-time algorithm for any $\epsilon > 0$ would imply an $O(n^{3-\delta})$-time algorithm for the All Pairs Shortest Paths problem, for some $\delta > 0$.

1998 ACM Subject Classification G.2.2 Graph Theory, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases TSP, treewidth, local search, XP algorithm, hardness in P

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.30

1 Introduction

In the Traveling Salesman Problem (TSP) one is given a complete graph $G = (V, E)$ and a weight function $w : E \to \mathbb{N}$. The goal is to find a Hamiltonian cycle in G (also called a *tour*) of minimum weight. This is one of the central problems in computer science and operation research. It is well known to be NP-hard and has been researched from different perspectives, most notably using approximation [1, 4, 25], exponential-time algorithms [13, 16] and heuristics [24, 20, 5].

In practice, TSP is often solved by means of local search heuristics where we begin from an arbitrary Hamiltonian cycle in G, and then the cycle is modified by means of some local

* The work of M. Cygan and Ł. Kowalik is a part of the project TOTAL that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 677651). A. Socała is supported by the National Science Centre of Poland, grant number 2013/09/B/ST6/03136.

† A full version of the paper is available at http://arxiv.org/abs/1703.05559.

changes in a series of steps. After each step the weight of the cycle should improve; when the algorithm cannot find any improvement it stops. One of the most successful examples of this approach is the k-opt heuristic, where in each step an improving k-move is performed. Given a Hamiltonian cycle H in a graph $G = (V, E)$ a *k-move* is an operation that removes k edges from H and adds k edges of G so that the resulting set of edges H' is a new Hamiltonian cycle. The k-move is *improving* if the weight of H' is smaller than the weight of H. The k-opt heuristic has been introduced in 1958 by Croes [5] for $k = 2$, and then applied for $k = 3$ by Lin [21] in 1965. Then in 1972 Lin and Kernighan designed a complicated heuristic which uses k-moves for unbounded values of k, though restricting the space of k-moves to search to so-called sequential k-moves. A variant of this heuristic called LKH, implemented by Helsgaun [14], solves optimally instances up to 85 900 cities. Among other modifications, the variant searches for non-sequential 4- and 5-moves. From the theory perspective, the quality of the solutions returned by k-opt, as well as the length of the sequence of k-moves needed to find a local optimum, was studied, among others, by Johnson, Papadimitriou and Yannakakis [15], Krentel [18] and Chandra, Karloff and Tovey [3]. More recently, smoothed analysis of the running time and approximation ratio was investigated by Manthey and Veenstra [19] and Künnemann and Manthey [22].

In this paper we study the k-opt heuristic but we focus on its basic ingredient, namely on finding a single improving k-move. The decision problem k-opt Detection is to decide, given a tour H in an edge weighted complete graph G, if there is an improving k-move. In its optimization version, called k-opt Optimization, the goal is to find a k-move that gives the largest weight improvement, if any. Unfortunately, this is a computationally hard problem. Namely, Marx [23] has shown that k-opt Detection is $W[1]$-hard, which means that it is unlikely to be solvable in $f(k)n^{O(1)}$ time, for any function f. Later Guo, Hartung, Niedermeier and Suchý [12] proved that there is no algorithm running in time $n^{o(k/\log k)}$, unless Exponential Time Hypothesis (ETH) fails. This explains why in practice people use exhaustive search running in $O(n^k)$ time for every fixed k, or faster algorithms which explore only a very restricted subset of all possible k-moves.

Recently, de Berg, Buchin, Jansen and Woeginger [8] have shown that it is possible to improve over the naive exhaustive search. For every fixed $k \geq 3$ their algorithm runs in time $O(n^{\lfloor 2k/3 \rfloor + 1})$ and uses $O(n)$ space. In particular, it gives $O(n^3)$ time for $k = 4$. Thus, the algorithm of de Berg et al. is of high practical interest: the complexity of the $k = 4$ case now matches the complexity of $k = 3$ case, and hence it seems that one can use 4-opt in all the applications where 3-opt was fast enough. De Berg et al. show also that a progress for $k = 3$ is unlikely, namely k-opt Detection has an $O(n^{3-\epsilon})$-time algorithm for some $\epsilon > 0$ iff All Pairs Shortest Paths problem can be solved in $O(n^{3-\delta})$-time algorithm for a $\delta > 0$.

Our Results. In this paper we extend the line of research started in [8]: we show an algorithm running in time $O(n^{(1/4+\epsilon_k)k})$ and using space $O(n^{(1/8+\epsilon_k)k})$ for every fixed k, where $\lim \epsilon_k = 0$. We are able to compute the values of ϵ_k for $k \leq 10$. These values show that our algorithm improves the state of the art for every $k = 5, \ldots, 10$ (see Table 1). A different adjustment of parameters of our algorithm results in time $O(n^{k/2+3/2})$ and additional space of $O(\sqrt{n})$, which improves the state of the art for every $k \geq 8$.

We also show a good reason why we could not improve over the $O(n^3)$-time algorithm of de Berg et al. for 4-opt Optimization: an $O(n^{3-\epsilon})$-time algorithm for some $\epsilon > 0$ would imply that All Pairs Shortest Paths can be solved in time $O(n^{3-\delta})$ for some $\delta > 0$. Note that although the family of 4-moves contains all 3-moves, it is still possible that there is no improving 3-move, but there is an improving 4-move. Thus the previous lower bound

Table 1 New running times for $k = 5, \ldots, 10$.

k	5	6	7	8	9	10
previous algorithm [8]	$O(n^4)$	$O(n^5)$	$O(n^5)$	$O(n^6)$	$O(n^7)$	$O(n^7)$
our algorithm	$O(n^{3.4})$	$O(n^4)$	$O(n^{4.25})$	$O(n^{4\frac{2}{3}})$	$O(n^5)$	$O(n^{5.2})$

of de Berg et al. does not imply our lower bound, though our reduction is essentially an extension of the one by de Berg et al. [8] with a few additional technical tricks.

We also devote special attention to the $k = 5$ case of k-opt Optimization problem, hoping that it can still be of a practical interest. Our generic algorithm works in $O(n^{3.67})$ time in this case. However, we show that it can be further refined, obtaining the $O(n^{3.4})$ running time. We suppose that similar improvements of order $n^{\Omega(1)}$ should be also possible for larger values of k. In Table 1 we present the running times for $k = 5, \ldots, 10$.

Our Approach. Our algorithm applies dynamic programming on a tree decomposition. This is a standard method for dealing with some sparse graphs, like series-parallel graphs or outerplanar graphs. However, in our case we work with complete graphs. The trick is to work on an implicit structure, called dependence graph D. Graph D has k vertices which correspond to the k edges of H that are chosen to be removed. A subset of edges of D corresponds to the pattern of edges to be added (as we will see the number of such patterns is bounded for every fixed k, and one can iterate over all patterns). The dependence graph can be thought of as a sketch of the solution, which needs to be embedded in the input graph G. Graph D is designed so that if it has a separator S, such that $D - S$ falls apart into two parts A and B, then once we find an optimal embedding of $A \cup S$ for some fixed embedding of S, one can forget about the embedding of A. This intuition can be formalized as dynamic programming on a tree decomposition of D, which is basically a tree of separators in D. The idea sketched above leads to an algorithm running in time $O(n^{(1/3+\epsilon_k)k})$ for every fixed k, where $\lim \epsilon_k = 0$. The reason for the exponent in the running time is that D is of maximum degree 4 and hence it has treewidth at most $(1/3 + \epsilon_k)k$, as shown by Fomin et al. [9].

The further improvement to $O(n^{(1/4+\epsilon_k)k})$ is obtained by yet another idea. We partition the n edges of H into $n^{1/4}$ buckets of size $n^{3/4}$ and we consider all possible distributions of the k edges to remove into buckets. If there are many nonempty buckets, then graph D has fewer edges, because some dependencies are forced by putting the corresponding edges into different buckets. As a result, the treewidth of D decreases and the dynamic programming runs faster. The case when there are few nonempty buckets does not give a large speed-up in the dynamic programming, but the number of such distributions is small.

2 Preliminaries

Throughout the paper let $w_1, w_2, \ldots, w_n$ and $e_1, \ldots, e_n$ be sequences of respectively subsequent vertices and edges visited by H, so that $e_i = \{w_i, w_{i+1}\}$ for $i = 1, \ldots, n-1$ and $e_n = \{w_n, w_1\}$. For $i = 1, \ldots, n-1$ we call w_i the *left endpoint* of e_i and w_{i+1} the *right endpoint* of e_i. Also, w_n is the left endpoint of e_n and w_1 is its right endpoint.

We work with undirected graphs in this paper. An edge between vertices u and v is denoted either as $\{u, v\}$ or shortly as uv.

For a positive integer i we denote $[i] = \{1, \ldots, i\}$.

2.1 Connection patterns and embeddings

Formally, a k-move is a pair of sets (E^-, E^+), both of cardinality k, where $E^- \subseteq \{e_1, \ldots, e_n\}$, $E^+ \subseteq E(G)$, and $E(H) \setminus E^- \cup E^+$ is a Hamiltonian cycle. This is the most intuitive definition of a k-move, however it has a drawback, namely it is impossible to specify E^+ without specifying E^- first. For this reason instead of listing the edges of E^+ explicitly, we will define a connection pattern, which together with E^- expressed as an *embedding* fully specifies a k-move.

A *k-embedding* (or shortly: *embedding*) is any function $f : [k] \to [n]$. A *connection k-pattern* (or shortly: *connection pattern*)[1] is any perfect matching in the complete graph on the vertex set $[2k]$. We call a connection pattern *valid* when one obtains a single k-cycle from M by identifying vertex $2i$ with vertex $(2i + 1) \bmod 2k$ for every $i = 1, \ldots, k$.

Let us show that every pair (E^-, E^+) that defines a k-move has a corresponding pair of an embedding and a connection pattern, consequently giving an intuitive explanation of the above definition of embeddings and connection patterns. Consider a move $Q = (E^-, E^+)$. Let $E^- = \{e_{i_1}, \ldots, e_{i_k}\}$, where $i_1 < i_2 < \cdots < i_k$. For every $j = 1, \ldots, k$, let v_{2j-1} and v_{2j} be the left and right endpoint of e_{i_j}, respectively. An *embedding* of the k-move Q is the function $f_Q : [k] \to [n]$ defined as $f_Q(j) = i_j$ for every $j = 1, \ldots, k$. Note that f_Q is increasing. A *connection pattern* of Q is every perfect matching M in the complete graph on the vertex set $[2k]$ such that $E^+ = \{\{v_i, v_j\} \mid \{i, j\} \in M\}$. Note that at least one such matching always exists, and if E^- contains two incident edges then there is more than one such matching. Note also that M is valid, because otherwise after applying the k-move Q we do not get a Hamiltonian cycle.

Conversely, consider a pair (f, M), where f is an increasing embedding and M is a valid connection pattern. We define $E^-_f = \{e_{f(j)} \mid j = 1, \ldots, k\}$. For every $j = 1, \ldots, k$, let v_{2j-1} and v_{2j} be the left and right endpoint of $e_{f(j)}$, respectively. Then we also define $E^+_{(f,M)} = \{v_i v_j \mid \{i, j\} \in M\}$. It is easy to see that $(E^-_f, E^+_{(f,M)})$ is a k-move.

Because of the equivalence shown above, in what follows we abuse the notation slightly and a k-move Q can be described both by a pair of edges to remove and add (E^-_Q, E^+_Q) and by an embedding-connection pattern pair (f_Q, M_Q). The *gain* of Q is defined as $\mathrm{gain}(Q) = w(E^-_Q) - w(E^+_Q)$. Given a connection pattern M and an embedding f, we can also define an M-gain of f, denoted by $\mathrm{gain}_M(f) = \mathrm{gain}(Q)$, where Q is the k-move defined by (f, M). Note that k-OPT OPTIMIZATION asks for a k-move with maximum gain.

2.2 Tree decomposition and nice tree decomposition

To make the paper self-contained, in this section we recall the definitions of tree and path decompositions and state their basic properties which will be used later in the paper. The content of this section comes from the textbook of Cygan et al. [6].

A *tree decomposition* of a graph G is a pair $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$, where T is a tree whose every node t is assigned a vertex subset $X_t \subseteq V(G)$, called a bag, such that the following three conditions hold:

(T1) $\bigcup_{t \in V(T)} X_t = V(G)$.
(T2) For every $uv \in E(G)$, there exists a node t of T such that $u, v \in X_t$.
(T3) For every $u \in V(G)$, the set $\{t \in V(T) \mid u \in X_t\}$ induces a connected subtree of T.

[1] We note that the notion of connection pattern of a k-move was essentially introduced by de Berg et al. [8] under the name of 'signature', though they used a permutation instead of a matching, which we find more natural.

The *width* of tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$, denoted by $w(\mathcal{T})$, equals $\max_{t \in V(T)} |X_t| - 1$. The *treewidth* of a graph G, denoted by $\mathrm{tw}(G)$, is the minimum possible width of a tree decomposition of G. When E is a set of edges and $V(E)$ the set of endpoints of all edges in E, by $\mathrm{tw}(E)$ we denote the treewidth of the graph $(V(E), E)$.

A *path decomposition* is a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$, where T is a path. Then $\mathcal{T}$ is more conveniently represented by a sequence of bags $(X_1, \ldots, X_{|V(T)|})$, corresponding to successive vertices of the path. The *pathwidth* of a graph G, denoted by $\mathrm{pw}(G)$, is the minimum possible width of a path decomposition of G.

In what follows we frequently use the notion of *nice tree decomposition*, introduced by Kloks [17]. These tree decompositions are more structured, making it easier to describe dynamic programming over the decomposition. A tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ can be rooted by choosing a node $r \in V(T)$, called the root of T, which introduces a natural parent-child and ancestor-descendant relations in the tree T. A rooted tree decomposition $(T, \{X_t\}_{t \in V(T)})$ is *nice* if $X_r = \emptyset$, $X_\ell = \emptyset$ for every leaf ℓ of T, and every non-leaf node of T is of one of the following three types:

- **Introduce node**: a node t with exactly one child t' such that $X_t = X_{t'} \cup \{v\}$ for some vertex $v \notin X_{t'}$.
- **Forget node**: a node t with exactly one child t' such that $X_t = X_{t'} \setminus \{w\}$ for some vertex $w \in X_{t'}$.
- **Join node**: a node t with two children t_1, t_2 such that $X_t = X_{t_1} = X_{t_2}$.

A path decomposition is nice when it is nice as tree decomposition after rooting the path in one of the endpoints. (Note that it does not contain join nodes.)

▶ **Proposition 1** (see Lemma 7.4 in [6]). *Given a tree (resp. path) decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G of width at most k, one can in time $O(k^2 \cdot \max(|V(T)|, |V(G)|))$ compute a nice tree (resp. path) decomposition of G of width at most k that has at most $O(k|V(G)|)$ nodes.*

We say that (A, B) is a *separation* of a graph G if $A \cup B = V(G)$ and there is no edge between $A \setminus B$ and $B \setminus A$. Then $A \cap B$ is a *separator* of this separation.

▶ **Lemma 2** (see Lemma 7.3 in [6]). *Let $(T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of a graph G and let ab be an edge of T. The forest $T - ab$ obtained from T by deleting edge ab consists of two connected components T_a (containing a) and T_b (containing b). Let $A = \bigcup_{t \in V(T_a)} X_t$ and $B = \bigcup_{t \in V(T_b)} X_t$. Then (A, B) is a separation of G with separator $X_a \cap X_b$.*

3 The algorithm

In this section we present our algorithms for k-opt Optimization. The brute-force algorithm verifies all possible k-moves. In other words, it iterates over all possible valid connection patterns and increasing embeddings. The brilliant observation of Berg et al. [8] is that we can iterate only over all possible connection patterns, whose number is bounded by $(2k)!$. In other words, we fix a valid connection pattern M and from now on, our goal is to find an increasing embedding $f : [k] \to [n]$ which, together with M, defines a k-move giving the largest weight improvement over all k-moves with connection pattern M. Instead of doing this by enumerating all $\Theta(n^k)$ embeddings, Berg et al. [8] fix carefully selected $\lfloor 2/3k \rfloor$ values of f in all $n^{\lfloor 2/3k \rfloor}$ possible ways, and then show that the optimal choice of the remaining values can be found by a simple dynamic programming running in $O(nk)$ time. Our idea is to find the optimal embedding for a given connection pattern using a more efficient approach.

3.1 Basic setup

Informally speaking, instead of guessing some values of f, we guess an *approximation* of f defined by appropriate bucketing. For each approximation b, finding an optimal embedding consistent with b is done by a dynamic programming over a tree decomposition. We stress that even without bucketing (i.e, by using a single trivial bucket of size n) our algorithm works in $n^{(1/3+\epsilon_k)k}$ time. Therefore bucketing is used to further improve the running time, but it is not essential to perform the dynamic programming on a tree decomposition.

More precisely, we partition the set $[n]$, corresponding to the edges of H, into buckets. Each bucket is an interval $\{i, i+1, \ldots, j\} \subseteq [n]$, for some $1 \leq i \leq j \leq n$. Let n_b be the number of buckets and let B_j denote the j-th bucket, for $j = 1, \ldots, n_b$. A *bucket assignment* is any nondecreasing function $b : [k] \to [n_b]$.

Unless explicitly modified, we use all buckets of the same size $\lceil n^\alpha \rceil$, for a constant α which we set later. Then, for $j = 1, \ldots, b$ the j-th bucket is the set $B_j = \{(j-1)\lceil n^\alpha \rceil + 1, \ldots, j\lceil n^\alpha \rceil\} \cap [n]$.

Given a bucket assignment b we define the set

$$O_b = \{\{i, i+1\} \subset [k] \mid b(i) = b(i+1)\}.$$

▶ **Definition 3** (b-monotone partial embedding)**.** Let $f : S \to [n]$ be a partial embedding for some $S \subseteq [k]$. We say that f is b-monotone when
(M1) for every $i \in S$ we have $f(i) \in B_{b(i)}$, and
(M2) for every $\{i, i+1\} \in O_b$, if $\{i, i+1\} \subseteq S$, then $f(i) < f(i+1)$.

Note that a b-monotone embedding $f : [k] \to [n]$ is always increasing, but a b-monotone partial embedding does not even need to be non-decreasing (this seemingly artificial design simplifies some of our proofs). In what follows, we present an efficient dynamic programming algorithm which, given a valid connection pattern M and a bucket assignment b finds a b-monotone embedding of maximum M-gain. To this end, we need to introduce the gain of a partial embedding. Let $f : S \to [n]$ be a b-monotone partial embedding, for $S \subseteq [k]$. For every $j \in S$, let v_{2j-1} and v_{2j} be the left and right endpoint of $e_{f(j)}$, respectively. We define

$$E_f^- = \{e_{f(i)} \mid i \in S\}$$

$$E_f^+ = \{\{v_{i'}, v_{j'}\} \mid i, j \in S, i' \in \{2i-1, 2i\}, j' \in \{2j-1, 2j\}, \{i', j'\} \in M\}.$$

Then, $\mathrm{gain}_M(f) = w(E_f^-) - w(E_f^+)$.

Note that $\mathrm{gain}_M(f)$ does not necessarily represent the *actual* cost gain of the choice of the edges to remove represented by f. Indeed, assume that for some pair $i, j \in [k]$ there are $i' \in \{2i-1, 2i\}$ and $j' \in \{2j-1, 2j\}$ such that $\{i', j'\} \in M$. Then we say that i *interferes* with j, which means that we plan to add an edge between an endpoint of the i-th deleted edge and the j-th deleted edge. Note that if $i \in S$ (the i-th edge is chosen) and $j \notin S$ (the j-th edge is not chosen yet) this edge to be added is not known yet, and its cost is not represented in $\mathrm{gain}_M(f)$. However, the value of $f(i)$ influences this cost. Consider the following set of interfering pairs:

$$I_M = \{\{i, j\} \mid i \text{ interferes with } j\}.$$

Note that I_M is obtained from M by identifying vertex $2i-1$ with vertex $2i$ for every $i = 1, \ldots, k$ (and the new vertex is simply called i). In particular, this implies that every connected component of the graph $([k], I_M)$ is a cycle or a single edge.

3.2 Dynamic programming over tree decomposition

Now we define the graph $D_{M,b}$, called *the dependence graph*, where $V(D_{M,b}) = [k]$ and $E(D_{M,b}) = O_b \cup I_M$. The vertices of $D_{M,b}$ correspond to the k edges to be removed from H (i.e., j corresponds to the j-th deleted edge in the sequence $e_1, \ldots, e_n$). The edges of $D_{M,b}$ correspond to dependencies between the edges to remove (equivalently, elements of the domain of an embedding). The edges from O_b are *order dependencies*: edge $\{i, i+1\}$ means that the $(i+1)$-th deleted edge should appear further on H than the i-th deleted edge. In O_b there are no edges between the last element of a bucket and the first element of the next bucket, because the corresponding constraint is forced by the assignment to buckets. The edges from I_M are *cost dependencies* (resulting from interference explained in Section 3.1).

The goal of this section is a proof of the following theorem.

▶ **Theorem 4.** *Let M be a valid connection k-pattern and let $b : [k] \to [n]$ be a bucket assignment, where every bucket is of size $\lceil n^\alpha \rceil$. Then, a b-monotone embedding of maximum M-gain can be found in $O(n^{\alpha(\mathrm{tw}(D_{M,b})+1)}k^2 + 2^k)$ time.*

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a nice tree decomposition of $D_{M,b}$ with minimum width. Such a decomposition can be found in $O^*(1.7347^k)$ time by an algorithm of Fomin and Villanger [11], though for practical purposes a simpler $O^*(2^k)$-time algorithm is advised by Bodlaender et al. [2]. For every $t \in V(T)$ we denote by V_t the union of all the bags in the subtree of T rooted in t.

For every node $t \in V(T)$, and for every b-monotone function $f : X_t \to [n]$, we will compute the following value.

$$T_t[f] = \max_{\substack{g : V_t \to [n] \\ g|_{X_t} = f \\ g \text{ is } b\text{-monotone}}} \mathrm{gain}_M(g).$$

Then, if r is the root of T, and $\emptyset$ denotes the unique partial embedding with empty domain, then $T_r[\emptyset]$ is the required maximum M-gain of a b-monotone embedding. The embedding itself (and hence the corresponding k-move) can be also found by using standard DP techniques. The values of $T_t[f]$ are computed in a bottom-up fashion. Let us now present the formulas for computing these values, depending on the kind of node in the tree T.

Leaf node. When t is a leaf of T, we know that $X_t = V_t = \emptyset$, and we just put $T_t[\emptyset] = 0$.

Introduce node. Assume $X_t = X_{t'} \cup \{i\}$, for some $i \notin X_{t'}$ where node t' is the only child of t. Denote $\Delta E_f^+ = E_f^+ \setminus E_{f|_{X_{t'}}}^+$. Then, we claim that for every b-monotone function $f : X_t \to [n]$,

$$T_t[f] = T_{t'}[f|_{X_{t'}}] + w(e_{f(i)}) - \sum_{\{u,v\} \in \Delta E_f^+} w(\{u,v\}). \tag{1}$$

We show that (1) holds by showing the two relevant inequalities. Let g be a function for which the maximum from the definition of $T_t[f]$ is attained. Let $g' = g|_{V_{t'}}$. Note that g' is b-monotone because g is b-monotone. Hence, $\mathrm{gain}_M(g') \leq T_{t'}[f|_{X_{t'}}]$. It follows that $T_t[f] = \mathrm{gain}_M(g) = \mathrm{gain}_M(g') + w(e_{f(i)}) - \sum_{\{u,v\} \in \Delta E_f^+} w(\{u,v\}) \leq T_{t'}[f|_{X_{t'}}] + w(e_{f(i)}) - \sum_{\{u,v\} \in \Delta E_f^+} w(\{u,v\})$.

Now we proceed to the other inequality. Assume g' is a function for which the maximum from the definition of $T_{t'}[f|_{X_{t'}}]$ is attained. Let $g : V_t \to [n]$ be the function such that

$g|_{V_{t'}} = g'$ and $g(i) = f(i)$. Let us show that g is b-monotone. The condition $(M1)$ is immediate, since g' and f are b-monotone. For $(M2)$, consider any $\{j, j+1\} \in O_b$ such that $\{j, j+1\} \subseteq V_t$. If $i \notin \{j, j+1\}$ then $g(j) < g(j+1)$ by b-monotonicity of g', so assume $i \in \{j, j+1\}$. Then $\{j, j+1\} \subseteq X_t$, for otherwise $X_t \cap X_{t'}$ does not separate j from $j+1$, a contradiction with Lemma 2. For $\{j, j+1\} \subseteq X_t$, we have $g(j) < g(j+1)$ since $f(j) < f(j+1)$. Hence g is b-monotone, which implies $T_t[f] \geq \mathrm{gain}_M(g)$. Then it suffices to observe that $\mathrm{gain}_M(g) = \mathrm{gain}_M(g') + w(e_{f(i)}) - \sum_{\{u,v\} \in \Delta E_f^+} w(\{u,v\}) = T_{t'}[f|_{X_{t'}}] + w(e_{f(i)}) - \sum_{\{u,v\} \in \Delta E_f^+} w(\{u,v\})$. This finishes the proof that (1) holds.

Forget node. Assume $X_t = X_{t'} \setminus \{i\}$, for some $i \in X_{t'}$ where node t' is the only child of t. Then the definition of $T_t[f]$ implies that

$$T_t[f] = \max_{\substack{f' : X_{t'} \to [n] \\ f'|_{X_t} = f \\ f' \text{ is } b\text{-monotone}}} T_{t'}[f']. \tag{2}$$

Join node. Assume $X_t = X_{t_1} = X_{t_2}$, for some nodes t, t_1 and t_2, where t_1 and t_2 are the only children of t. Then, we claim that for every b-monotone function $f : X_t \to [n]$ the following holds,

$$T_t[f] = T_{t_1}[f] + T_{t_2}[f] + \left(w(E_f^-) - w(E_f^+) \right), \tag{3}$$

which we prove by using arguments very similar to the ones used for the introduce nodes, and hence due to space limitations the proof is omitted and can be found in the full version [7].

Running time. Since $|V(T)| = O(k)$, in order to complete the proof of Theorem 4 it suffices to prove the following lemma.

▶ **Lemma 5.** *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a nice tree decomposition of D. Let t be a node of T. For every $i \in X_t$ let s_i be the size of the bucket assigned to i. Then, all the values of T_t can be found in time $O(k \prod_{i \in X_t} s_i)$. In particular, if all buckets are of size $\lceil n^\alpha \rceil$, then t can be processed in time $O(kn^{\alpha |X_t|})$.*

Proof. Obviously, in every leaf node the algorithm uses only $O(1)$ time.

For an introduce node, observe that evaluation of the formula (1) takes $O(k)$ time for every f, since $|\Delta E_f^+| \leq 2$ (the factor $O(k)$ is needed to read off a single value from the table). By $(M1)$, each value $f(i)$ of a b-monotone function f can be fixed in s_i ways, so the number of b-monotone functions $f : X_t \to [n]$ is bounded by $\prod_{i \in X_t} s_i$. Hence all the values of T_t are computed in time $O(k \prod_{i \in X_t} s_i)$, which is $O(kn^{\alpha |X_t|})$ when all buckets are of size $\lceil n^\alpha \rceil$.

For a forget node, a direct evaluation of (2) for all b-monotone functions $f : X_t \to [n]$ takes $O(k \prod_{i \in X_{t'}} s_i)$ time, where t' is the only child of t.

Finally, for a join node a direct evaluation of (3) takes $O(k)$ time, since $|E_f^-| \leq k$ and $|E_f^+| \leq k$. Hence all the values of T_t are computed in time $O(k \prod_{i \in X_t} s_i)$. ◀

3.3 An algorithm running in time $O(n^{(1/3+\epsilon)k})$ for k large enough

We will make use of the following theorem due to Fomin, Gaspers, Saurabh, and Stepanov [9].

▶ **Theorem 6** (Fomin et al. [9]). *For any $\epsilon > 0$, there exists an integer n_ϵ such that for every graph G with $n > n_\epsilon$ vertices,*

$$\mathrm{pw}(G) \leq \frac{1}{6}n_3 + \frac{1}{3}n_4 + \frac{13}{30}n_5 + \frac{23}{45}n_6 + n_{\geq 7} + \epsilon n,$$

where n_i is the number of vertices of degree i in G for any $i \in \{3, \ldots, 6\}$ and $n_{\geq 7}$ is the number of vertices of degree at least 7.

We actually use the following corollary, which is rather immediate.

▶ **Corollary 7.** *For any $\epsilon > 0$, there exists an integer n_ϵ such that for every multigraph G with $n > n_\epsilon$ vertices and m edges where for every vertex $v \in V(G)$ we have $2 \leq \deg_G(v) \leq 4$, the pathwidth of G is at most $(m-n)/3 + \epsilon n$.*

Proof. The corollary follows from Theorem 6 by the following chain of equalities.

$$\begin{aligned}\frac{1}{6}n_3 + \frac{1}{3}n_4 &= \frac{1}{3}\left(\frac{1}{2}n_3 + n_4\right) = \frac{1}{3}\left(\frac{1}{2}(2n_2 + 3n_3 + 4n_4) - (n_2 + n_3 + n_4)\right) \\ &= \frac{1}{3}\left(\frac{1}{2}\sum_{v \in V(G)} \deg_G(v) - n\right) = \frac{1}{3}(m-n). \qquad (4)\end{aligned}$$

◀

Let $P_k = \{\{i, i+1\} \mid i \in [k-1]\}$.

▶ **Lemma 8.** *For any $A \subseteq P_k$ we have $\mathrm{pw}(I_M \cup A) \leq |A|/3 + \epsilon_k k$, where $\lim_{k\to\infty} \epsilon_k = 0$.*

Proof. Although $([k], I_M \cup A)$ may not be of minimum degree 2, we may consider the edge multiset I'_M of the graph obtained from $([k], I_M)$ by replacing every single edge component $\{u, v\}$ by a 2-cycle uvu. Then I'_M is a cycle cover, so every vertex in multigraph $([k], I'_M \cup A)$ has degree between 2 and 4. Hence, by Corollary 7, for some sequence ϵ_k with $\lim_{k\to\infty} \epsilon_k = 0$ we have that $\mathrm{pw}(I_M \cup A) = \mathrm{pw}(I'_M \cup A) \leq (|I'_M| + |A| - k)/3 + \epsilon_k k \leq |A|/3 + \epsilon_k k$. ◀

By Lemma 8 it follows that the running time in Theorem 4 is bounded by $O(n^{(\frac{\alpha}{3}+\epsilon)k})$. If we do not use the buckets at all, i.e., $\alpha = 1$ and we have one big bucket of size n, we get the $O(n^{(1/3+\epsilon)k})$ bound. By iterating over all at most $(2k)!$ connection patterns we get the following result, which already improves over the state of the art for large enough k.

▶ **Theorem 9.** *For every fixed integer k, k-opt Optimization can be solved in time $O(n^{(1/3+\epsilon_k)k})$, where $\lim_{k\to\infty} \epsilon_k = 0$.*

3.4 An algorithm running in time $O(n^{(1/4+\epsilon)k})$ for k large enough

Let $\mathcal{M}_k$ be the set of all valid connection k-patterns.

▶ **Lemma 10.** *k-opt Optimization can be solved in time $2^{O(k \log k)} n^{c(k)}$, where*

$$c(k) = \max_{M \in \mathcal{M}_k} \min_{\alpha \in [0,1]} \max_{A \subseteq P_k} \left((1-\alpha)(k - |A|) + \alpha(\mathrm{tw}(I_M \cup A) + 1)\right). \qquad (5)$$

Proof. We perform the algorithm from Theorem 4 for each possible valid connection pattern M and every bucket assignment b, with all the buckets of size $\lceil n^{\alpha_M} \rceil$, for some $\alpha_M \in [0, 1]$. Let us bound the total running time. Let $A \subseteq P_k$ and consider a bucket assignment b such that $O_b = A$. There are $n^{(1-\alpha_M)(k-|A|)}$ such bucket assignments, and by Theorem 4 for each

of them the algorithm uses time $O(n^{\alpha_M(\mathrm{tw}(I_M \cup A)+1)}k^2 + 2^k)$. Hence the total running time is bounded by

$$\sum_{M \in \mathcal{M}_k} \sum_{A \subseteq P_k} \sum_{\substack{b:[k]\to[\lceil n/\lceil n^{\alpha_M}\rceil\rceil] \\ b \text{ nondecreasing} \\ O_b = A}} O(n^{\alpha_M(\mathrm{tw}(I_M \cup A)+1)}k^2 + 2^k) =$$

$$O(2^k) \sum_{M \in \mathcal{M}_k} \sum_{A \subseteq P_k} n^{(1-\alpha_M)(k-|A|)} \cdot n^{\alpha_M(\mathrm{tw}(I_M \cup A)+1)} \tag{6}$$

For every $M \in \mathcal{M}_k$, the optimal value of α_M can be found by a simple LP (see Section 3.6). The claim follows. ◀

▶ **Theorem 11.** *For every fixed integer k, k-opt Optimization can be solved in time $O(n^{(1/4+\epsilon_k)k})$, where $\lim_{k\to\infty} \epsilon_k = 0$.*

Proof. Fix the same value $\alpha = 3/4$ for every connection pattern M. By Lemma 8 we have $(1-\alpha)(k-|A|) + \alpha(\mathrm{tw}(I_M \cup A)+1) \leq (\frac{1}{4} + \frac{3}{4k} + \frac{3}{4}\epsilon'_k)k$. The claim follows by Lemma 10, after putting $\epsilon_k = \frac{3}{4k} + \frac{3}{4}\epsilon'_k$. ◀

3.5 Saving space

The algorithm from Theorem 11, as described above, uses $O(n^{(1/4+\epsilon_k)k})$ space. However, a closer look reveals that the space can be decreased to $O(n^{(1/8+\epsilon_k)k})$. This is done by exploiting some properties of the specific tree decomposition of graphs of maximum degree 4, described by Fomin et al. [9], which we used in Theorem 6.

This decomposition is obtained as follows. Let D be a k-vertex graph of maximum degree 4. As long as D contains a vertex v of degree 4, we remove v. As a result we get a set of removed vertices S and a subgraph $D' = D - S$ of maximum degree 3. Then we construct a tree decomposition $\mathcal{T}'$ of D', of width at most $(1/6+\epsilon_k)k$, given in the paper of Fomin and Høie [10]. The tree decomposition $\mathcal{T}$ of D is then obtained by adding S to every bag of $\mathcal{T}'$. An inductive argument (see [9]) shows that the width of $\mathcal{T}$ is at most $\frac{1}{3}k_4 + \frac{1}{6}k_3 + \epsilon_k k$.

Assume we are given a partial b-monotone embedding $f_0 : S \to [n]$, where S is the set of removed vertices mentioned in the previous paragraph. Consider the dynamic programming algorithm from Theorem 4, which finds a b-monotone embedding of maximum M-gain, for a given bucket assignment b and connection pattern M. It is straightforward to modify this algorithm so that it computes a b-monotone embedding of maximum M-gain that extends f_0. The resulting algorithm runs in time $O(n^{\alpha(\mathrm{tw}(D-S)+1)}k^2)$ and uses space $O(n^{\alpha(\mathrm{tw}(D-S)+1)})$. Recalling that $\alpha = 3/4$ and $\mathrm{tw}(D-S) \leq (1/6+\epsilon_k)k$, we get the space bound of $O(n^{(1/8+\epsilon_k)k})$. Repeating this for each of $n^{\alpha|S|}$ embeddings of S takes time $O(n^{\alpha(|S|+\mathrm{tw}(D-S)+1)})$ instead of $O(n^{\alpha(\mathrm{tw}(D)+1)})$ from Theorem 4. However, as explained above, the bound on $\mathrm{tw}(D)$ from Theorem 6 used in the proof of Theorem 11 is also a bound on $|S| + \mathrm{tw}(D-S)$, so the time of the whole algorithm is still bounded by $O(n^{(1/4+\epsilon_k)k})$.

Another interesting observation is that if we build set S by picking an arbitrary vertex of every edge in O_b, then $D' := D - S$ contains no edges of O_b, so it has maximum degree at most 2. It follows that $\mathrm{tw}(D') \leq 2$. Thus, in Lemma 10 we can bound $\mathrm{tw}(I_M \cup A) \leq |A| + 2$ and for $\alpha = 1/2$ we get the running time of $O(n^{k/2+3/2})$. By using the approach of fixing all embeddings of S described above, we get the space of $O(n^{\alpha \mathrm{tw}(D')}) = O(n^{3/2})$ which is less than the $\Theta(n^2)$ space needed to store all the distances of the TSP instance. The additional space can be further improved to $O(n^{1/2})$, details in the full version [7].

3.6 Small values of k

The value of $c(k)$ in Lemma 10 can be computed using a computer programme for small values of k, by enumerating all connection patterns and using formula (5) to find optimum α. We used a C++ implementation (see `http://www.mimuw.edu.pl/~kowalik/localtsp/localtsp.cpp` for the source code) including a simple $O(2^k)$ dynamic programming for computing treewidth described in the work of Bodlaender et al. [2]. For every valid connection pattern M our program finds the value of $\min_{\alpha\in[0,1]} \max_{A\subseteq P_k} ((1-\alpha)(k-|A|)+\alpha(\mathrm{tw}(I_M\cup A)+1))$ by solving a simple linear program, as follows.

$$\begin{aligned} \text{minimize} \quad & v \\ \text{subject to} \quad & v \geq (1-\alpha)(k-s)+\alpha \max_{\substack{A\subseteq P_k \\ |A|=s}} (\mathrm{tw}(I_M\cup A)+1), \quad s=0,\ldots,k-1 \\ & \alpha\in[0,1] \end{aligned}$$

We get running times for $k=5,\ldots,10$ as described in Table 1, except that for $k=5$ the running time is $n^{3\frac{2}{3}}$. Because of the practical relevance we investigated the $k=5$ case by hand. A closer look reveals that the source of hardness of this case is a single (up to isomorphism) graph $([5], I_M\cup A)$ of treewidth 3. It turns out that using a different bucket partition design one can decrease the running time to $O(n^{3.4})$. The full argument proving the theorem below requires extensive case analysis, and does not fit in the page limit of the present conference version. It can be found in the full version [7].

▶ **Theorem 12.** 5-opt Optimization *can be solved in time* $O(n^{3.4})$.

4 Lower bound for $k=4$

In this section we show a hardness result for 4-opt Optimization. More precisely, we work with the decision version, called 4-opt Detection, where the input is the same as in 4-opt Optimization and the goal is to determine if there is a 4-move which improves the weight of the given Hamiltonian cycle. To this end, we reduce the Negative Edge-Weighted Triangle problem, where the input is an undirected, complete graph G, and a weight function $w: E(G)\to\mathbb{Z}$. The goal is to determine whether G contains a triangle whose total edge-weight is negative.

▶ **Lemma 13.** *Every instance* $I=(G,w)$ *of* Negative Edge-Weighted Triangle *can be reduced in* $O(|V(G)|^2)$ *time into an instance* $I'=(G',w',C)$ *of* 4-opt Detection *such that* G *contains a triangle of negative weight iff* I' *admits an improving* 4*-move. Moreover,* $|V(G')|=O(|V(G)|)$*, and the maximum absolute weight in* w' *is larger by a constant factor than the maximum absolute weight in* w*.*

Proof. Let $V(G)=\{v_1,\ldots,v_n\}$, $V_{\mathsf{up}}=\{a_1,b_1,\ldots,a_n,b_n\}$, $V_{\mathsf{down}}=\{a'_1,b'_1,\ldots,a'_n,b'_n\}$ and $V(G')=V_{\mathsf{up}}\dot\cup V_{\mathsf{down}}$. Let W be the maximum absolute value of a weight in w. Then let $M_1=5W+1$ and $M_2=21M_1+1$ and let

$$w'(u,v)=\begin{cases} 0 & \text{if } (u,v) \text{ is of the form } (a_i,b'_i) \\ w(v_i,v_j) & \text{if } (u,v) \text{ is of the form } (a_i,b_j) \text{ for } i<j \text{ or } (a'_i,b_j) \text{ for } j<i \\ M_1 & \text{if } (u,v) \text{ is of the form } (a_i,b_i) \\ -3M_1 & \text{if } (u,v) \text{ is of the form } (a'_i,b'_i) \\ -M_2 & \text{if } (u,v) \text{ is of the form } (b_i,a_{i+1}) \text{ or } (b'_i,a'_{i+1}) \text{ or } (a_1,a'_1) \text{ or } (b_n,b'_n) \\ M_2 & \text{in other case.} \end{cases}$$

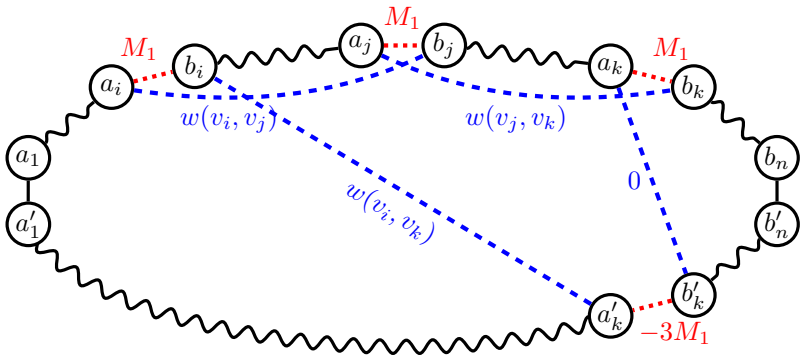


Figure 1 A simplified view of the instance (G', w', C) together with an example of a 4-move. The added edges are marked as blue (dashed) and the removed edges are marked as red (dotted).

Note that the cases are not overlapping. (Note also that although some weights are negative, we can get an equivalent instance with nonnegative weights by adding M_2 to all the weights.) The construction is illustrated in Fig. 1

If there is a negative triangle v_i, v_j, v_k for some $i < j < k$ in G then we can improve C by removing edges $(a_i, b_i), (a_j, b_j), (a_k, b_k)$ and (a'_k, b'_k) and inserting edges $(a_i, b_j), (a_j, b_k), (a_k, b'_k)$ and (a'_k, b_i). The total weight of the removed edges is $M_1 + M_1 + M_1 + (-3M_1) = 0$ and the total weight of the inserted edges is $w(v_i, v_j) + w(v_j, v_k) + 0 + w(v_k, v_i) < 0$ hence indeed the cycle is improved.

The proof in the other direction is presented in a shortened form due to space constraints (see the full version [7] for a more elaborate proof). Let us assume that C can be improved by removing 4 edges and inserting 4 edges. Note that all the edges of weight $-M_2$ belong to C and all the edges of weight M_2 do not belong to C. By the way the weights M_1 and M_2 are defined, we treat edges of weights $\pm M_2$ as fixed, i.e., they cannot be inserted or removed from the cycle in any improving 4-move. Note that the edges of C that can be removed are only the edges of the form (a_i, b_i) (of weights M_1) and (a'_i, b'_i) (of weights $-3M_1$).

All the edges of weight $-3M_1$ already belong to C, and in the next step we prove that we cannot remove more than one edge of the weight $-3M_1$ from C. Also, if we do remove one edge of the weight $-3M_1$ (i.e., of the form (a'_i, b'_i)) from C we need to remove also three edges of the weights M_1 (i.e., of the form (a_j, b_j)) in order to compensate the loss of $3M_1$.

Next, we investigate the possible locations of removed edges in an improving 4-move. We show, that if any edge is removed, then exactly three edges of the form (a_i, b_i) and exactly one edge of the form (a'_j, b'_j) have to be removed. Note that this implies also that the total weight of the removed edges has to be equal to zero.

Clearly the move has to remove at least one edge in order to improve the weight of the cycle. Let us assume that the removed edges are (a_i, b_i), (a_j, b_j) and (a_k, b_k) for some $i < j < k$ and (a'_ℓ, b'_ℓ) for some ℓ. We argue that in order to obtain a Hamiltonian cycle one of the inserted edges has to be the edge (a'_ℓ, b_i). Also the vertex b_j has to be connected with something but the vertex a'_ℓ is already taken and hence it has to be connected with the vertex a_i. Similarly the vertex b_k has to be connected with a_j because a'_ℓ and a_i are already taken. Thus a_k has to be connected with b'_ℓ and this means that $k = \ell$. The total weight change of the move is negative and therefore the total weight of the added edges has to be negative. Thus we have $w(v_i, v_j) + w(v_j, v_k) + w(v_k, v_i) = w'(a_i, b_j) + w'(a_j, b_k) + w'(a'_k, b_i) + w'(a_k, b'_k) < 0$. So v_i, v_j, v_k is a negative triangle in (G, w). ◀

▶ **Theorem 14.** *If there is $\epsilon > 0$ such that* 4-opt Detection *admits an algorithm in time $O(n^{3-\epsilon} \cdot \text{polylog}(M))$, then there is $\delta > 0$ such that both* Negative Edge-Weighted Triangle *and* All Pairs Shortest Paths *admit an algorithm in time $O(n^{3-\delta} \cdot \text{polylog}(M))$, where in all cases we refer to n-vertex input graphs with integer weights from $\{-M, \ldots, M\}$.*

Proof. The first part of the claim follows from Lemma 13, while the second part follows from the reduction of All Pairs Shortest Paths to Negative Edge-Weighted Triangle by Vassilevska-Williams and Williams (Theorem 1.1 in [26]). ◀

References

1. Sanjeev Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)*, 45(5):753–782, 1998.
2. Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On exact algorithms for treewidth. *ACM Trans. Algorithms*, 9(1):12:1–12:23, 2012.
3. Barun Chandra, Howard J. Karloff, and Craig A. Tovey. New results on the old k-opt algorithm for the traveling salesman problem. *SIAM J. Comput.*, 28(6):1998–2029, 1999.
4. Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document, 1976.
5. Georges A Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812, 1958.
6. Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.
7. Marek Cygan, Lukasz Kowalik, and Arkadiusz Socala. Improving TSP tours using dynamic programming over tree decomposition. *CoRR*, abs/1703.05559, 2017. URL: `http://arxiv.org/abs/1703.05559`.
8. Mark de Berg, Kevin Buchin, Bart M. P. Jansen, and Gerhard J. Woeginger. Fine-grained complexity analysis of two classic TSP variants. In *ICALP*, volume 55 of *LIPIcs*, pages 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
9. Fedor V. Fomin, Serge Gaspers, Saket Saurabh, and Alexey A. Stepanov. On two techniques of combining branching and treewidth. *Algorithmica*, 54(2):181–207, 2009.
10. Fedor V. Fomin and Kjartan Høie. Pathwidth of cubic graphs and exact algorithms. *Inf. Process. Lett.*, 97(5):191–196, 2006.
11. Fedor V. Fomin and Yngve Villanger. Finding Induced Subgraphs via Minimal Triangulations. In Jean-Yves Marion and Thomas Schwentick, editors, *27th International Symposium on Theoretical Aspects of Computer Science*, volume 5 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 383–394, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.STACS.2010.2470`.
12. Jiong Guo, Sepp Hartung, Rolf Niedermeier, and Ondrej Suchý. The parameterized complexity of local search for tsp, more refined. *Algorithmica*, 67(1):89–110, 2013.
13. Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
14. Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000. `doi:10.1016/S0377-2217(99)00284-2`.
15. David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. How easy is local search? *J. Comput. Syst. Sci.*, 37(1):79–100, 1988.
16. Richard M Karp. Dynamic programming meets the principle of inclusion and exclusion. *Operations Research Letters*, 1(2):49–51, 1982.

17 Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

18 Mark W. Krentel. On finding and verifying locally optimal solutions. *SIAM J. Comput.*, 19(4):742–749, 1990.

19 Marvin Künnemann and Bodo Manthey. Towards understanding the smoothed approximation ratio of the 2-opt heuristic. In *ICALP (1)*, volume 9134 of *Lecture Notes in Computer Science*, pages 859–871. Springer, 2015.

20 S. Lin and Brian W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973. doi:10.1287/opre.21.2.498.

21 Shen Lin. Computer solutions of the traveling salesman problem. *The Bell System Technical Journal*, 44(10):2245–2269, 1965.

22 Bodo Manthey and Rianne Veenstra. Smoothed analysis of the 2-opt heuristic for the TSP: polynomial bounds for gaussian noise. In *ISAAC*, volume 8283 of *Lecture Notes in Computer Science*, pages 579–589. Springer, 2013.

23 Dániel Marx. Searching the k-change neighborhood for TSP is w[1]-hard. *Oper. Res. Lett.*, 36(1):31–36, 2008.

24 Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991. doi:10.1137/1033004.

25 András Sebö and Jens Vygen. Shorter tours by nicer ears: 7/5-approximation for the graph-tsp, 3/2 for the path version, and 4/3 for two-edge-connected subgraphs. *Combinatorica*, 34(5):597–629, 2014.

26 Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *FOCS*, pages 645–654. IEEE Computer Society, 2010.

On Minimizing the Makespan When Some Jobs Cannot Be Assigned on the Same Machine*

Syamantak Das[1] and Andreas Wiese[2]

1 **University of Bremen, Bremen, Germany**
syamanta@uni-bremen.de
2 **Department of Industrial Engineering and Center for Mathematical Modeling, Universidad de Chile, Chile**
awiese@dii.uchile.cl

Abstract

We study the classical scheduling problem of assigning jobs to machines in order to minimize the makespan. It is well-studied and admits an EPTAS on identical machines and a $(2-1/m)$-approximation algorithm on unrelated machines. In this paper we study a variation in which the input jobs are partitioned into *bags* and no two jobs from the same bag are allowed to be assigned on the same machine. Such a constraint can easily arise, e.g., due to system stability and redundancy considerations. Unfortunately, as we demonstrate in this paper, the techniques of the above results break down in the presence of these additional constraints.

Our first result is a PTAS for the case of identical machines. It enhances the methods from the known (E)PTASs by a finer classification of the input jobs and careful argumentations why a good schedule exists after enumerating over the large jobs. For unrelated machines, we prove that there can be no $(\log n)^{1/4-\epsilon}$-approximation algorithm for the problem for any $\epsilon > 0$, assuming that $\mathsf{NP} \not\subseteq \mathsf{ZPTIME}(2^{(\log n)^{O(1)}})$. This holds even in the restricted assignment setting. However, we identify a special case of the latter in which we can do better: if the same set of machines we give an 8-approximation algorithm. It is based on rounding the LP-relaxation of the problem in phases and adjusting the residual fractional solution after each phase to order to respect the bag constraints.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases approximation algorithms, scheduling, makespan minimization

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.31

1 Introduction

Minimizing the makespan is a classical problem in scheduling [8, 9]. Given a set of machines M and set of jobs J, we seek to assign each job to a machine. In the setting where all machines are *identical*, the processing time of each job j is given by a value p_j for each job j. For *unrelated* machines the processing time of a job j can depend on the machine i on which it is scheduled. In this case the input contains a value $p_{ij} \in \mathbb{R}_0^+ \cup \{\infty\}$ for each combination of a machine i and a job j. The objective is to minimize the makespan, i.e., the maximum load of a machine i which is the total processing time of jobs assigned to i. The problem is well-studied, for identical machines it is strongly NP-hard and there are PTASs [11, 17] and even EPTASs, e.g., [12, 13, 10]. For unrelated machines there is a 2-approximation algorithm

* This work was partially supported by the Millennium Nucleus Information and Coordination in Networks ICM/FIC RC130003.

due to Lenstra, Shmoys, and Tardos [16], an improvement to $2 - 1/m$ due to Shchepin and Vakhania [19], and a lower bound of 3/2 [16].

In practice, one often finds side constraints in addition to the above scheduling setting that make the problem harder. A typical constraint is that some jobs have to be assigned on different machines. For instance, on-board computers of aeroplanes typically have several CPUs (modeled as machines) and for system stability considerations some tasks need to be executed on different CPUs [6]. The idea is that if one CPU fails then the plane still continues to operate safely. Minimizing the makespan is closely related to the bin packing problem where the bins and the items correspond to the machines and the jobs, respectively. There are several applications of bin packing where the items are partitioned into groups and no two items from the same group can be assigned to the same bin, for instance in distributed systems and other settings, see [18].

To model the above, in this paper we assume that the input jobs are partitioned into *bags* $J = B_1 \dot\cup B_2 \dot\cup ... \dot\cup B_b$ and that no two jobs from the same bags are allowed to be assigned on the same machine. We call these new requirements the *bag-constraints*.

In this paper we study the problem of minimizing the makespan on identical and unrelated machines with bag-constraints.

1.1 Identical machines

The known (E)PTASs [12, 13, 10, 11, 17] for minimizing the makespan on identical machines follow the idea of enumerating the solution for the large jobs, e.g., jobs that are larger than $\epsilon \cdot OPT$, and then adding the small jobs via a greedy algorithm. More precisely, one enumerates patterns for the large jobs that indicate how many large jobs of each size are assigned on each machine. The large jobs are then assigned according to these patterns and it does not matter which exact job is assigned to which slot of each pattern as long as the size of the slot is respected. In the case of bag-constraints this unfortunately does not work directly anymore. One can still enumerate the mentioned patterns and, with some additional effort, assign the large jobs to them such that they respect the bag-constraints. However, we cannot guarantee that the large jobs are assigned exactly like in the optimal solution. It could be that the jobs from the different bags are distributed completely differently on the machines than in the optimal solution (while still respecting the enumerated slots). In fact, there are instances for which the above procedure can lead to an assignment of the large jobs such that *any* solution for the remaining jobs has a makespan of at least $(2 - O(\epsilon))OPT$, see Figure 1 for an example.

Hence, we need additional ideas for the setting with bag-constraints. First, we observe that in the mentioned example many bags have relatively many large jobs (more than $\epsilon \cdot m$ many). There can be only $O_\epsilon(1)$ such *large bags* and hence we can afford to be more careful for them when we enumerate their large jobs. Indeed, we manage to assign the large jobs in such bags as in an optimal solution. Then we assign all other large jobs according to the enumerated pattern such that we respect the bag-constraints. To assign the remaining (non-large) jobs, we partition them into medium and small jobs such that the total processing time of the medium jobs is small, at most $\epsilon^2 OPT \cdot m$. We find a way to assign the latter to the machines such that via some swapping and charging arguments we can guarantee that for the remaining small jobs there exists a solution with small overall makespan. For the small jobs the argumentation is again not as easy as without the bag-constraints since some machines already have jobs from some bags which prevents small jobs of such bags to be assigned to them. We solve the remaining problem with a combination of a dynamic programming algorithm and a modified greedy routine. Overall, we obtain a PTAS for minimizing the makespan under bag-constraints. Hence, like without bag-constaints, there is

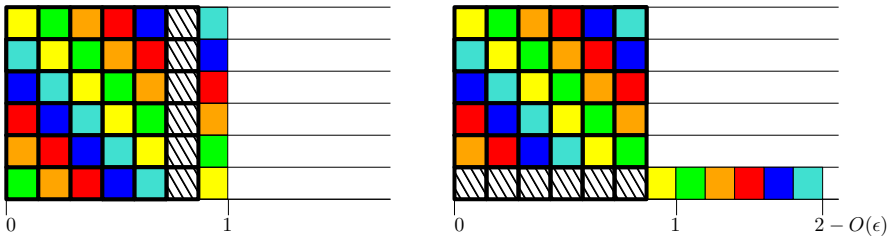


Figure 1 Left: an optimal schedule for the given instance. The bold lines indicate the enumerated patterns for the big jobs, all of them having size ϵ. The colors show the different bags of the jobs. Each white (striped) job j is in a (private) bag that contains only j. Right: a schedule in which the big jobs are assigned according to the same patterns but differently than in OPT. Thus, all non-big jobs have to be assigned to the last machine in order to satisfy the bag-constraints. This yields an approximation ratio of $2 - O(\epsilon)$.

a $(1+\epsilon)$-approximation in polynomial time, but clearly new ideas are necessary to construct such an algorithm.

▶ **Theorem 1.** *There is a PTAS for minimizing the makespan on identical machine with bag-constraints.*

1.2 Unrelated machines

For makespan minimization on unrelated machines, the mentioned LP-based 2- and $(2-1/m)$-approximation algorithms [16, 19] are known. There are several rounding strategies for the natural LP such as an argumentation via bipartite matchings [16], rounding via (sparse) extreme point solutions [19], and, related to the latter, iterated rounding [20]. The bag-constraints induce a linear constraint for each combination of a bag and a machine. Thus, it seems natural to enhance the normal LP by these constraints and try to adapt one of the known rounding techniques. However, in this paper we show that this is deemed to fail. We prove that on unrelated machines the problem is hard to approximate with a ratio of $(\log n)^{1/4-\epsilon}$ for any $\epsilon > 0$. This holds also in the restricted assignment case where each job j has a size p_j and there are some machines on which it cannot be assigned, i.e, $p_{ij} \in \{p_j, \infty\}$ for each job j and each machine i. On the other hand, we show that a randomized rounding algorithm yields a $O(\log n / \log\log n)$-approximation.

▶ **Theorem 2.** *For minimizing the makespan on unrelated machines with bag-constraints there can be no $(\log n)^{1/4-\epsilon}$-approximation algorithm for any $\epsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{ZPTIME}(2^{(\log n)^{O(1)}})$. This holds even for the restricted assignment case.*

Thus, in contrast to the case of identical machines we see here an increase in complexity due to the bag-constraints. However, we identify a special case of the restricted assignment setting where we can do better than in the general case: if all jobs from each bag can be assigned to exactly the same set of machines then we obtain a 8-approximation algorithm based on the above mentioned LP. For this we need several new ideas on top of the above mentioned known rounding techniques. First, we round the job sizes to powers of 2 and process the jobs in groups according to their sizes. We show that if all jobs have exactly the same size then even with the bag-constraints the LP is almost exact. We then assign the jobs with largest size via LP-rounding. Together with the fractional solution of the remaining jobs this might violate the bag-constraints. However, by carefully exploiting the properties of our special case we are able to construct a new fractional solution for the remaining jobs

that satisfies the bag-constraints. We continue iteratively. When we change the residual fractional solution after each iteration we employ some careful geometric sum arguments in order to ensure that the final solution has a makespan that is at most by a factor 8 larger than the value of the initial LP-solution.

▶ **Theorem 3.** *There is a 8-approximation algorithm for minimizing the makespan with bag-constraints in the restricted assignment case if for each bag all jobs in the bag can be assigned to the same set of machines.*

Due to space constraints we give the statement of several lemmas and theorems without proofs and details in this extended abstract.

1.3 Other related work

Makespan. For the restricted assignment case without bag-constraints, Svensson [21] gave an estimation algorithm with a ratio of $33/17+\epsilon$, i.e., his algorithm can estimate the optimal makespan up to this factor in polynomial time but does not necessarily find the corresponding schedule within this time bound. This algorithm was improved recently by Jansen and Rohwedder [15] to an $(11/6+\epsilon)$-estimation algorithm. If there are only two different jobs sizes, there is even a 5/3-estimation due to Jansen, Land, and Maack [14]. Moreover, for the special case that each job has either size 1 or size ϵ there is a $(2-\delta)$-approximation algorithm due to Chakrabarty, Khanna, and Li for a small constant $\delta > 0$ [2]. In contrast to the previous algorithms, it computes the actual schedule in polynomial time. All above algorithms are based on the configuration-LP in the restricted assignment case. Note that for general unrelated machines the latter LP has an integrality gap of (asympotically) 2 [5, 22].

Scheduling with conflicts. Typically, in these settings, there is an underlying conflict graph with the jobs forming the vertices; there is an edge between any two vertices if and only if the corresponding jobs cannot be scheduled on the same machine. In [1], for example, the authors give a tight 2-approximation algorithm for makespan minimization when the underlying conflict graph is polynomial time colorable. A slightly different setting is considered in [7]. Here, an edge between any two jobs in the conflict graph dictates that they cannot be scheduled in overlapping intervals on different machines. The authors, among other results, prove that the makespan minimization problem is APX-hard even for 4 different job sizes and give 4/3-approximation algorithms for the case of three different job sizes and an exact algorithm for two different job sizes. A related setting, called the multi-level bottleneck assignment is considered in [4]. It can be thought of as a generalization of our setting where each bag has the same number of jobs and the additional restriction that each machine gets exactly the same number of jobs in a schedule. The authors prove a 2-approximation for the special case with 3 bags.

2 A PTAS for identical machines

In this section we present our PTAS for minimizing the makespan under bag-constraints on an arbitrary number of identical machines. As we will see, the standard techniques of enumerating over large jobs and then adding small jobs greedily are not sufficient since a bag can contain large and small jobs and, therefore, these jobs interact with each other much more than without the bag-constraints.

Let $\epsilon > 0$ and assume for simplicity that $1/\epsilon \in \mathbb{N}$. First, we assume that we guess the optimal makespan $T^\star$ via a binary search framework and we assume by scaling that $T^\star = 1$.

We round all job lengths to powers of $1+\epsilon$, i.e., we assume that for each job $j \in J$ we have that $p_j = (1+\epsilon)^k$ for some $k \in \mathbb{N}$. Due to this we lose at most a factor of $1+\epsilon$ in the objective.

2.1 Straight-forward approach

As mentioned in the introduction, a natural approach would be to classify jobs into large and small jobs, e.g., define a job j to be large if $p_j \geq \epsilon$ and small otherwise and to enumerate over the large jobs. More precisely, one would enumerate over the patterns of the large jobs where a pattern indicates how many large jobs of each size are assigned to a machine. Since each machine can have at most $1/\epsilon$ large jobs and there are only $O(\log_{1+\epsilon} 1/\epsilon)$ many different sizes of large jobs, this yields $(1/\epsilon)^{O(\log_{1+\epsilon} 1/\epsilon)} =: K_\epsilon$ many different patterns. Thus, in time $(m+1)^{K_\epsilon}$ we can enumerate how many machines follow each pattern and thus enumerate the machine patterns of the optimal solution (up to permutation of machines). Then, one would compute a solution for the large jobs following the enumerated patterns, e.g., via assigning them greedily to the patterns' slots. However, the computed assignment of jobs to slots might be different than in the optimal solution and if a large job is assigned to a machine i then a small job from the same bag cannot be assigned to i anymore. Figure 1 shows an example where such an algorithm enumerates the patterns of some optimal solution but then assigns the large jobs differently than OPT such that any assignment for the remaining small jobs yields a makespan of at least $(2-O(\epsilon))T^\star$. Hence, this approach does not work directly.

2.2 Refined job classification and enumeration

Instead, we use a classification of the jobs into large, medium, and small jobs. Using a standard shifting argument we define these groups such that the medium jobs have small total processing time.

▶ **Lemma 4.** *For any given instance we can compute a value $k \in \{1, ..., 1/\epsilon^2\}$ such that $\sum_{j \in J: p_j \in [\epsilon^{k+1}, \epsilon^k)} p_j \leq m \cdot \epsilon^2$.*

With the value k from Lemma 4 we define a job j to be *large* if $p_j \geq \epsilon^k$, *medium* if $p_j \in [\epsilon^{k+1}, \epsilon^k)$ and *small* if $p_j < \epsilon^{k+1}$. Note that in the example in Figure 1 there are some bags that have a large number of large jobs ($m-1$ many). We call a bag *large* if it contains at least $\epsilon \cdot m$ large or medium jobs and *small* otherwise. The following proposition shows that there can be only constantly many large bags (since otherwise the total processing time of their jobs would be bigger than m).

▶ **Proposition 5.** *There can be at most $O(1/\epsilon^{k+2})$ large bags.*

In our algorithm, we want to enumerate over patterns that contain large jobs and additionally medium jobs from large bags. However, for the large and medium jobs in large bags we want to be more careful: we want to assign them to the slots of the enumerated pattern like in an optimal solution. Since there are only $O_\epsilon(1)$ large bags, we can incorporate the enumeration of this this assignment in to the enumeration of the patterns.

Assume that after our rounding the medium and large jobs have sizes $\mathcal{S} = \{s_1, ..., s_{|\mathcal{S}|}\}$ with $|\mathcal{S}| = O(\log_{1+\epsilon}(1/\epsilon^{k+1}))$. A pattern p consists of at most $(1+\epsilon)/\epsilon^{k+1}$ *slots* (note that each machine can have at most $(1+\epsilon)/\epsilon^{k+1}$ jobs that are medium or large) where each slot is characterized by a size $s \in \mathcal{S}$ and a label that specifies either one of the $O(1/\epsilon^{k+2})$ large bags (and then only jobs from that bag can be assigned to the slot) or that the slot can be used only for jobs from small bags. If s belongs to the size of a medium job then we do not

allow the latter type of label, i.e., we allow slots of medium size only for jobs from large bags. Let $K'_\epsilon = O_\epsilon(1)$ be the total number of patterns. Hence, in time $(m+1)^{K'_\epsilon}$ we can guess a pattern for each machine such that all patterns together correspond to an optimal solution. From these patterns we can directly conclude the complete assignment of medium and large jobs from the large bags. Hence, we obtain an assignment for the latter jobs and a pattern for the large jobs in small bags.

▶ **Lemma 6.** *In time m^K, where $K = (\frac{1}{\epsilon^2}\log(\frac{1}{\epsilon}))^{O(\frac{1}{\epsilon^3})}$ we can guess the assignment of the large and medium jobs from the large bags and a pattern for each machine for the large jobs in small bags (both corresponding to an optimal solution).*

Next, we assign the large jobs of the small bags to the slots given by the enumerated pattern. We do this via a dynamic program (DP). This DP will successfully assign all remaining large jobs, however, not necessarily to the slots to which the optimal solution assigned them.

▶ **Lemma 7.** *There is a dynamic program that assigns all large jobs in small bags to the machines such that (i) no two large jobs from the same (small) bag are assigned to the same machine and (ii) for each size $s \in \mathcal{S}$ each machine i gets the same number of jobs of size s as there are slots of size s for jobs from small bags in the pattern assigned to i.*

2.3 Assignment of remaining medium jobs

So far we have assigned all large jobs and additionally all medium jobs in large bags. We want to assign the medium jobs from the small bags now. If we were allowed to assign each such job to any machine then we could distribute them evenly on the machines (essentially with some greedy algorithm) such that each machine gets at most $\frac{m\cdot\epsilon^2}{m\cdot\epsilon^{k+1}} = 1/\epsilon^{k-1}$ jobs with thus a total load of at most $\epsilon^k/\epsilon^{k-1} = \epsilon$. For the medium jobs of a small bag B we observe that up to $\epsilon \cdot m$ machines already have a large job from B assigned to them but we can still use at least $(1-\epsilon)m$ machines for the medium jobs from B. We obtain almost the same bound as above due to an assignment via a flow network that we use to round a fractional solution in which each bag distributes its medium jobs evenly among its at least $(1-\epsilon)m$ available machines.

▶ **Lemma 8.** *In polynomial time we can compute an assignment of the medium jobs of the small bags such that each machine gets at most $2/\epsilon^{k-1}$ medium jobs with a total load of at most $O(\epsilon)$ and no machine has two medium or a medium and a large job from the same bag.*

2.4 Assignment of small jobs

It remains to assign all small jobs, from the large as well as from the small bags. Note that at this point it is not even clear that after the assignment of the large and medium jobs we can add the small jobs such that the overall makespan is $1 + O(\epsilon)$ (recall the example in Figure 1). Therefore, we prove this in the following lemma.

▶ **Lemma 9.** *Given the previously computed assignment of large and medium jobs, there exists an assignment of the small jobs to the machines such that the overall makespan is bounded by $1 + O(\epsilon)$.*

Proof. Consider the (possibly infeasible) schedule S in which the small jobs are assigned as in the optimal solution and the large and medium jobs are assigned as in our so far computed solution. Let B be a bag. There might be a machine i such that two jobs of B are assigned

to i in S. Note that B has to be a small bag. Let m' be the number of these machines and call these machines and the corresponding small jobs *problematic.* Assume w.l.o.g. that B contains exactly m jobs (if not then we can add some small dummy jobs of zero length). Then there must be m' machines on which no job of B was assigned, we call these machines *free.* We take the problematic small jobs of B from their (problematic) machines and distribute them on the free machines such that no two small jobs are assigned to the same machine. We do this operation for each bag. Denote by S' the resulting schedule. We argue that our operation did not increase the load on each machine by more than $O(\epsilon)$. Suppose that we moved a problematic job j in a small bag B from some machine i to some machine i'. Since i' did not have any job from B assigned to it in S and each bag has exactly m jobs, this means that in OPT machine i' must have a medium or a large job from B assigned to it. If i' has a large job j' from B in OPT then we charge p_j to $p_{j'}$, using that $p_j < \epsilon p_{j'}$. If i' has a medium job j'' from B in OPT then we charge p_j to $p_{j''}$. Recall that we assigned the medium jobs of the small bags to the machines such that the load of each machine due to medium jobs in small bags is $O(\epsilon)$ (see Lemma 8). Hence, we can still use $p_{j''}$ to pay for one other job assigned to i' in S'.

For a machine i let $OPT_i^{\text{large}}, OPT_i^{\text{med}}, OPT_i^{\text{small}}$ denote the load in OPT due to large, medium, and small jobs, respectively, and by S_i^{med} the load due to the medium jobs in S. Thus, in S' the load of machine i is bounded by

$$OPT_i^{\text{large}} + OPT_i^{\text{med}} + OPT_i^{\text{small}} + \epsilon OPT_i^{\text{large}} + S_i^{\text{med}} \leq (1 + O(\epsilon))OPT.$$

where OPT_i^{large} bounds the load due to large jobs, $OPT_i^{\text{med}} + \epsilon OPT_i^{\text{large}}$ bounds the load due to medium jobs in large bags and reassigned small jobs from small bags, OPT_i^{small} bounds the load from non-reassinged small jobs, and S_i^{med} bounds the load from medium jobs in small bags. ◀

In order to compute an assignment of the small jobs, observe that after assigning the large and medium jobs the machines have only $O_\epsilon(1)$ different loads since there are at most $1/\epsilon^{k+1}$ jobs on each machine that are large or medium and the size of each of them comes from a set of only $O(\log_{1+\epsilon} 1/\epsilon^{k+1})$ different values. Thus, we can partition the machines into $O_\epsilon(1)$ different groups such that two machines in the same group have the same load and their medium and large jobs from large bags come from exactly the same set of large bags (there are only $O_\epsilon(1)$ possibilities for the latter property). We devise a dynamic program that assigns the small jobs to these *groups* of machines, rather than directly to machines. This DP ensures that the average load of a machine in each group is at most $1 + O(\epsilon)$ and that for each machine group and each bag B we can schedule all small jobs in B assigned to this group on its machines without violating the bag-constraint. Formally, assume that the machines are divided into groups $M_1, ..., M_{K''}$ with the above properties where $K''_\epsilon = O_\epsilon(1)$ and let α_s denote the load due to medium and large jobs of each machine in group M_s. Let β_s denote the load of the small jobs assigned to machines in group M_s. For each bag B and each machine group M_s denote by $M_s^B \subseteq M_s$ the machines in M_s that do not have a medium or large job from B assigned to it.

▶ **Lemma 10.** *There is a polynomial time algorithm that assigns the small jobs to the machine groups $M_1, ..., M_{K''}$ such that for each $s \in [K'']$ we have that from each bag B at most $|M_s^B|$ jobs are assigned to M_s and $\alpha_s + \frac{\beta_s}{|M_s|} \leq 1 + O(\epsilon)$. The algorithm runs in time $b^2(\frac{mn}{\epsilon})^{O(K'')}$, where $K'' = (\frac{1}{\epsilon})^2 \log(\frac{1}{\epsilon})$.*

Once the jobs are assigned to the groups, we need to assign the jobs to the machines within each group. For the case without the bag-constraints, one can easily show that a

simple greedy algorithm will ensure that the load of the small jobs will be almost equally distributed among the machines, i.e., the makespan of any two machines will differ by at most the size of one small job. In the setting of the bag-constraints this is no longer that easy. Consider a group M_s and let J_s denote the small jobs assigned to M_s due to Lemma 10. We group the small jobs by their respective bags, denote by $J_s^\ell := J_s \cap B_\ell$ for each bag B_ℓ. Assume w.l.o.g. that $|J_s^\ell| = |M_s^{B_\ell}|$ for each bag B_ℓ. In each iteration we assign all jobs from one bag as follows. Consider the ℓ-th iteration in which we assign the jobs in J_s^ℓ. We order the jobs in J_s^ℓ non-increasingly by length, i.e., assume that $J_s^\ell = \{j_1, j_2, ..., j_{|B_\ell|}\}$ such that $p_1 \geq p_2 \geq ... \geq p_{|B_\ell|}$. We order the machines in $M_s^{B_\ell}$ non-decreasingly by the total load that they obtained from jobs in $B_1, ..., B_{\ell-1}$ that we previously assigned to them. Let $i_1, ..., i_{|B_\ell|}$ be this order. Then for each $\ell' \in \{1, ..., |J_s^\ell|\}$ we assign job $j_{\ell'}$ to machine $i_{\ell'}$. We call this algorithm *bag-LPT*.

If for each bag B_ℓ we have that $M_s^{B_\ell} = M_s$ then we can again argue that at the end the load on any two machines differs by at most the size of one small job, i.e., ϵ^{k+1}. However, this is no longer the case if $M_s^{B_\ell} \neq M_s$ for some bag B_ℓ since then there is a machine $i \in M_s \setminus M_s^{B_\ell}$ that does not gspaet a small job from a bag B_ℓ. This happens if B_ℓ is a small bag and machine i already has a large or medium job from B_ℓ assigned to it. However, to each machine i we assigned in total at most $O(1/\epsilon^k)$ jobs in small bags that are medium or large: at most $O(1/\epsilon^k)$ large jobs since each large job has a size of at least ϵ^k and at most $O(1/\epsilon^{k-1})$ medium jobs due to Lemma 8. Hence there can be only $O(1/\epsilon^k)$ bags B_ℓ such that $i \in M_s \setminus M_s^{B_\ell}$. This allows us to bound the error due to the above by $O(1/\epsilon^k) \cdot \epsilon^{k+1} = O(\epsilon)$.

▶ **Lemma 11.** *For each group M_s bag-LPT assigns the small jobs such that each machine in M_s has a load of at most $\alpha_s + \frac{\beta_s}{|M_s|} + O(1/\epsilon^k) \cdot \epsilon^{k+1} \leq 1 + O(\epsilon)$.*

Hence, we assigned all large, medium, and small jobs such that each machine has a load of $1 + O(\epsilon)$. This completes the proof of Theorem 1.

3 Special Case of Restricted Assignment

In this section we present our 8-approximation algorithm for minimizing the makespan on unrelated machines under bag-constraints in the restricted assignment case where we additionally assume that all jobs in each bag B_ℓ can be assigned to the same set of machines.

Our starting point is the LP-relaxation for the minimization the makespan on unrelated machines as it was used in [16, 19] and we add additional inequalities for the bag-constraints to it. Let T be a guessed value of the optimal makespan. We define a linear program $LP(T)$ that models the problem of finding a solution with makespan T. Recall that for each job j there is a value p_j such that $p_{ij} \in \{p_j, \infty\}$ for each machine i. For each job j denote by M_j the set of machines i such that $p_{ij} = p_j$.

$$LP(T): \sum_{j: i \in M_j} x_{ij} p_{ij} \leq T \qquad \forall i \in M \tag{1}$$

$$\sum_{i \in M_j} x_{ij} = 1 \qquad \forall j \in J \tag{2}$$

$$\sum_{j \in B_\ell} x_{ij} \leq 1 \qquad \forall i \in M, \forall \ell \in [b] \tag{3}$$

$$x_{ij} \geq 0 \qquad \forall j \in J, i \in M_j \tag{4}$$

$$x_{ij} = 0 \qquad \text{if } p_{ij} > T, \forall j \in J, i \in M \tag{5}$$

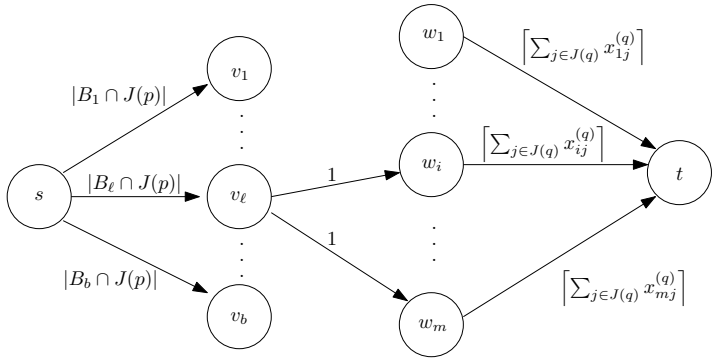


Figure 2 The flow-network for assigning the jobs in $J(p)$. Note that arcs of the type $\{v_\ell, w_i\}$ exist only if there is a job $j \in B_\ell \cap J(q)$ such that $x_{ij} > 0$. The values above the arcs indicate their respective capacities.

Using a binary search framework we determine $T^\star$ which we define to be the smallest value T for which $LP(T)$ is feasible. Denote by $x^\star$ the corresponding fractional solution. In the remainder of this section, we will prove that from $x^\star$ we can obtain an integral solution of makespan at most $4T^\star + 4\max_{i,j} p_{ij} \leq 8OPT$.

Assume w.l.o.g. that $p_{ij} \in \mathbb{N}$ for each machine $i \in M$ and each job $j \in J$. We round each finite job size p_{ij} and each value p_j to the next larger power of 2, denote by $\bar{p}_{ij}$ and $\bar{p}_j$ the new respective values. This increases the fractional load on each machine by at most a factor of 2. Based on this, we group the jobs into *classes*. A job j belongs to class q if $\bar{p}_j = 2^q$. We define $\mathrm{cl}(j)$ to be the class of a job j and $J(q)$ to be the set of all jobs of class q. Let $q_{\max}$ denote the highest class of a job in the instance. We compute our integral job assignment in phases, one phase for each job class in the order $q_{\max}, q_{\max} - 1, ..., 0$. In each phase q we determine an integral assignment of all jobs of class q.

Assume that we are given a fractional solution $\left\{x_{ij}^{(q)}\right\}_{i\in M, j\in J_{\leq}(q)}$ at the beginning of phase q that satisfies constraints (2)-(5) of $LP(T)$ for all jobs in $J_{\leq}(q)$ where for each q' we define $J_{\leq}(q') := \bigcup_{q'':q''\leq q'} J(q'')$. In the first phase where $q = q_{\max}$ this solution $x^{(q)}$ equals the optimal LP-solution $x^\star$.

3.1 Job assignment via flow network

Similarly as in the proof of Lemma 8 we interpret the fractional assignment of the jobs in $J(q)$ given by $x^{(q)}$ as the fractional solution to an instance of maximum flow with integral edge capacities. Then, using flow theory we will argue that there exists also an integral solution to this instance which will then yield our integral job assignment.

Our (directed) flow-network consists of a source node s, a node v_ℓ for each bag B_ℓ, a node w_i for each machine i, and a sink node t (see Figure 2 for a sketch). For each bag B_ℓ there is an arc (s, v_ℓ) whose capacity equals the number of jobs in B_ℓ of class q, i.e., $|B_\ell \cap J(q)|$. For each bag B_ℓ and each machine i there is an arc (v_ℓ, w_i) of capacity 1 if and only if there is a job $j \in B_\ell \cap J(q)$ such that $x_{ij} > 0$. For each machine i there is an arc (w_i, t) whose capacity equals $\lceil \sum_{j\in J(q)} x_{ij}^{(q)} \rceil$, i.e., the fractional number of jobs of class q assigned to i, rounded up. Let $G^{(p)}$ be the resulting graph and denote by $(G^{(p)}, s, t)$ the overall flow network. The solution x yields a fractional flow in this network that sends $|J(q)|$ units of flow from s to t. Hence, standard flow theory implies the following proposition.

▶ **Proposition 12.** *There is an integral flow y for $(G^{(p)}, s, t)$ that sends $|J(q)|$ units of flow from s to t.*

The integral flow due to Proposition 12 yields an assignment of the jobs in $J(q)$ that respects the bag constraints: we assign a job from a bag B_ℓ to a machine i if and only if $y_{(v_\ell, w_i)} = 1$. This yields the following lemma.

▶ **Lemma 13.** *Given a class q and a solution $x^{(q)}$ that satisfies constraints (2)-(5) of $LP(T)$ for all jobs in $J_{\leq}(q)$. Then in polynomial time we can compute an integral assignment $\{\bar{x}_{ij}^{(q)}\}_{i \in M, j \in J(q)}$ for all jobs in $J(q)$ such that (i) each machine i has at most $\lceil \sum_{j \in J(q)} x_{ij}^{(q)} \rceil$ jobs of $J(q)$ assigned to it and (ii) if for some bag B_ℓ a job $j \in B_\ell \cap J(q)$ is assigned to a machine i then there is a job $j' \in B_\ell \cap J(q)$ with $x_{ij'}^{(q)} > 0$, and (iii) the solution $\bar{x}^{(q)}$ assigns at most one job from each bag to each machine.*

3.2 Reassignment of jobs

In the next phase $q-1$ we cannot directly apply the above procedure to assign the jobs in $J(q-1)$ starting with the solution $x^{(q)}$: it might be that there is a bag B_ℓ and two jobs $j, j' \in B_\ell$ such that $\mathrm{cl}(j) = q$, $\mathrm{cl}(j') = q-1$, j is assigned to some machine i in phase q, and $x_{ij'}^{(q)} > 0$. Hence, in phase $q-1$ potentially j' is also assigned to machine i which then violates the bag constraints. Therefore, based on $x^\star$ we construct a solution $\{x_{ij}^{(q-1)}\}_{i \in M, j \in J_{\leq}(q-1)}$ in which all jobs in $J_{\leq}(q-1)$ are fractionally assigned such that if $x_{ij}^{(q-1)} > 0$ for a job $j \in J_{\leq}(q-1)$ in some bag B_ℓ then there is no job $j' \in B_\ell$ with $\mathrm{cl}(j') \geq q$ that we assigned integrally to machine i in any of the previous phases. As we will see, this might increase the load on some machines, but only by a bounded amount.

Intuitively, suppose that $x_{ij}^\star > 0$ for some job $j \in J_{\leq}(q-1)$ in some bag B_ℓ and some machine i, and assume that in phase $q' \geq q$ we assigned a job $j' \in J(q') \cap B_\ell$ to machine i. We call such a pair (i, j) *problematic*. Then $x_{ij'}^\star \leq 1 - x_{ij}^\star$ due to constraint (3). Hence, at least a fraction of $x_{ij}^\star$ of job j' was assigned to some machine $i' \neq i$ by $x^\star$. Therefore, we can move $x_{ij}^\star$ units of job j to machine i' without violating the bag constraints. Even more, if job j is of class $\tilde{q}$ (note that $\tilde{q} \leq q-1$) then $\bar{p}_j \leq 2^{\tilde{q}-q'} \bar{p}_{j'}$, i.e., the additional load on machine i' is by a factor $2^{\tilde{q}-q'}$ smaller than the original load due to job j'.

More formally, initially we define $x^{(q-1)} := x^\star$. Consider a bag B_ℓ and denote by $\tilde{M}_\ell$ the set of all machines to which the jobs in B_ℓ can be assigned and to which we did not assign a job from B_ℓ in phases $q, ..., q_{\max}$. As long as there is a problematic pair (i, j) with $x_{ij}^{(q-1)} > 0$ we reassign the "problematic fraction" $x_{ij}^{(q-1)}$ of job j greedily to the machines in $\tilde{M}_\ell$ while we ensure that each machine $i \in \tilde{M}_\ell$ gets at most $\sum_{j \in B_\ell \cap (J \setminus J_{\leq}(q-1))} x_{ij}^\star$ jobs from B_ℓ reassigned overall (fractionally). Thus, at the end for each problematic pair (i, j) we have that $x_{ij}^{(q-1)} = 0$. We perform this reassignment for each bag B_ℓ.

▶ **Lemma 14.** *Given the integral assignment $\{\bar{x}_{ij}^{(q')}\}_{i \in M, j \in J(q')}$ of the jobs in $J(q')$ for each phase $q' \geq q$ due to Lemma 13. In polynomial time we can compute a (fractional) solution $\{x_{ij}^{(q-1)}\}_{i \in M, j \in J_{\leq}(q-1)}$ for the jobs in $J_{\leq}(q-1)$ such that*

- **(i)** *$x^{(q-1)}$ satisfies constraints (2)-(5) of $LP(T)$,*
- **(ii)** *for each job $j \in J(q-1)$ in some bag B_ℓ we have that $x_{ij}^{(q-1)} = 0$ if $\bar{x}_{ij'}^{(q')} = 1$ for some job $j \in J(q') \cap B_\ell$ for some $q' \geq q$,*
- **(iii)** *for each machine i we have that $\sum_{j \in J(q-1)} x_{ij}^{(q-1)} \bar{p}_{ij} \leq \sum_{j \in J(q-1)} x_{ij}^\star \bar{p}_{ij} + \sum_{q' > q-1} 2^{(q-1)-q'} \sum_{j \in J(q')} x_{ij}^\star \bar{p}_{ij}$.*

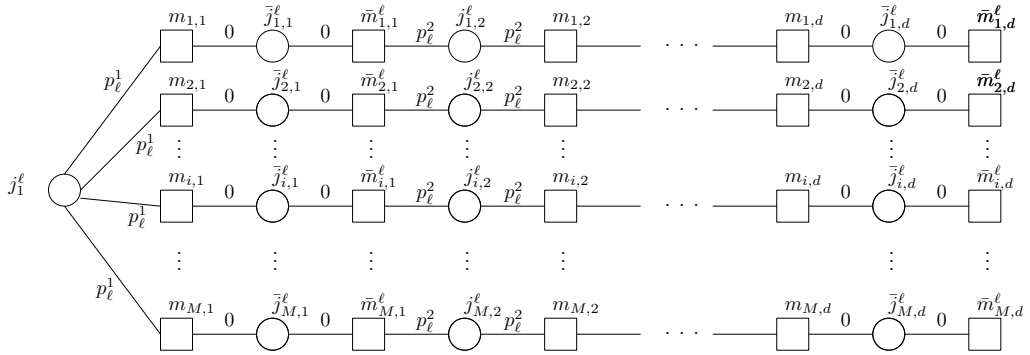


Figure 3 Sketch of the reduction from vector scheduling to group-restricted assignment.

We then proceed with phase $q-1$ where we start with the fractional solution $x^{(q-1)}$ as computed in Lemma 14. When we finish the last phase, we have computed an integral assignment $\{\bar{x}_{ij}\}_{i\in M, j\in J}$ of the jobs to the machines. Due to Lemma 13(ii) and Lemma 14(ii) our assignment respects the bag-constraints. In the next lemma we bound the load on each machine in the computed assignment which completes the proof of Theorem 3.

▶ **Lemma 15.** *In the computed assignment each machine has a load of at most* $4T^\star + 4\max_{i,j} p_{ij} \leq 8T^\star \leq 8OPT$.

4 Hardness of restricted assignment with bag-constraints

Our goal is to prove Theorem 2, i.e., we want to show that for minimizing the makespan on unrelated machines with bag-constraints there can be no $(\log n)^{1/4-\epsilon}$-approximation algorithm for any $\epsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{ZPTIME}(2^{(\log n)^{O(1)}})$, even for the restricted assignment case. We reduce the vector scheduling (VS) problem [3] to the problem of minimizing the makespan in the restricted assignment setting with bag-constraints. In the vector scheduling problem, we are given a set of identical machines M, a dimension $d \in \mathbb{N}$, and a set of n d-dimensional vectors $p_1, ..., p_n \in [0,\infty)^d$. The goal is to assign each vector to a machine, i.e., find a partition $A_1, ..., A_{|M|}$ of the vectors. The objective is to minimize $\max_{i\in M} \left\| \sum_{j\in A_i} p_j \right\|_\infty$. Our reduction is gap-preserving and in particular we will show that any c-approximation algorithm for our problem yields a c-approximation algorithm for vector scheduling for any value c. In [3] it was shown that the vector scheduling problem does not admit a c-approximation algorithm for any constant c, assuming that $\mathsf{P} \neq \mathsf{ZPP}$ (with the newer in approximability result for Independent Set in [23] it suffices to assume that $\mathsf{P} \neq \mathsf{NP}$). We are able to prove that one cannot get a $(\log n)^{1/4-\gamma}$-approximation for VS for any constant $\gamma > 0$ in polynomial time or quasi-polynomial time, assuming that $\mathsf{NP} \not\subseteq \mathsf{ZPTIME}(2^{(\log n)^{O(1)}})$. Then the same inapproximability bound holds for our problem as well.

Given an instance I of vector scheduling, defined by a number of dimensions d, a set of M identical machines, and n vectors $p_1, ..., p_n$ where for each vector p_i we denote by p_i^k its size in the k-th dimension. Denote by $OPT(I)$ its optimal objective value. We define an instance of makespan minimization on unrelated machines with bag-constraints. For each combination of a machine i and a dimension k in I we introduce a machine $m_{i,k}$, see Figure 3 for a sketch. For each vector p_ℓ we introduce a set of jobs that form a group J_ℓ. There is one job j_1^ℓ with size p_ℓ^1. The job j_1^ℓ can be assigned to each machine $m_{i,1}$ for each i. Intuitively, if j_1^ℓ is assigned to machine $m_{i,1}$ this corresponds to assigning the vector p_ℓ to machine i in I.

We will design the remaining machines and jobs for vector p_ℓ such that if j_1^ℓ is assigned to machine $m_{i,1}$ then for each dimension k

- each machine $m_{i,k}$ will get a load of p_ℓ^k from the jobs in J_ℓ and
- each machine $m_{i',k}$ with $i' \neq i$ will get a load of 0 from the jobs in J_ℓ.

Then, for each combination of a vector p_ℓ, a dimension $k \leq d$, and a machine i in I we introduce

- a dummy machine $\bar{m}_{i,k}^\ell$,
- a dummy job $\bar{j}_{i,k}^\ell$ of size 0 that can be assigned to only $m_{i,k}$ and $\bar{m}_{i,k}^\ell$, and
- if $k \geq 2$ then we also introduce a job $j_{i,k}^\ell$ of size p_ℓ^k that can be assigned to only $m_{i,k}$ and $\bar{m}_{i,k-1}^\ell$.

Observe that for each dummy machine $\bar{m}_{i,k}^\ell$ there are globally at most two jobs that can be assigned to it (and both are in J_ℓ). Denote by I' the resulting instance and denote by $OPT(I')$ its optimal solution value. Intuitively, we want to show that $OPT(I) = OPT(I')$ and that any solution $S(I)$ to I yields a solution $S(I')$ to I' with the same objective value. This needs some preparation. We have the following two lemmas.

▶ **Lemma 16.** *Let $\ell, i, k \in \mathbb{N}$. Consider any feasible solution. If the job j_1^ℓ is assigned to machine $m_{i,1}$ then each machine $m_{i,k}$ has exactly one job $j \in J_\ell$ assigned to it with $p_j = p_\ell^k$.*

The next lemma intuitively states that we can restrict ourselves to solutions to I' that are of the form described in the statement of the lemma.

▶ **Lemma 17.** *Consider any feasible solution S for the instance I' and let $i, \ell, k \in \mathbb{N}$. Further let job $j_{\ell,1}$ is assigned to machine $m_{i,1}$ in S. Then there exists another feasible solution S' such that each machine $m_{i',k}$ with $i' \neq i$ has at most one job from J_ℓ assigned to it and this job has size 0, while all other jobs have exactly the same assignment as that in S. Further, the makespan of S' is at most the makespan of S.*

▶ **Theorem 18.** *For any given instance I of the vector scheduling problem, in polynomial time we can construct an instance I' of the restricted assignment case of makespan minimization on unrelated machines with bag-constraints such that for any solution $S(I)$ for I there is a corresponding solution $S'(I')$ for I' with objective value at most that of $S(I)$ and vice versa. In particular, this implies that $OPT(I) = OPT(I')$.*

5 Conclusion

In this paper we showed that for minimizing the makespan on identical machines with bag-constraints there is a $(1+\epsilon)$-approximation algorithm like in the setting without the bag-constraints (and the problem is strongly NP-hard). However, we proved that for unrelated machines we see a change in complexity since in the classical setting the problem admits a $(2-1/m)$-approximation [19] while with the bag-constraints it seems unlikely to obtain a better approximation ratio than $(\log n)^{1/4-\epsilon}$ for any $\epsilon > 0$. It remains open to investigate more scheduling scenarios under bag-constraints such as makespan minimization on related machines or minimizing the weighted sum of completion time in any machine model. Also, for identical machines it remains open whether there is an EPTAS (which exists without the bag-constraints [12, 13, 10, 11, 17]).

Acknowledgments. We would like to thank the anonymous referees of this paper for many helpful comments and for pointing us to papers related to the setting of the bag-constraints.

References

1 Hans L. Bodlaender, Klaus Jansen, and Gerhard J. Woeginger. Scheduling with incompatible jobs. *Discrete Applied Mathematics*, 55(3):219–232, 1994.

2 Deeparnab Chakrabarty, Sanjeev Khanna, and Shi Li. On (1, ε)-restricted assignment makespan minimization. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1087–1101. SIAM, 2015.

3 Chandra Chekuri and Sanjeev Khanna. On multi-dimensional packing problems. In *SODA*, volume 99, pages 185–194. Citeseer, 1999.

4 Trivikram Dokka, Anastasia Kouvela, and Frits C. R. Spieksma. Approximating the multilevel bottleneck assignment problem. *Oper. Res. Lett.*, 40(4):282–286, 2012.

5 Tomáš Ebenlendr, Marek Krčál, and Jiří Sgall. Graph balancing: A special case of scheduling unrelated parallel machines. *Algorithmica*, 68(1):62–80, 2014.

6 Friedrich Eisenbrand, Karthikeyan Kesavan, Raju S. Mattikalli, Martin Niemeier, Arnold W. Nordsieck, Martin Skutella, José Verschae, and Andreas Wiese. *Solving an Avionics Real-Time Scheduling Problem by Advanced IP-Methods*, pages 11–22. Springer, 2010. doi:10.1007/978-3-642-15775-2_2.

7 Guy Even, Magnús M. Halldórsson, Lotem Kaplan, and Dana Ron. Scheduling with conflicts: online and offline algorithms. *J. Scheduling*, 12(2):199–224, 2009.

8 Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

9 Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

10 D. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1997.

11 Dorit S Hochbaum and David B Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162, 1987.

12 Klaus Jansen. An eptas for scheduling jobs on uniform processors: using an milp relaxation with a constant number of integral variables. *SIAM Journal on Discrete Mathematics*, 24(2):457–485, 2010.

13 Klaus Jansen, Kim-Manuel Klein, and José Verschae. Closing the Gap for Makespan Scheduling via Sparsification Techniques. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *LIPIcs*, pages 72:1–72:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2016.72.

14 Klaus Jansen, Kati Land, and Marten Maack. Estimating The Makespan of The Two-Valued Restricted Assignment Problem. In *15th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2016)*, volume 53 of *LIPIcs*, pages 24:1–24:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.SWAT.2016.24.

15 Klaus Jansen and Lars Rohwedder. On the configuration-lp of the restricted assignment problem. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*. SIAM, 2017. to appear.

16 Jan Karel Lenstra, David B. Shmoys, and Eva Tardos. Approximation algorithms for scheduling unrelated parallel machines. In *SFCS'87: Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 217–224, Washington, DC, USA, 1987. IEEE Computer Society. doi:10.1109/SFCS.1987.8.

17 Joseph YT Leung. Bin packing with restricted piece sizes. *Information Processing Letters*, 31(3):145–149, 1989.

18 Bill McCloskey and AJ Shankar. *Approaches to bin packing with clique-graph conflicts.* Computer Science Division, University of California, 2005.

19 E. V. Shchepin and N. Vakhania. An optimal rounding gives a better approximation for scheduling unrelated machines. *Operations Research Letters*, 33:127–133, 2005.

20 Mohit Singh. *Iterative methods in combinatorial optimization.* PhD thesis, Carnegie Mellon University, 2008.

21 Ola Svensson. Santa claus schedules jobs on unrelated machines. *SIAM Journal on Computing*, 41(5):1318–1341, 2012.

22 José Verschae and Andreas Wiese. On the configuration-lp for scheduling on unrelated machines. *Journal of Scheduling*, 17(4):371–383, 2014.

23 D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3:103–128, 2007.

Optimal Stopping Rules for Sequential Hypothesis Testing

Constantinos Daskalakis[*1] **and Yasushi Kawase**[†2]

1 **Massachusetts Institute of Technology, Massachusetts, USA**
costis@mit.edu

2 **Tokyo Institute of Technology, Tokyo, Japan, and**
RIKEN AIP Center, Tokyo, Japan
kawase.y.ab@m.titech.ac.jp

Abstract

Suppose that we are given sample access to an unknown distribution p over n elements and an explicit distribution q over the same n elements. We would like to reject the null hypothesis "$p = q$" after seeing *as few samples as possible*, when $p \neq q$, while we never want to reject the null, when $p = q$. Well-known results show that $\Theta(\sqrt{n}/\epsilon^2)$ samples are necessary and sufficient for distinguishing whether p equals q versus p is ϵ-far from q in total variation distance. However, this requires the distinguishing radius ϵ to be fixed prior to deciding how many samples to request. Our goal is instead to design *sequential hypothesis testers*, i.e. online algorithms that request i.i.d. samples from p and stop as soon as they can confidently reject the hypothesis $p = q$, without being given a lower bound on the distance between p and q, when $p \neq q$. In particular, we want to minimize the number of samples requested by our tests as a function of the distance between p and q, and if $p = q$ we want the algorithm, with high probability, to never reject the null. Our work is motivated by and addresses the practical challenge of sequential A/B testing in Statistics.

We show that, when $n = 2$, any sequential hypothesis test must see $\Omega\left(\frac{1}{d_{\mathrm{tv}}(p,q)^2}\log\log\frac{1}{d_{\mathrm{tv}}(p,q)}\right)$ samples, with high (constant) probability, before it rejects $p = q$, where $d_{\mathrm{tv}}(p,q)$ is the—unknown to the tester—total variation distance between p and q. We match the dependence of this lower bound on $d_{\mathrm{tv}}(p,q)$ by proposing a sequential tester that rejects $p = q$ from at most $O\left(\frac{\sqrt{n}}{d_{\mathrm{tv}}(p,q)^2}\log\log\frac{1}{d_{\mathrm{tv}}(p,q)}\right)$ samples with high (constant) probability. The $\Omega(\sqrt{n})$ dependence on the support size n is also known to be necessary. We similarly provide two-sample sequential hypothesis testers, when sample access is given to both p and q, and discuss applications to sequential A/B testing.

1998 ACM Subject Classification F.2.2 Computations on discrete structures, G.3 Probability and Statistics

Keywords and phrases property testing, sequential hypothesis testing, A/B testing

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.32

1 Introduction

A central problem in Statistics is testing how well observations of a stochastic phenomenon conform to a statistical hypothesis. A common scenario involves access to i.i.d. samples from an unknown distribution p over some set Σ and a hypothesis distribution q over the same

* Supported by NSF Awards CCF-1617730 and CCF-1650733 and ONR Grant N00014-12-1-0999.
† Supported by JSPS KAKENHI Grant Number JP16K16005.

set. The goal is to distinguish between $p = q$ and $p \neq q$. This problem, in myriads of forms, has been studied since the very beginnings of the field. Much of the focus has been on the asymptotic analysis of tests in terms the error exponents of their type I or type II errors.

More recently, the problem received attention from property testing, with emphasis on the finite sample regime. A formulation of the problem that is amenable to finite sample analysis is the following: given sample access to p and a hypothesis q as above, together with some $\epsilon > 0$, how many samples are needed to distinguish, correctly with probability at least 2/3,[1] between $p = q$ and $d(p, q) > \epsilon$, for some distance of interest d? For several distances d, we know tight answers on the number of samples required. For instance, when we take d to be the total variation distance, d_{tv},[2] we know that $\Theta(\sqrt{n}/\epsilon^2)$ samples are necessary and sufficient, where $n = |\Sigma|$ [7, 27, 33]. Tight answers are also known for other distances, variants of this problem, and generalizations [15, 8, 32, 9, 29, 13, 2, 1, 10, 12, 11], but our focus will be on distinguishing the identity of p and q under total variation distance.

While the existing literature gives tight upper and lower bounds for this problem, it still requires a lower bound ϵ on the distance between p and q when they differ, aiming for that level of distinguishing accuracy, when choosing the sample size. This has two implications:

1. Even when p and q are blatantly far from each other, the test will still request $\Theta(\sqrt{n}/\epsilon^2)$ samples, as the distance of p and q is unknown to the test when the sample size is determined.
2. When $p \neq q$, but $d_{\mathrm{tv}}(p, q) \leq O(\epsilon)$, there are no guarantees about the output of the test, just because the sample is not big enough to confidently decide that $p \neq q$.

Both issues above are intimately related to the fact that these tests predetermine the number of samples to request, as a function of the support size n and the desired distinguishing radius ϵ.

In practice, however, samples are costly to acquire. Even, when they are in abundance, they may be difficult to process. As a result, it is a common practice in clinical trials or online experimentation to "peek" at the data before an experiment is completed, in the hopes that significant evidence is collected supporting or rejecting the hypothesis. Done incorrectly this may induce statistical biases invalidating the reported significance and power bounds of the experiment [25].

Starting with a demand for more efficient testing during World War II, there has been a stream of work in Statistics addressing the challenges of *sequential hypothesis testing*; see, e.g., [34, 35, 28, 30, 26, 24, 36, 23, 22, 6, 31, 18, 20, 3, 5] and their references. These methods include the classical *sequential probability ratio test* (SPRT) [34, 35] and its generalizations [22, 6], where the alternative hypothesis is either known exactly or is parametric (i.e. p either equals q, or p is different than q, but belongs in the same parametric class as q). An alternative to SPRT methods are methods performing *repeated significance tests* (RST) [28, 26, 24, 36, 18, 5]. These methods target scalar distributions and either make parametric assumptions about p and q (e.g. Bernoulli, Gaussian, or Exponential family assumptions), or compare moments of p and q (usually their means). In particular these methods are closely related to the task of choosing the best arm in bandit settings; see, e.g., [17] and its references.

In contrast to the existing literature, we want to study categorical random variables, and do not want to make any parametric assumptions about p and q. In particular, we do not

[1] As usual the probability "2/3" in the definition of the problem can be boosted to any constant $1 - \delta$, at a cost of an extra $\log \frac{1}{\delta}$ factor in the sample complexity.

[2] See Section 2 for a definition.

want to make any assumptions about the alternatives. If the null hypothesis, $p = q$, fails, we do not know *how* it will fail. We are simply interested in determining whether $p = q$ or $p \neq q$, as soon as possible. Our goal is to devise an online policy such that, given any sequence of samples from p, the policy decides to

(i) either continue, drawing another sample from p; or
(ii) stop and declare $p \neq q$.

We want that our policy:

1. has *small error rate*, i.e. for some user-specified constants $\alpha, \beta > 0$,
   a. If $p = q$, the policy will stop, with probability at most α; i.e. the type I error is α.
   b. If $p \neq q$, the policy will stop (i.e. declare $p \neq q$), with probability at least $1 - \beta$; i.e. the type II error is β.
2. draws as few samples as possible, when $p \neq q$, in the event that it stops (which happens with probability at least $1 - \beta$).

In other words, we want to define a *stopping rule* such that, for as small a function $k = k(n, \cdot)$ as possible, the stopping time τ satisfies:

(i) $\Pr_q[\tau = +\infty] \geq 1 - \alpha$, and
(ii) $\Pr_p[\tau < k(n, d_{\mathrm{tv}}(p, q))] \geq 1 - \beta$ for all $p \neq q$,

where n is the cardinality of the set Σ on which p and q are supported. Henceforth we will call a stopping rule *proper* if it satisfies property (i) above. We want to design proper stopping rules that satisfy (ii) for as small a function $k(\cdot, \cdot)$ as possible. That is, with probability at least $1 - \beta$, we want to reject the hypothesis "$p = q$" as soon as possible. As we focus on the dependence of our stopping times on n and $d_{\mathrm{tv}}(p, q)$, we only state and prove our results throughout this paper for $\alpha = \beta = 1/3$. Changing α and β to different constants will only change the constants in our bounds. Our results are the following:[3]

1. In Theorem 1, we show that, when $n = 2$, i.e. when p and q are Bernoulli, and even when q is uniform, there is no proper stopping rule such that $k(2, d_{\mathrm{tv}}(p, q)) < \frac{1}{16 d_{\mathrm{tv}}(p,q)^2} \log\log \frac{1}{d_{\mathrm{tv}}(p,q)}$.[4,5] Our lower bound is reminiscent of the lower bound on the number of samples needed to identify the best of two arms in a bandit setting, proven in [17]. This was shown by an application of an information theoretic lower bound of Farrel for distinguishing whether an exponential family has positive or negative mean [14]. Farrel lower bounds the expected number of observations that are needed, while we show that not even a constant probability of stopping below our bound can be achieved. This is a weaker target, hence the lower bound is stronger. Finally, our goal is even weaker as we only want to determine whether $p \neq q$, but not to identify the Bernoulli with the highest mean. Our proof is combinatorial and concise.
2. In Theorem 2, we construct, for any q and n, a proper stopping rule satisfying $k(n, d_{\mathrm{tv}}(p, q)) < \frac{c\sqrt{n}}{d_{\mathrm{tv}}(p,q)^2} \log\log \frac{1}{d_{\mathrm{tv}}(p,q)}$, for some constant c. By Theorem 1 the dependence of this bound on $d_{\mathrm{tv}}(p, q)$ is optimal. Moreover, it follows from standard testing lower bounds, that the dependence on n is also optimal.[6] In fact Theorem 2 achieves something stronger. It shows that, whenever $p \neq q$, with probability at least $2/3$, the stopping rule will actually stop later than $\Omega\left(\frac{\sqrt{n}}{\chi^2(p,q)} \log\log \frac{1}{\chi^2(p,q)}\right)$ and prior to $O\left(\frac{\sqrt{n}}{d_{\mathrm{tv}}(p,q)^2} \log\log \frac{1}{d_{\mathrm{tv}}(p,q)}\right)$.

[3] The formal statements of our theorems are given in the notation introduced in Section 1.1.
[4] Note that for Bernoulli's $d_{\mathrm{tv}}(p, q)$ equals the difference of their means.
[5] Throughout the paper, we assume that log means logarithm to the base e.
[6] In particular, as we have already noted, it is known that $\Omega(\sqrt{n}/\epsilon^2)$ samples are necessary to distinguish $p = q$ from $d_{\mathrm{tv}}(p, q) > \epsilon$, for small enough constant ϵ. As our task is harder than distinguishing whether $p \neq q$ for a fixed radius of accuracy ϵ, we need to pay at least this many samples.

3. In Theorem 3, we study *two-sample sequential hypothesis testing*, where we are given sample access to both distributions p and q. Similarly to the one-sample case, our goal is to devise a stopping rule that is proper, i.e. when $p = q$, it does not stop with probability at least $2/3$, while also minimizing the samples it takes to determine that $p \neq q$. That is, when $p \neq q$, it stops, with probability at least $2/3$, after having seen as few samples as possible. We show that there is a proper stopping rule which, whenever $p \neq q$, stops after having seen $\Theta\left(\frac{n/\log n}{d_{\mathrm{tv}}(p,q)^2} \log\log \frac{1}{d_{\mathrm{tv}}(p,q)}\right)$ samples, with probability at least $2/3$. The dependence on $d_{\mathrm{tv}}(p,q)$ is optimal from Theorem 1. As our tight upper and lower bounds on the number of samples allow us to estimate $d_{\mathrm{tv}}(p,q)$ to within a constant factor, the lower bounds of [32] for estimating the distance between distributions imply that the dependence of our bounds on n is also optimal.
4. The dependence of our upper bounds on $d_{\mathrm{tv}}(p,q)$ is reminiscent of recent work in the bandit literature [19, 17] and sequential non-parametric testing [5], where stopping times with *iterated log complexity* have appeared. These results are intimately related to the Law of Iterated Logarithm [21, 4]. Our results are instead obtained in a self-contained and purely combinatorial fashion. Moreover, as discussed earlier, our testing goals are different than those in these works. While both works study scalar distributions, distinguishing them in terms of their means, we study categorical random variables distinguishing them in terms of their total variation distance.

1.1 Model

Let p, q be discrete distributions over $\Sigma = [n]$, where $[n] = \{0, 1, \ldots, n-1\}$. We assume that $n \geq 2$. In the *one-sample sequential hypothesis testing problem*, distributions q and sample access is provided to distribution p. Our goal is to distinguish between $p = q$ and $p \neq q$. Since p and q could be arbitrarily close even when they differ, our goal is to reject hypothesis $p = q$ as soon as possible when $p \neq q$, as explained below.

Let $[n]^*$ be the *Kleene star* of $[n]$, i.e., the set of all strings of finite length consisting of symbols in $[n]$. A function $T : [n]^* \to \{0, 1\}$ is called a *stopping rule* if $T(x_1 \cdots x_k) = 1$ implies $T(x_1 \cdots x_k x_{k+1} \cdots x_{k+\ell}) = 1$ for any integers $k, \ell \geq 0$ and $x_i \in [n]$ $(i = 1, \ldots, k+\ell)$. For all sequences $x \in \{0,1\}^*$, $T(x) = 1$ and $T(x) = 0$ mean respectively that the rule rejects hypothesis $p = q$ or it continues testing, after having seen x. For an infinite sequence $x = (x_1 x_2 \ldots) \in [n]^{\mathbb{N}}$, we define the *stopping time* to be the $\min\{t \mid T(x_1 \cdots x_t) = 1\}$. Let $N(a \mid x)$ be the number of times symbol $a \in [n]$ occurs in the sequence $x \in [n]^*$. Let $\tau(T, p)$ be a random variable that represents the stopping time when the sequence is generated by p, i.e, for all k:

$$\Pr[\tau(T,p) \leq k] = \sum_{x \in [n]^k} \left(T(x) \prod_{i=1}^{k} p_{x_i} \right) = \sum_{x \in [n]^k} \left(T(x) \prod_{i \in [n]} p_i^{N(i|x)} \right).$$

With the above notation, our goal in the one-sample sequential hypothesis testing problem is to find, for a given distribution q, a stopping rule T such that

(a) $\Pr[\tau(T,q) \leq k] \leq 1/3$ for any k, and

(b) $\Pr[\tau(T,p) \leq k] \geq 2/3$ for k as small as possible whenever $p \neq q$.

We call a stopping rule *proper* if it satisfies the condition (a).[7]

We also consider the *two-sample sequential hypothesis testing problem* where p and q are both unknown distributions over $[n]$, and sample access is given to both. For simplicity, this paper only studies stopping rules that use the same number of samples from each

[7] As noted earlier there is nothing special with the constants "1/3" and "2/3" here. We could turn these to any constants α and $1 - \beta$ respectively at a cost of a constant factor in our sample complexity.

distribution. This assumption increases the sample complexity by a factor of at most 2. Then a stopping rule T is defined as a function from $\bigcup_{k\in\mathbb{N}}([n]^k \times [n]^k)$ to $\{0,1\}$ such that $T(x,y)=1$ implies $T(xz,yw)=1$ for any strings $x,y,z,w\in[n]^*$ with $|x|=|y|$ and $|z|=|w|$. Here, $|x|$ represents the length of $x\in[n]^*$. The stopping time for infinite sequences $x,y\in[n]^{\mathbb{N}}$ is given by $\min\{t \mid T(x_1\cdots x_t, y_1\cdots y_t)=1\}$. Also, the stopping time $\tau(T,p,q)$ is a random variable such that

$$\Pr[\tau(T,p,q)\le k] = \sum_{x\in[n]^k}\sum_{y\in[n]^k}\left(T(x,y)\prod_{i\in[n]}p_i^{N(i|x)}\prod_{j\in[n]}q_j^{N(j|y)}\right).$$

Now, our task is to find a stopping rule T such that
(1) $\Pr[\tau(T,p,q)\le k]\le 1/3$ for any k whenever $p=q$, and
(2) $\Pr[\tau(T,p,q)\le k]\ge 2/3$ for k as small as possible whenever $p\neq q$.

Before describing our results, we briefly review notations and definitions used in the results. The *total variation distance* between p and q, denoted by $d_{\mathrm{tv}}(p,q)$, is defined to be $d_{\mathrm{tv}}(p,q)=\frac{1}{2}\sum_{i\in[n]}|p_i-q_i|=\frac{1}{2}\|p-q\|_1$. The χ^2*-distance* between p and q (which is not a true distance) is given by $\chi^2(p,q)=\sum_{i\in[n]}(p_i-q_i)^2/q_i=\left(\sum_{i\in[n]}p_i^2/q_i\right)-1$. Note that these two distances satisfy $d_{\mathrm{tv}}(p,q)^2\le\frac{1}{4}\chi^2(p,q)$ for any distributions p,q by Cauchy–Schwartz inequality.

1.2 Our results

We first prove that any proper stopping rule for the one-sample sequential hypothesis testing problem, must see $\frac{1}{16\cdot d_{\mathrm{tv}}(p,q)^2}\log\log\frac{1}{d_{\mathrm{tv}}(p,q)}$ samples before it stops, even when $n=2$, i.e. both distributions are Bernoulli, and the known distribution q is Bernoulli(0.5).

▶ **Theorem 1** (One-Sample Sequential Hypothesis Testing Lower Bound). *Even when $n=2$ and $q=(1/2,1/2)$, there exist no proper stopping rule T and positive real ϵ_0 such that*

$$\Pr\left[\tau(T,p)\le\frac{1}{16\cdot d_{\mathrm{tv}}(p,q)^2}\log\log\frac{1}{d_{\mathrm{tv}}(p,q)}\right]\ge 2/3 \quad (\textit{whenever } 0<d_{\mathrm{tv}}(p,q)<\epsilon_0). \quad (1)$$

Here, we remark that $d_{\mathrm{tv}}(p,q)=|1/2-p_0|=|1/2-p_1|$.

As we noted earlier, our lower bound involving the iterated logarithm appears similar to that of Farrel [14], but it is a slightly stronger statement. More precisely, he proved that $\limsup_{p\to q}\frac{d_{\mathrm{tv}}(p,q)^2\cdot\mathbb{E}[\tau(T,p)]}{\log\log\frac{1}{d_{\mathrm{tv}}(p,q)}}\ge c$ for a certain positive constant c. Theorem 1 implies the result but not vice versa. Also, our proof is elementary and purely combinatorial. It is given in Section 3.

We next provide a black-box reduction, obtaining optimal sequential hypothesis testers from "robust" non-sequential hypothesis testers. In particular, we use algorithms for *robust identity testing* where the goal is, given some accuracy ϵ, to distinguish whether p and q are $O(\epsilon)$-close in some distance versus $\Omega(\epsilon)$-far in some (potentially) different distance [32, 1]. We propose a schedule for repeated significance tests, which perform robust identity testing with different levels of accuracy ϵ, ultimately compounding to optimal sequential testers. In the inductive step, given the current value of ϵ, we run the non-sequential test with accuracy ϵ for $\Theta(\log\log 1/\epsilon)$ times, and take the majority vote. If the majority votes ϵ-far, we stop the procedure. Otherwise, we decrease ϵ geometrically and continue. The accuracy improvement by the $\Theta(\log\log 1/\epsilon)$-fold repetition allows the resulting stopping rule to be proper.

Our theorems for one-sample and two-sample sequential hypothesis testing are stated below and proven in Section 4. As noted earlier, stopping times involving the iterated logarithm have appeared in the multi-armed bandit and sequential hypothesis testing literature. As

explained, our testing goals are different than those in this prior work. While they study scalar distributions, distinguishing them in terms of their means, we study categorical random variables distinguishing them in terms of their total variation distance. Moreover, our results do not appeal to the law of the iterated logarithm and are obtained in a purely combinatorial fashion, using of course prior work on property testing.

▶ **Theorem 2** (One-Sample Sequential Hypothesis Testing Upper Bound). *For any known distribution q over $[n]$, there exists a proper stopping rule T and positive reals ϵ_0 and c such that*

$$\Pr\left[\frac{\sqrt{n}}{c\cdot\chi^2(p,q)}\log\log\frac{1}{\chi^2(p,q)} \le \tau(T,p) \le \frac{c\sqrt{n}}{d_{\mathrm{tv}}(p,q)^2}\log\log\frac{1}{d_{\mathrm{tv}}(p,q)}\right] \ge 2/3 \tag{2}$$

holds for any p satisfying $0 < \chi^2(p,q) < \epsilon_0$. Note that $0 < d_{\mathrm{tv}}(p,q) < \sqrt{\epsilon_0}/2$ holds when $0 < \chi^2(p,q) < \epsilon_0$ since $d_{\mathrm{tv}}(p,q)^2 \le \frac{1}{4}\chi^2(p,q)$.

▶ **Theorem 3** (Two-Sample Sequential Hypothesis Testing Upper Bound). *There exists a proper stopping rule T and positive reals ϵ_0 and c such that*

$$\Pr\left[\frac{n/\log n}{c\cdot d_{\mathrm{tv}}(p,q)^2}\log\log\frac{1}{d_{\mathrm{tv}}(p,q)} \le \tau(T,p,q) \le \frac{c\cdot n/\log n}{d_{\mathrm{tv}}(p,q)^2}\log\log\frac{1}{d_{\mathrm{tv}}(p,q)}\right] \ge 2/3 \tag{3}$$

holds for any unknown distributions p, q over $[n]$ satisfying $0 < d_{\mathrm{tv}}(p,q) < \epsilon_0$.

Since the lower bounds on the stopping time in both (2) and (3) go to infinity as p goes to q, the stopping rules never stop with probability at least $2/3$ when $p = q$. Hence, the stopping rules are proper. As noted earlier, we can improve the confidence from $2/3$ to $1-\delta$ at the cost of a multiplicative factor $\log(1/\delta)$ in the sample complexity. The dependence of both upper bounds on $d_{\mathrm{tv}}(p,q)$ is tight as per Theorem 1. The $\sqrt{n}$ dependence in Theorem 2 is tight because it is known that testing whether $d_{\mathrm{tv}}(p,q) = 0$ or $d_{\mathrm{tv}}(p,q) \ge 1/2$ requires $\Omega(\sqrt{n})$ samples [15, 8]. In addition, Theorem 3, allows us to estimate the total variation distance between p and q because the stopping time and the total variation distance satisfy the relation $\tau(T,p) = \Theta\left(\frac{n/\log n}{d_{\mathrm{tv}}(p,q)^2}\log\log\frac{1}{d_{\mathrm{tv}}(p,q)}\right)$. This and the lower bounds for estimating the ℓ_1 distance of distributions provided in [32], imply that the dependence of Theorem 3 on n is also optimal.

As a simple corollary of the above results, we can also provide an efficient algorithm for sequential A/B testing, replicating the bounds obtainable from [19, 17, 5], without appealing to the Law of the Iterated Logarithm.

▶ **Theorem 4.** *There exists an algorithm that distinguishes between the cases (a) $p > q$ and (b) $q > p$, using $\Theta\left(\frac{1}{|p-q|^2}\log\log\frac{1}{|p-q|}\right)$ samples for any unknown Bernoulli distributions with success probabilities p and q.*

2 Known Results

In this section, we state known results for robust identity testing, which we use in our upper bounds.

▶ **Theorem 5** ([2]). *For any known distribution q, there exists an algorithm with sample complexity $\Theta(\sqrt{n}/\epsilon^2)$ which distinguishes between the cases*
(a) *$\sqrt{\chi^2(p,q)} \le \epsilon/2$ and*
(b) *$d_{\mathrm{tv}}(p,q) \ge \epsilon$,*
with probability at least $2/3$.

▶ **Theorem 6** ([32]). *Given sample access to two unknown distributions p and q, there exists an algorithm with sample complexity $\Theta(\frac{n}{\epsilon^2 \log n})$ which distinguishes between the cases*
(a) $d_{\mathrm{tv}}(p,q) \leq \epsilon/2$ *and*
(b) $d_{\mathrm{tv}}(p,q) \geq \epsilon$,
with probability at least $2/3$.

We remark that, even though the proofs of Theorems 5 and 6 may use Poisson sampling, i.e., the sample complexities are Poisson distributed, we can assume that the numbers of samples are deterministically chosen. This is because the Poisson distribution is sharply concentrated around the expected value.

In our analysis of the upper and lower bounds of sample complexities, we use the following Hoeffding's inequality.

▶ **Theorem 7** (Hoeffding's inequality [16]). *Let X be a binomial distribution with n trials and probability of success p. Then, for any real ϵ, we have* $\Pr[X \leq (p-\epsilon)n] = \sum_{i=0}^{\lfloor (p-\epsilon)n \rfloor} \binom{n}{i} p^i (1-p)^{n-i} \leq \exp(-2\epsilon^2 n)$.

3 Lower bound

In this section, we prove Theorem 1, i.e., our lower bound on the sample complexity for the binary alphabet case $n = 2$. We abuse notation using p, q to denote the probabilities that our distributions output 1. In particular, $1-p$ and $1-q$ are the probabilities they output 0.

We first observe that, for any stopping rule T, the stopping times $\tau(T,p)$ and $\tau(T,q)$ take similar values when p, q are close.

▶ **Lemma 8.** *Let $p < 1/2$, $q = 1/2$, $1 > \alpha > 0$ and s, t be positive integers such that $s > t$. If* $\Pr[t \leq \tau(T,p) \leq s] \geq \alpha$, *then we have*

$$\Pr[t \leq \tau(T,q) \leq s] \geq (\alpha - \alpha^2) \cdot (1/e)^{4(\frac{1}{2}-p)^2 \cdot s + 4(\frac{1}{2}-p)\sqrt{s \log(1/\alpha)}}.$$

Proof. Let $A = \{x \in \{0,1\}^s \mid T(x_1 \dots x_{t-1}) = 0 \text{ and } T(x_1 \dots x_s) = 1\}$. Then the stopping probability for p can be written as

$$\Pr[t \leq \tau(T,p) \leq s] = \sum_{x \in A} p^{N(1|x)} (1-p)^{N(0|x)}.$$

Recall that $N(a \mid x)$ is the number of times a symbol $a \in \{0,1\}$ occurs in a string $x \in \{0,1\}^*$. Note that $|x| = N(1 \mid x) + N(0 \mid x)$. Let $A_1 = \{x \in A \mid N(1 \mid x) < p \cdot s - \sqrt{s \log(1/\alpha)}\}$ and $A_2 = \{x \in A \mid N(1 \mid x) \geq p \cdot s - \sqrt{s \log(1/\alpha)}\}$. By using Hoeffding's inequality, we have

$$\begin{aligned}\sum_{x \in A_1} p^{N(1|x)} (1-p)^{N(0|x)} &\leq \sum_{x \in \{0,1\}^s :\ N(1|x) < p \cdot s - \sqrt{s \log(1/\alpha)}} p^{N(1|x)} (1-p)^{N(0|x)} \\ &\leq \sum_{k=0}^{\lfloor p \cdot s - \sqrt{s \log(1/\alpha)} \rfloor} \binom{s}{k} p^k (1-p)^{s-k} \leq \exp\left(-2 \left(\frac{\sqrt{s \log(1/\alpha)}}{s}\right)^2 \cdot s\right) = \alpha^2.\end{aligned}$$

Hence, it holds that

$$\sum_{x \in A_2} p^{N(1|x)} (1-p)^{N(0|x)} = \sum_{x \in A \setminus A_1} p^{N(1|x)} (1-p)^{N(0|x)} \geq \alpha - \alpha^2. \tag{4}$$

In what follows, we bound the value $\Pr[t \le \tau(T,q) \le s]$. Since $A_2 \subseteq A$ and $s = N(1 \mid x) + N(0 \mid x)$, we have

$$\Pr[t \le \tau(T,q) \le s] = \sum_{x \in A} \frac{1}{2^s} \ge \sum_{x \in A_2} \frac{1}{2^s} = \sum_{x \in A_2} \frac{1}{4^{N(1|x)}} \cdot \frac{1}{2^{N(0|x)-N(1|x)}}.$$

Since $p(1-p) = -(p-1/2)^2 + 1/4 \le 1/4$, it holds that

$$\begin{aligned}\sum_{x \in A_2} \frac{1}{4^{N(1|x)}} \cdot \frac{1}{2^{N(0|x)-N(1|x)}} &\ge \sum_{x \in A_2} p^{N(1|x)}(1-p)^{N(1|x)} \cdot \left(\frac{1}{2}\right)^{N(0|x)-N(1|x)} \\ &= \sum_{x \in A_2} p^{N(1|x)}(1-p)^{N(0|x)} \cdot \left(\frac{1/2}{1-p}\right)^{N(0|x)-N(1|x)}. \end{aligned} \tag{5}$$

Note that, for $x \in A_2$, we have $N(0 \mid x) - N(1 \mid x) = s - 2N(1 \mid x) \le s - 2(p \cdot s - \sqrt{s \log(1/\alpha)}) = 2(1/2-p)s + 2\sqrt{s \log(1/\alpha)}$ since $s = N(1 \mid x) + N(0 \mid x)$ and $N(1 \mid x) \ge p \cdot s - \sqrt{s \log(1/\alpha)}$. Also, we have $\frac{1/2}{1-p} = \frac{1}{1+(1-2p)} < 1$ since $p < 1/2$. Thus, we get

$$\left(\frac{1/2}{1-p}\right)^{N(0|x)-N(1|x)} \ge \left(\frac{1}{1+(1-2p)}\right)^{2(1/2-p)s+2\sqrt{s \log(1/\alpha)}}. \tag{6}$$

Applying (6) and (4) to (5) yields

$$\begin{aligned}&\sum_{x \in A_2} p^{N(1|x)}(1-p)^{N(0|x)} \cdot \left(\frac{1/2}{1-p}\right)^{N(0|x)-N(1|x)} \\ &\quad \ge \sum_{x \in A_2} p^{N(1|x)}(1-p)^{N(0|x)} \cdot \left(\frac{1}{1+(1-2p)}\right)^{2(1/2-p)s+2\sqrt{s \log(1/\alpha)}} \\ &\quad \ge (\alpha - \alpha^2) \cdot \left(\frac{1}{1+(1-2p)}\right)^{2(1/2-p)s+2\sqrt{s \log(1/\alpha)}}.\end{aligned}$$

Here, $1 + (1-2p) \le e^{1-2p}$ holds since $1 + x \le e^x$ for any x. Therefore, we conclude that

$$\Pr[t \le \tau(T,q) \le s] \ge (\alpha - \alpha^2) \cdot (1/e)^{4(1/2-p)^2 s + 4(1/2-p)\sqrt{s \log(1/\alpha)}},$$

which is our claim. ◀

Next, we see that the stopping time $\tau(T,p)$ is not so small when T is proper.

▶ **Lemma 9.** *Suppose that $1/4 < p < 1/2$, $q = 1/2$, and T is a proper stopping rule. Then we have $\Pr\left[\tau(T,p) \le \frac{1}{10000 \cdot |p-1/2|^2}\right] \le 1/2$.*

Proof. Let $s = \left\lfloor \frac{1}{10000 \cdot |p-1/2|^2} \right\rfloor$ and $B = \{x \in \{0,1\}^s \mid T(x) = 1\}$. By the assumption that the rule is proper, we have $\Pr\left[\tau(T,q) \le \frac{1}{10000 \cdot |p-1/2|^2}\right] = \Pr[\tau(T,q) \le s] = |B|/2^s \le 1/3$. Let $B_1 = \{x \in \{0,1\}^s \mid T(x) = 1,\ |N(1 \mid x) - ps| > 2\sqrt{s}\}$ and $B_2 = \{x \in \{0,1\}^s \mid T(x) = 1,\ |N(1 \mid x) - ps| \le 2\sqrt{s}\}$. Then we have $\Pr[\tau(T,p) \le s] = \sum_{x \in B} p^{N(1|x)}(1-p)^{N(0|x)} = \sum_{x \in B_1} p^{N(1|x)}(1-p)^{N(0|x)} + \sum_{x \in B_2} p^{N(1|x)}(1-p)^{N(0|x)}$. We bound the two terms separately. By using Hoeffding's inequality, we have

$$\sum_{x \in B_1} p^{N(1|x)}(1-p)^{N(0|x)} \le 2\exp\left(-2\left(\frac{2\sqrt{s}}{s}\right)^2 \cdot s\right) = \frac{2}{e^8} < 0.1.$$

Also, we have

$$\begin{aligned}
\sum_{x\in B_2} p^{N(1|x)}(1-p)^{N(0|x)} &\le \sum_{x\in B_2} p^{ps-2\sqrt{s}}(1-p)^{(1-p)s+2\sqrt{s}} \\
&\le \sum_{x\in B}(p(1-p))^{ps}\cdot(1-p)^{(1-2p)s}\cdot\left(\frac{1-p}{p}\right)^{2\sqrt{s}} \\
&\le \frac{2^s}{3}\cdot\left(\frac{1}{4}\right)^{ps}\cdot(1-p)^{(1-2p)s}\cdot\left(1+\frac{1-2p}{p}\right)^{2\sqrt{s}} \\
&= \frac{1}{3}\cdot(1+(1-2p))^{(1-2p)s}\cdot\left(1+\frac{1-2p}{p}\right)^{2\sqrt{s}} \\
&\le \frac{1}{3}\cdot\exp\left((1-2p)^2 s+8\cdot(1-2p)\sqrt{s}\right) \qquad (7)\\
&\le \frac{1}{3}\cdot\exp\left(\frac{4}{10000}+\frac{16}{100}\right) = \frac{e^{0.1604}}{3} < 0.4.
\end{aligned}$$

Here, (7) holds since $1+x\le e^x$ for any $x\ge 0$ and $1/4<p<1/2$.

Therefore, we obtain

$$\Pr\left[\tau(T,p)\le\frac{1}{10000\cdot|p-1/2|^2}\right] < 0.1+0.4=\frac{1}{2}. \qquad \blacktriangleleft$$

Now we are ready to prove Theorem 1. Recall that $q=1/2$.

Proof of Theorem 1. To obtain a contradiction, suppose that a proper stopping rule T satisfies Condition (1) for some ϵ_0, i.e., $\Pr\left[\tau(T,p)\le\frac{\log\log\frac{1}{|p-1/2|}}{16|p-1/2|^2}\right]\ge\frac{2}{3}$ holds for any p such that $0<|p-1/2|<\epsilon_0$. By Lemma 9, we have $\Pr\left[\tau(T,p)>\frac{1}{10000\cdot|p-1/2|^2}\right]\ge\frac{1}{2}$ holds for any p such that $1/4<p<1/2$. Hence, we have

$$\Pr\left[\frac{1}{10000\cdot|p-1/2|^2}<\tau(T,p)\le\frac{\log\log\frac{1}{|p-1/2|}}{16|p-1/2|^2}\right]\ge\frac{2}{3}+\frac{1}{2}-1=\frac{1}{6}$$

for any p such that $1/2-\min\{\epsilon_0,1/4\}<p<1/2$.

Let $p(k)=1/2-1/M^{k^2}$ where k is a natural number and M is a real number that satisfies $M>\max\{e^{e^{32}},1/\epsilon_0\}$. Since $0<1/2-p(k)<1/M<\min\{\epsilon_0,1/e^{e^{32}}\}\le\min\{\epsilon_0,1/4\}$ for any $k\ge 1$, we have

$$\Pr\left[\frac{M^{2k^2}}{10000}<\tau(T,p(k))\le\frac{M^{2k^2}}{16}\log\log M^{k^2}\right]\ge\frac{1}{6}.$$

Let U_k be the interval $\left(\frac{M^{2k^2}}{10000},\frac{M^{2k^2}}{16}\log\log M^{k^2}\right]$. Then $U_i\cap U_j=\emptyset$ holds, for any distinct natural numbers i,j, because we have

$$\frac{M^{2(k+1)^2}}{10000}=\frac{M^{2k^2+4k+2}}{10000}=\frac{M^2}{10000}\cdot M^{2k^2}\cdot M^{4k}>\frac{M^{2k^2}}{16}\log\log M^{k^2}.$$

Here, we use the facts that $M^2/10000>1/16$ and $M^{4k}>\log\log M^{k^2}$. The former fact holds by $M>e^{e^{32}}>25=\sqrt{10000/16}$. The later fact holds since $M^{4k}=M^{3k}\cdot M^k>2M^k>M+M^k>\log\log M+\log k^2=\log\log M^{k^2}$ by $M>e^{e^{32}}>2$.

In what follows, we produce a contradiction by evaluating the probability

$$P(\ell) = \Pr\left[\tau(T,q) \le \frac{M^{2\ell^2}}{16}\log\log M^{\ell^2}\right]$$

for a sufficiently large integer ℓ. As the intervals U_i are disjoint, we have

$$P(\ell) \ge \Pr\left[\tau(T,q) \in \bigcup_{k=1}^{\ell} U_k\right] = \sum_{k=1}^{\ell} \Pr\left[\tau(T,q) \in U_k\right].$$

Applying Lemma 8 with $p = 1/2 - 1/M^{k^2}$, $s = \frac{M^{2k^2}}{16}\log\log M^{k^2}$, $t = \frac{M^{2k^2}}{10000}$, and $\alpha = 1/6$, we have

$$\begin{aligned}\Pr\left[\tau(T,q) \in U_k\right] &\ge \frac{5}{36}\cdot\left(\frac{1}{e}\right)^{4\cdot\frac{1}{M^{2k^2}}\cdot\frac{M^{2k^2}}{16}\log\log M^{k^2}+4\cdot\frac{1}{M^{k^2}}\sqrt{\frac{M^{2k^2}}{16}(\log\log M^{k^2})\log 6}} \\ &\ge \frac{5}{36}\cdot\left(\frac{1}{e}\right)^{\frac{1}{4}\log\log M^{k^2}+\sqrt{2\log\log M^{k^2}}}.\end{aligned}$$

Since $\frac{1}{4}\log\log x \ge \sqrt{2\log\log x}$ holds for $\log\log x \ge 32$ (i.e., $x \ge e^{e^{32}}$), we have

$$\frac{1}{4}\log\log M^{k^2} + \sqrt{2\log\log M^{k^2}} \le \frac{1}{2}\log\log M^{k^2}.$$

Hence, we obtain

$$\begin{aligned}P(\ell) &\ge \sum_{k=1}^{\ell}\frac{5}{36}\cdot\left(\frac{1}{e}\right)^{\frac{1}{4}\log\log M^{k^2}+\sqrt{\log\log M^{k^2}}} \ge \sum_{k=1}^{\ell}\frac{5}{36}\cdot\left(\frac{1}{e}\right)^{\frac{1}{2}\log\log M^{k^2}} \\ &= \sum_{k=1}^{\ell}\frac{5}{36}\cdot\left(\frac{1}{\log M^{k^2}}\right)^{1/2} = \frac{5}{36\sqrt{\log M}}\sum_{k=1}^{\ell}\frac{1}{k} \ge \frac{5}{36\sqrt{\log M}}\int_1^{\ell+1}\frac{dx}{x} = \frac{5\log(\ell+1)}{36\sqrt{\log M}}.\end{aligned}$$

By choosing $\ell = \lfloor M \rfloor$, we get $P(\lfloor M \rfloor) \ge \frac{5\log M}{36\sqrt{\log M}} = \frac{5}{36}\sqrt{\log M} > \frac{5}{36}\sqrt{\log e^{e^{32}}} > 1$, which is a contradiction. ◀

4 Upper bounds

In this section, we give stopping rules for testing identity with small sample complexity.

4.1 The case when q is explicit but p is unknown

In this subsection, we first provide a framework to obtain stopping rules from algorithms for robust identity testing and then prove Theorem 2.

We state a lemma to improve the success probability of a test by repeatedly running the test and taking a majority vote.

▶ **Lemma 10.** *Suppose that we have an algorithm for a decision problem with success probability at least* $2/3$*. Then, by running the algorithm* $\lceil 18\log(3k)\rceil$ *times and taking the majority, the success probability increases to at least* $1 - \frac{1}{9k^2}$*.*

Algorithm 1: Stopping rule T^q induced by $T^{q,\epsilon}$

input : $x_1 \cdots x_t \in [n]^*$, distributions q over $[n]$ **output:** 0 or 1

Let $s_0 = 0$;
for $k = 1, 2, \ldots$ **do**
  Let $\epsilon_k = 1/2^k$ and $s_k = s_{k-1} + f(q, \epsilon_k) \cdot \lceil 18 \log(3k) \rceil$;
  if $s_k > t$ **then return** 0;
  else if $T^{q,\epsilon_k}(x_{s_{k-1}+1} \cdots x_{s_k}) = 1$ **then return** 1;

Suppose that we have an algorithm, for a given q, with sample complexity $f(q, \epsilon)$ that distinguishes between the cases
(a) $d_1(p, q) \geq \epsilon$ and
(b) $d_2(p, q) \leq \epsilon/2$,
with probability at least $2/3$, where d_1 and d_2 are distance measures that depend on the application. Then, by Lemma 10, we can obtain a stopping rule $T^{q,\epsilon}$ such that

- $\Pr\left[\tau(T^{q,\epsilon}, p) \leq f(q, \epsilon) \cdot \lceil 18 \log(3k) \rceil\right] \geq 1 - \frac{1}{9k^2}$ if $d_1(p, q) \geq \epsilon$, and
- $\Pr\left[\tau(T^{q,\epsilon}, p) \leq f(q, \epsilon) \cdot \lceil 18 \log(3k) \rceil\right] \leq \frac{1}{9k^2}$ if $d_2(p, q) \leq \epsilon/2$.

We then formulate a stopping rule T^q for identity testing as follows. The tester guesses ϵ and then tests identity of p, q by using $T^{q,\epsilon}$. If $T^{q,\epsilon}$ does not stop with $f(q, \epsilon) \cdot \lceil 18 \log(3k) \rceil$ samples, it reduces ϵ to half and continue the procedure recursively. The stopping rule T^q is summarized as Algorithm 1.

We show that T^q is the desired stopping rule.

▶ **Lemma 11.** *If $p \neq q$, the stopping time $\tau(T^q, p)$ for T^q in Algorithm 1 satisfies* $\Pr[s_a \leq \tau(T^q, p) \leq s_b] \geq 2/3$, *where* $a = \lfloor \log_2 \frac{1}{2d_2(p,q)} \rfloor$, $b = \lceil \log_2 \frac{1}{d_1(p,q)} \rceil$, *and* $s_\ell = \sum_{k=1}^{\ell} f(q, \epsilon_k) \cdot \lceil 18 \log(3k) \rceil$.

Proof. Since $d_1(p, q) \geq 1/2^b = \epsilon_b$ by $b = \lceil \log_2 \frac{1}{d_1(p,q)} \rceil$, the stopping time is larger than s_b with probability at most

$$\begin{aligned}
\Pr\left[\tau(T^q, p) > s_b\right] &= \Pr\left[\tau(T^q, p) \geq s_b\right] \cdot \Pr\left[\tau(T^q, p) \neq s_b \mid \tau(T^q, p) \geq s_b\right] \\
&= \Pr\left[\tau(T^q, p) \geq s_b\right] \cdot \left(1 - \Pr\left[\tau(T^q, p) = s_b \mid \tau(T^q, p) \geq s_b\right]\right) \\
&\leq 1 - \Pr\left[\tau(T^q, p) = s_b \mid \tau(T^q, p) \geq s_b\right] \\
&= 1 - \Pr\left[\tau(T^{q,\epsilon_b}, p) \leq f(q, \epsilon_b) \cdot \lceil 18 \log(3b) \rceil\right] \leq \frac{1}{9} \cdot \frac{1}{b^2} \leq \frac{1}{9}.
\end{aligned}$$

On the other hand, since $d_2(p, q) \leq \frac{1}{2} \cdot \frac{1}{2^a} \leq \epsilon_k/2$ for any $1 \leq k \leq a$ by $a = \lfloor \log_2 \frac{1}{2d_2(p,q)} \rfloor$, the stopping time is smaller than s_a with probability at most

$$\begin{aligned}
\Pr\left[\tau(T^q, p) < s_a\right] &= \sum_{k=1}^{a-1} \Pr\left[\tau(T^q, p) = s_k\right] \leq \sum_{k=1}^{a-1} \Pr\left[\tau(T^q, p) = s_k \mid \tau(T^q, p) \geq s_k\right] \\
&= \sum_{k=1}^{a-1} \Pr\left[\tau(T^{q,\epsilon_k}, p) \leq f(q, \epsilon_k) \cdot \lceil 18 \log(3k) \rceil\right] \\
&\leq \sum_{k=1}^{a-1} \frac{1}{9} \cdot \frac{1}{k^2} < \sum_{k=1}^{\infty} \frac{1}{9} \cdot \frac{1}{k^2} = \frac{1}{9} \cdot \frac{\pi^2}{6} < \frac{2}{9}.
\end{aligned}$$

Hence, the stopping time of T^q satisfies

$$\Pr[s_a \le \tau(T^q, p) \le s_b] = 1 - \Pr[\tau(T^q, p) > s_b] - \Pr[\tau(T^q, p) < s_a] \ge 1 - \frac{1}{9} - \frac{2}{9} = \frac{2}{3},$$

which completes the proof. ◀

Next, we prove Theorem 2. To provide stopping rules, we use robust identity testing algorithm in Theorem 5. When T^q is the stopping rule induced by the algorithm in Theorem 5, we have $d_1(p,q) = d_{\mathrm{tv}}(p,q)$, $d_2(p,q) = \sqrt{\chi^2(p,q)}$, and $f(q,\epsilon) = \lfloor \frac{c_q\sqrt{n}}{\epsilon^2} \rfloor$ for a constant c_q, which depends only on q. Note that $\max_q c_q = O(1)$. Then, we have

$$s_\ell = \sum_{k=1}^{\ell} \left\lfloor \frac{c_q\sqrt{n} \cdot \lceil 18\log(3k) \rceil}{\epsilon_k^2} \right\rfloor = \sum_{k=1}^{\ell} \left\lfloor c_q\sqrt{n} \cdot 4^k \cdot \lceil 18\log(3k) \rceil \right\rfloor = \Theta(\sqrt{n} \cdot 4^\ell \log \ell).$$

Here, the last equality holds since

$$4^\ell \log(3\ell) < \sum_{k=1}^{\ell} 4^k \log(3k) < \sum_{k=1}^{\ell} 4^k \log(3\ell) = \frac{4}{3}(4^\ell - 1)\log(3\ell) < \frac{4}{3} \cdot 4^\ell \log(3\ell).$$

By setting $a = \left\lfloor \log_2 \frac{1}{2\sqrt{\chi^2(p,q)}} \right\rfloor$ and $b = \left\lceil \log_2 \frac{1}{d_{\mathrm{tv}}(p,q)} \right\rceil$, we have

$$s_a = \Theta\left(\frac{\sqrt{n}}{\chi^2(p,q)} \log\log \frac{1}{\chi^2(p,q)} \right) \quad \text{and} \quad s_b = \Theta\left(\frac{\sqrt{n}}{d_{\mathrm{tv}}(p,q)^2} \log\log \frac{1}{d_{\mathrm{tv}}(p,q)} \right),$$

and hence, we obtain Theorem 2.

4.2 The case when p and q are both unknown

We next consider the case when p and q are both unknown. We build a similar framework for the case and then provide a stopping rule for Theorem 3.

Suppose that we have an algorithm with sample complexity $g(\epsilon)$ that distinguishes between the cases

(a) $d_1(p,q) \ge \epsilon$ and

(b) $d_2(p,q) \le \epsilon/2$,

with probability at least $2/3$. Then, by Lemma 10, we can obtain a stopping rule T^ϵ such that

- $\Pr\left[\tau(T^{q,\epsilon}, p) \le g(\epsilon) \cdot \lceil 18\log(3k) \rceil\right] \ge 1 - \frac{1}{9k^2}$ if $d_1(p,q) \ge \epsilon$, and
- $\Pr\left[\tau(T^{q,\epsilon}, p) \le g(\epsilon) \cdot \lceil 18\log(3k) \rceil\right] \le \frac{1}{9k^2}$ if $d_2(p,q) \le \epsilon/2$.

Our framework is almost the same as Algorithm 1. The stopping rule T induced by T^ϵ is shown as Algorithm 2.

Then we can prove the following lemma in the same way as the proof of Lemma 11.

▶ **Lemma 12.** *If $p \ne q$, the stopping time $\tau(T,p,q)$ for T in Algorithm 2 satisfies*

$$\Pr[s_a \le \tau(T,p,q) \le s_b] \ge 2/3$$

where $a = \left\lfloor \log_2 \frac{1}{2d_2(p,q)} \right\rfloor$, $b = \left\lceil \log_2 \frac{1}{d_1(p,q)} \right\rceil$, and $s_\ell = \sum_{k=1}^{\ell} f(q,\epsilon_k) \cdot \lceil 18\log(3k) \rceil$.

Algorithm 2: Stopping rule T induced by T^ϵ

input : $x_1 \cdots x_t \in [n]^*$ and $y_1 \cdots y_t \in [n]^*$ **output:** 0 or 1

Let $s_0 = 0$;
for $k = 1, 2, \ldots$ **do**
  Let $\epsilon_k = 1/2^k$ and $s_k = s_{k-1} + g(\epsilon_k) \cdot \lceil 18 \log(3k) \rceil$;
  if $s_k > t$ **then return** 0;
  else if $T^{\epsilon_k}(x_{s_{k-1}+1} \cdots x_{s_k}, y_{s_{k-1}+1} \cdots y_{s_k}) = 1$ **then return** 1;

When T^q is the stopping rule induced by the algorithm in Theorem 6, we have $d_1(p,q) = d_2(p,q) = d_{\mathrm{tv}}(p,q)$ and $g(\epsilon) = \lfloor \frac{cn}{\epsilon^2 \log n} \rfloor$ for a constant c. Then, we have

$$s_\ell = \sum_{k=1}^{\ell} \left\lfloor \frac{cn \cdot \lceil 18 \log(3b) \rceil}{\epsilon_k^2 \log n} \right\rfloor = \sum_{k=1}^{\ell} \left\lfloor \frac{cn \cdot 4^k \cdot \lceil 18 \log(3b) \rceil}{\log n} \right\rfloor = \Theta\left(\frac{n \cdot 4^\ell \log l}{\log n} \right).$$

By setting $a = \left\lfloor \log_2 \frac{1}{2 d_{\mathrm{tv}}(p,q)} \right\rfloor$ and $b = \left\lceil \log_2 \frac{1}{d_{\mathrm{tv}}(p,q)} \right\rceil$, we have

$$s_a = \Theta\left(\frac{n/\log n}{d_{\mathrm{tv}}(p,q)^2} \log\log \frac{1}{d_{\mathrm{tv}}(p,q)} \right) \quad \text{and} \quad s_b = \Theta\left(\frac{n/\log n}{d_{\mathrm{tv}}(p,q)^2} \log\log \frac{1}{d_{\mathrm{tv}}(p,q)} \right).$$

Hence, we obtain Theorem 3.

References

1 Jayadev Acharya, Clément L. Canonne, and Gautam Kamath. A chasm between identity and equivalence testing with conditional queries. In *Proceedings of APPROX/RANDOM*, 2015.
2 Jayadev Acharya, Constantinos Daskalakis, and Gautam C. Kamath. Optimal Testing for Properties of Distributions. In *Proceedings of NIPS*, pages 3591–3599, 2015.
3 Akshay Balsubramani. Sharp finite-time iterated-logarithm martingale concentration. page 25, 2014. `arXiv:1405.2639`.
4 Akshay Balsubramani. Sharp finite-time iterated-logarithm martingale concentration. *arXiv preprint arXiv:1405.2639*, 2014.
5 Akshay Balsubramani and Aaditya Ramdas. Sequential nonparametric testing with the law of the iterated logarithm. *arXiv preprint arXiv:1506.03486*, 2015.
6 Jay Bartroff, Tze Leung Lai, and Mei-Chiung Shih. *Sequential experimentation in clinical trials: design and analysis*, volume 298. Springer Science & Business Media, 2012.
7 Tuğkan Batu, Eldar Fischer, Lance Fortnow, Ravi Kumar, Ronitt Rubinfeld, and Patrick White. Testing random variables for independence and identity. In *Proceedings FOCS*, pages 442–451, 2001.
8 Tuğkan Batu, Lance Fortnow, Ronitt Rubinfeld, Warren D Smith, and Patrick White. Testing that distributions are close. In *Proceedings of FOCS*, pages 259–269, 2000.
9 Arnab Bhattacharyya, Eldar Fischer, Ronitt Rubinfeld, and Paul Valiant. Testing monotonicity of distributions over general partial orders. In *Proceedings of ICS*, pages 239–252, 2011.
10 Clément L. Canonne. A survey on distribution testing: Your data is big. but is it blue? *Electronic Colloquium on Computational Complexity (ECCC)*, 22:63, 2015.
11 Clément L. Canonne, Ilias Diakonikolas, Themis Gouleakis, and Ronitt Rubinfeld. Testing shape restrictions of discrete distributions. In *Proceedings of STACS*, 2016.

12 Clément L. Canonne, Dana Ron, and Rocco A. Servedio. Testing probability distributions using conditional samples. *SIAM Journal on Computing*, 44(3):540–616, 2015.

13 Siu-On Chan, Ilias Diakonikolas, Gregory Valiant, and Paul Valiant. Optimal algorithms for testing closeness of discrete distributions. In *Proceedings of SODA*, pages 1193–1203, 2014.

14 Roger H. Farrell. Asymptotic behavior of expected sample size in certain one sided tests. *The Annals of Mathematical Statistics*, pages 36–72, 1964.

15 Oded Goldreich and Dana Ron. On Testing Expansion in Bounded-Degree Graphs. *Electronic Colloquium on Computational Complexity (ECCC)*, 7(20), 2000.

16 Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

17 Kevin Jamieson, Matt Malloy, Robert Nowak, and Sebastien Bubeck. lil' UCB : An Optimal Exploration Algorithm for Multi-Armed Bandits. In *Proceedings of COLT*, 2014.

18 Ramesh Johari, Leo Pekelis, and David J Walsh. Always valid inference: Bringing sequential analysis to A/B testing. *arXiv preprint arXiv:1512.04922*, 2015.

19 Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *Proceedings of ICML*, 2013.

20 Eugene Kharitonov, Aleksandr Vorobev, Craig Macdonald, Pavel Serdyukov, and Iadh Ounis. Sequential testing for early stopping of online experiments. In *Proceedings of SIGIR*, pages 473–482, 2015.

21 Aleksandr Khintchine. über einen satz der wahrscheinlichkeitsrechnung. *Fundamenta Mathematicae*, 6(1):9–20, 1924.

22 Martin Kulldorff, Robert L Davis, Margarette Kolczak, Edwin Lewis, Tracy Lieu, and Richard Platt. A maximized sequential probability ratio test for drug and vaccine safety surveillance. *Sequential analysis*, 30(1):58–78, 2011.

23 Tze Leung Lai. *Sequential analysis*. Wiley Online Library, 2001.

24 K.K. Gordon Lan and David L. DeMets. Discrete sequential boundaries for clinical trials. *Biometrika*, 70(3):659–663, 1983.

25 Evan Miller. How Not To Run An A/B Test. 2010. URL: `http://www.evanmiller.org/how-not-to-run-an-ab-test.html`.

26 Peter C. O'Brien and Thomas R Fleming. A multiple testing procedure for clinical trials. *Biometrics*, pages 549–556, 1979.

27 Liam Paninski. A coincidence-based test for uniformity given very sparsely sampled discrete data. *IEEE Transactions on Information Theory*, 54(10):4750–4755, 2008.

28 Stuart J. Pocock. Group sequential methods in the design and analysis of clinical trials. *Biometrika*, pages 191–199, 1977.

29 Ronitt Rubinfeld. Taming big probability distributions. *XRDS*, 19(1):24–28, 2012.

30 David Siegmund. Estimation following sequential tests. *Biometrika*, 65(2):341–349, 1978.

31 David Siegmund. *Sequential analysis: tests and confidence intervals*. Springer Science & Business Media, 2013.

32 Gregory Valiant and Paul Valiant. The power of linear estimators. In *Proceedings of FOCS*, pages 403–412, 2011.

33 Gregory Valiant and Paul Valiant. An automatic inequality prover and instance optimal identity testing. In *Proceedings of FOCS*, pages 51–60, 2014.

34 Abraham Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 1945.

35 Abraham Wald and Jacob Wolfowitz. Optimum character of the sequential probability ratio test. *The Annals of Mathematical Statistics*, pages 326–339, 1948.

36 Samuel K. Wang and Anastasios A. Tsiatis. Approximately optimal one-parameter boundaries for group sequential trials. *Biometrics*, pages 193–199, 1987.

The Online House Numbering Problem: Min-Max Online List Labeling*

William E. Devanny[1], Jeremy T. Fineman[2], Michael T. Goodrich[3], and Tsvi Kopelowitz[4]

1 **University of California, Irvine, CA, USA**
wdevanny@uci.edu
2 **Georgetown University, Washington, D.C., USA**
jfineman@cs.georgetown.edu
3 **University of California, Irvine, CA, USA**
goodrich@uci.edu
4 **University of Waterloo, Ontario, Canada**
kopelot@gmail.com

Abstract

We introduce and study the *online house numbering* problem, where houses are added arbitrarily along a road and must be assigned labels to maintain their ordering along the road. The online house numbering problem is related to classic online list labeling problems, except that the optimization goal here is to minimize the maximum number of times that any house is relabeled. We provide several algorithms that achieve interesting tradeoffs between upper bounds on the number of maximum relabels per element and the number of bits used by labels.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases house numbering, list labeling, file maintenance

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.33

1 Introduction

In this paper we study a new version of the fundamental *online monotonic list labeling problem* [14, 6, 23] (OMLL), where the goal is to maintain labels for a dynamic ordered list of at most n elements that, due to monotonicity requirements of appropriate applications, must have (integer) labels that strictly increase in the direction of the ordering. When a new element is inserted into the list, either between two existing elements or at an endpoint of the list, we must assign a label to the new element that is consistent with the order of the list. To avoid labels becoming too long, algorithms for list-labeling problems relabel elements from time to time thereby maintaining the ordering using relatively few bits for the labels. There are several common variants of OMLL that differ in the number of bits allowed for each label. For example, the special case of $\log n + O(1)$ bits[1] is known as the file-maintenance problem [27, 26, 6, 7], where labels are viewed as corresponding to addresses in a size $O(n)$ array.

* This research was supported in part by NSF grants CCF-1617727, 1228639, CCF-1526631, CCF-1514383 and CCF-16375, and by the Canada Research Chair for Algorithm Design. This article is also supported in part by DARPA under agreement no. AFRL FA8750-15-2-0092. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

[1] Unless another base is indicated, all logarithms in this paper are base 2.

Solutions for OMLL are used as foundational building blocks in several areas of computer science, ranging from cache-oblivious data structures [4, 5, 10] to distributed computing [17], and they play a central role in deamortization [14, 9, 16]. As an illustration of the data structure's central role in the field, it is used as a black box in order-maintenance data structures [14, 23], which themselves are used as black boxes throughout computer science (for examples see [2, 15, 13, 20, 1, 18, 24]).

The focus of previous work solving the OMLL problem in the RAM model has been on minimizing the worst-case or amortized number of relabels per update. For example, when using $O(\log n)$ bits per label the worst-case number of relabels per update is known to be $O(\log n)$ [23], which is tight [11]. A particular element, however, can be relabeled as many as $\Omega(n)$ times during a sequence of n insertions using existing algorithms. This paper considers the goal of minimizing the maximum number of times an element in the list is relabeled, while using only a small number of bits per label. We refer to this version of OMLL as the *online house numbering problem*, since it captures the challenges that take place when maintaining a strictly increasing numbering for a collection of houses representing their order along a road. When a new house is built between any two existing houses (or at either end of the row of houses), this new house needs to be assigned a house number. If no such integer house number is available, however, then other houses need to be renumbered (or relabeled) to make room for a number for the new house. Formally stated, the online house numbering problem is to maintain a labelling of an initially empty ordered list subject to n operations of the form, $\mathsf{insert}(x, a)$: insert x immediately after a in the ordered list. Remarkably, existing solutions for list labeling problems do not seem to lead to efficient solutions for the online house numbering problem.

The online house numbering problem raises some interesting combinatorial questions while also addressing label-update complexity, which is motivated from use of solid-state memories, like flash memory, that have an upper bound on the number of erasures that can occur for any memory cell [8, 25, 28]. For example, consider a database with an ordered set of large records, where each record maintains a label respecting the order. Due to the use of these modern types of memory, the number of times that the label is changed must be minimized, since each relabeling entails rewriting that area in memory. A typical assumption in models for solid-state memories is that the algorithm or data structures also have access to a sublinear amount of additional *scratch space* for computational purposes (see Ben-Aroya and Toledo [3]), which is exempt from the erasure limits. In the context of our online house numbering, this would mean that each element in the data structure has a fixed record containing, e.g., the label and any other auxiliary information that is updated whenever a label changes (for our solution, we also store a counter as part of the record). Any additional components of the data structure must be restricted to the $o(n)$ scratch space.

There are two competing objectives that we consider in designing solutions for the online house numbering problem. The first objective is to minimize, over all elements in the list, the maximum number of times that the label of the element changes throughout the n insertions. Notice that with large labels, a trivial solution in which no relabels are needed is obtainable by assigning x the average of a and b, where b is the element succeeding x. This trivial solution requires $\Omega(n)$ bits per label, and so if each word of memory contains $\Theta(\log n)$ bits (which is a standard assumption), each label requires $\Omega(n/\log n)$ words. A large number of words directly impacts the efficiency of establishing the order of two elements, since comparing their labels entails scanning that many words. Thus, the second objective is to minimize the number of bits used in labels.

Since we are interested in minimizing two competing objectives, we express the complexities of our data structures using a pair of functions. A data structure supporting n insertions

with $g(n)$ maximum relabels and using $h(n)$ bits per label is said to have complexity of $\langle g(n), h(n)\rangle$. Notice that $h(n) \geq \lceil \log n \rceil$ since n elements must be labeled. If one is interested in $h(n) = O(\log n)$ (constant number of words per label), then the OMLL lower bounds of [11] imply that $g(n) = \Omega(\log n)$. Thus, if there existed a solution for online house numbering with complexity $\langle O(\log n), O(\log n)\rangle$, it would be asymptotically optimal.

1.1 Our Results

In this paper we describe two data structures that are close to the target bound of $\langle O(\log n), O(\log n)\rangle$, but each solution introduces an extra logarithmic factor in one of the functions. In a third solution, we investigate the dependence on the leading constant of $h(n)$ and provide a solution with complexity $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$. Our solutions, which can be adapted to work with $o(n)$ scratch space (deferred to the full version of the paper), establish the following results.

▶ **Theorem 1.** *There exists a house numbering data structure with complexity* $\langle O(\log^2 n), O(\log n)\rangle$.

▶ **Theorem 2.** *There exists a house numbering data structure with complexity* $\langle O(\log n), O(\log^2 n)\rangle$.

▶ **Theorem 3.** *For any positive constant* ϵ*, there exists a house numbering data structure with complexity* $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$.

Proofs of Theorems 1 and 3 appear in Section 3. Theorem 2 is deferred to the full version of the paper. Our solution complexities exhibit an interesting feature: the online house numbering problem seems to exhibit a different tradeoff from OMLL, depending more strongly on the label lengths.

Overview of Challenges and Techniques. The main idea of our approach is that once a particular element has been relabeled many times, structural restrictions assure that this element will not be relabeled much in the future. To achieve this goal, we employ a tree-like structure similar to an (a, b)-tree (or B-tree) that stores at most $O(\gamma)$ elements in nodes of the tree, where $\gamma \geq 2$ is a parameter controlling the tradeoff between the two objectives. (For the purpose of this overview it is helpful to assume $\gamma = 2$.) Roughly speaking, the inorder traversal of this tree corresponds to the order of elements in the list. The elements in a node each have a local label, which is local to that node. The global label assigned to an element corresponds to the concatenation of the local labels on the path from the root to the element (with 0s padded at the end if the element is in an inner node). We require the local labels of elements to respect the order of elements in each node, thereby guaranteing that the global labels respect the total order of the list. To simplify things when extending to nonconstant γ, we employ (classic) file-maintenance data structures within each node for maintaining the local labels. Notice that changing the local label of an element also changes the global labels, which must be stored explicitly, of all elements in that element's subtree.

Our main strategy is to employ node splits to "promote" elements that have been relabeled too many times to higher levels in the tree. Promoting an element e that is currently in node u entails: splitting the elements of u around e into two new nodes, moving e into u's parent, and making the two new nodes children of u's parent. The intuition behind the promotions is that elements in higher nodes are less affected by insertions, and hence these elements need not be relabeled as often. Element promotions happen due to three possible reasons:

(1) if the element has been relabeled too many times since its last promotion,
(2) if the node has too many elements, which would cause the performance of the file-maintenance black box to degrade, or
(3) if the node becomes sufficiently imbalanced, according to a non-obvious weight-balance rule.

The key component of the analysis is ensuring that the height H of the data structure is well bounded, i.e., that elements are never promoted too far. The height H not only places an upper bound on the length of the labels, but in conjunction with the first trigger for element promotion also directly implies a bound on the number of times each element can be relabeled. We emphasize that the analysis bounding H leverages a potential argument in an atypical and non-obvious way, which we view as a surprising component of our data structure.

1.2 Related Prior Work

There is no prior work for the online house numbering problem, but it is closely related to the classic *file maintenance* and *online list labeling* problems for which several authors have shown how to achieve optimal polylogarithmic update times, in either worst-case or amortized senses (e.g., see [6, 14, 22, 23, 11, 7]).

Regarding algorithms in computational models that capture the challenges of solid-state memory, Ben-Aroya and Toledo [3] provide competitive analyses for several such algorithms, but they do not study OMLL problems as a specific topic of interest. See also the work of Irani *et al.* [21]. Subsequent work on efficiently implementing specific data structures and algorithms in such models includes methods for database algorithms [12] and hash tables [19].

2 Preliminaries

In our house numbering data structures, we make use of instances of file maintenance data structures. The following lemma highlights the features that our algorithms leverage. Here, a file-maintenance data structure corresponds to an array, where placing an element in the ith slot in the array corresponds to assigning a label of i to the element.

▶ **Lemma 4.** *For any capacity η, there exists a file maintenance data structure with the following properties:*

- *The data structure assigns to each element a slot in the range $[1, 4\eta]$. Slots are such that a is before b if and only if in the total order a's slot is before b's slot.*
- *If the data structure has at most η elements then it can be split into two data structures with each element being moved at most once.*
- *Starting from an empty data structure, or a data structure that is the output of a split, as long as the number of elements in the structure does not exceed η, the amortized number of elements that are moved to a new slot per insertion is $O(\log^2 \eta)$.*

Proof. A data structure by Itai et al. satisfies these conditions [22]. More detail is given in the full version of the paper. ◀

Using the notation of the statement of Lemma 4, a file maintenance data structure f is characterized by a **capacity** (i.e., η), a **slot range** (i.e., $[1, 4\eta]$), and an amortized **moving cost** $cost(\eta)$ (i.e., $O(\log^2 \eta)$), which are all static. The capacity specifies how many elements can be inserted while still maintaining the $cost(\eta)$ bound. In addition, we define the **usage** of f, denoted $usage(f)$, to be the number of elements currently inside f.

3 A Generic House Numbering Data Structure

We describe our data structure in terms of several variables, namely κ_1, κ_2, κ_3, π_1, and π_2. These will allow us to balance various overheads in the data structure and shall be fixed in the analysis. Additionally, our house numbering data structure is parameterized by a value $\gamma \geq 2$, which controls a tradeoff between the label lengths and the number of relabels performed. Setting γ to a constant attains the $\langle O(\log^2 n), O(\log n)\rangle$ data structure. When considering a data structure for flash memory, the data structure itself, in addition to the actual list, should reside in scratch space. We ignore this issue in the current section but address it in the full version of the paper.

The tree. Our house-numbering structure is a perfect rooted $(4\kappa_1\gamma + 1)$-ary tree T. Each internal node u in the tree maintains an instance f_u of a $\kappa_1\gamma$-capacity file-maintenance data structure (à la Lemma 4) and the leaves of the tree are associated with space for a single element. The leaves store the actual elements e in the tree, but leaves may be empty. Each element e also maintains a **relabel counter** $c(e)$.

The internal nodes store (conceptual) copies of elements, which we call **representatives**, that have been promoted to a higher level in the tree. We refer to all the copies of a particular element e as **the representatives** e. Representatives are analogous to duplicate keys in internal nodes of a B^+ tree, with each non-empty node containing exactly one representative that has been promoted to the parent node.

Each file-maintenance data structure f_u assigns slots in the range $[1, 4\kappa_1\gamma]$ to the representatives in node u. Equivalently, the file-maintenance structure specifies how to store the representatives in a size-$\kappa_1\gamma$ array, starting from slot 1. We use the 0th slot in the array for a special **dummy representative** d_u^-, which corresponds to the only representative in node u that has also been promoted to the parent. (As such, d_u^- is a representative of the leftmost left element in u's subtree.) Note that since the slot storing the dummy representative is not part of the file-maintenance structure f_u, the dummy representative never moves from slot 0. The i-th slot in f_u corresponds to the i-th child of u in an inorder tree walk. For representative r in f_u let $s(r)$ denote the slot in f_u that is assigned to r.

Without yet worrying about precisely how elements are labelled, we state a property about how they must appear in T. Naturally, the order property constrains the way we label elements.

▶ **1 (Order Property).** *An inorder traversal of T encounters the elements in their house numbering order.*

We use T_u to denote the subtree rooted at node u. Let v be the parent of u in T. Since u is represented as a slot s in the slot range of f_v, we will abuse notation and sometimes denote T_u by T_s. A subtree is empty if it contains no elements.

The labels. The label for an element e, denoted $\ell_T(e)$, is based on the root-to-leaf path down to the leaf node containing e. In particular, labels are base-$(4\kappa_1\gamma + 1)$ numbers with a number of digits equal to the height of the tree. Consider the path $u_1, u_2, \ldots, u_{H_T+1}$ down to the leaf containing e, where u_1 is the root, u_{H_T+1} is the leaf containing e, and H_T is the height of T. For $1 \leq i \leq H_T$ let s_i denote the slot of u_{i+1} in f_{u_i}. Then e's label is the concatenation of digits $s_1, s_2, \ldots, s_{H_T}$. An example of determining the labels of elements is depicted in Figure 1. The label of each element uses $\lceil \log(4\kappa_1\gamma + 1)\rceil$ bits per level of T for a total of $H_T\lceil \log(4\kappa_1\gamma + 1)\rceil$ bits.

Notice that by construction and the Order Property, the labels of elements respect the order of the elements in the house numbering.

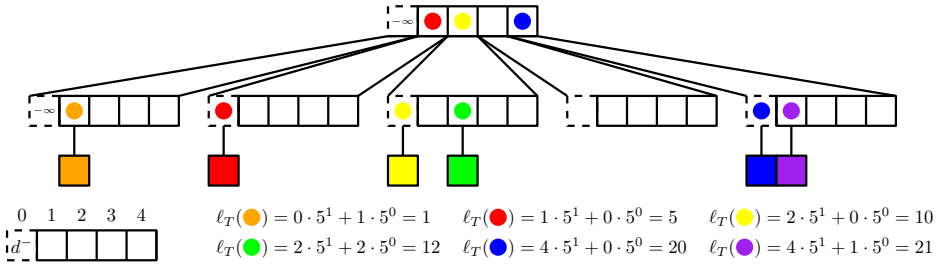


Figure 1 This tree illustrates how elements are labelled based on their representative's node's file maintenance labels and the node labels of their parents. Empty leaf nodes are omitted and we note that the root node is violating the Capacity Property.

Relabeling and subtrees. To maintain the Order Property, whenever a representative r is moved from slot s to slot s' in f_u, all of the elements and file-maintenance representatives in T_s are moved to the same exact location, but in $T_{s'}$. The following property will guarantee that this movement does not violate the Order Property.

▶ **2 (Representative Property).** *The representatives for an element e induce a path from the parent of the leaf containing e to the highest representative. Each representative of e except for the highest one is the dummy representative of its corresponding node.*

Following the Representative Property, we abuse notation and refer to the highest representative of an element e as the canonical representative of e, and denote this representative by $r(e)$.

3.1 Insertions

We now discuss the implementation of the insert(x, a) operation. Let u be the parent of the leaf node containing a, and assume for now that $usage(f_u) < \kappa_1\gamma$. A new representative r of x is inserted into f_u immediately after the representative representing a in f_u (possibly causing elements in f_u to change slots). Because this insertion is into a file maintenance data structure, this insertion may cause some movement of other elements. Element x is placed into the leaf node corresponding to the slot assigned to r in f_u.

The insertion respects the Order Property, so x receives a valid label. The insertion causes some number of other representatives in f_u to be relabelled and also increases the usage of f_u. Eventually f_u will reach capacity. The capacity of the file maintenance instances needs to be respected and so when f_u reaches capacity we move around representatives in T to create room (thereby guaranteeing again that f_u is below capacity before the next insertion). This is captured by the following property.

▶ **3 (Capacity Property).** *For any internal node u in T, $usage(f_u) < \kappa_1\gamma$.*

In order to maintain the Capacity Property, the data structure employs *promoting* canonical representatives to higher nodes in T. The promotion procedure is detailed in Section 3.2.

Relabel counters. The relabel counter of an element is incremented whenever the label of the element is changed. To prevent any one element from being relabelled too many times, we enforce a bound on the relabel counter. Recall that the $cost(\eta)$ function is defined in Section 2.

Algorithm 1 Insert x into the house numbering data structure immediately after element a.

```
1: function INSERT(x, a)
2:     u ← parent of a's leaf
3:     insert a representative of x into f_u
4:     move any relabeled elements to their new leaves
5:     place x into the corresponding leaf of u
6:     repeatedly fix property violations using promote()
7: end function
```

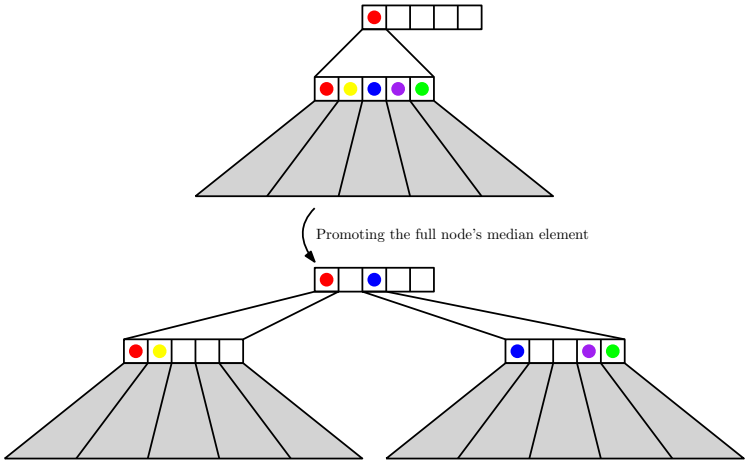


Figure 2 The blue median representative of the full file maintenance data structure is promoted dividing the subtree into two parts.

▶ **4. Counter Property:** *For any element* e, $c(e) < \kappa_2\, cost(\kappa_1\gamma)$.

In order to enforce the Counter Property, whenever the counter of element e reaches its threshold, $r(e)$ is moved one level higher in the tree by a *promotion operation*, which we describe shortly, and sets $c(e) = 0$.

Notice that moving a representative to a new slot higher up in T will tend to relabel more elements compared to moving a representative to a new slot in a lower node in T. This presents a subtle challenge. Consider two representatives in f_u where u is relatively high up in T, such that one representative r has every file maintenance instance in $T_{s(r)}$ half full while the other representative r' has every file maintenance instance in $T_{s(r')}$ just below capacity. This implies that the number of elements contained in the two subtrees differ exponentially in the height of the subtrees. The consequence of this imbalance is that insertions of elements into the lighter subtree $T_{s(r)}$ can cause frequent promotions into f_u, each time causing r' to move to a new slot in f_u. When r' moves to a new slot, all of the elements in $T_{s(r')}$ must be relabeled. This imbalance creates some difficulties in keeping a tab on the complexities of the data structure. To overcome these difficulties, we enforce a requirement on the data structure to have the following property, which helps ensure a promotion does not relabel too many elements that are too high in the tree by restricting the weight of any subtree. For a representative r in f_u where u has height h in T, let $w(r) = \gamma^h$.

▶ **5. Balance Property:** *For any node* u, $\sum_{node\ v \in T_u} \sum_{canonical\ representative\ r \in f_v} w(r) < \kappa_3 \gamma^{height(u)}$.

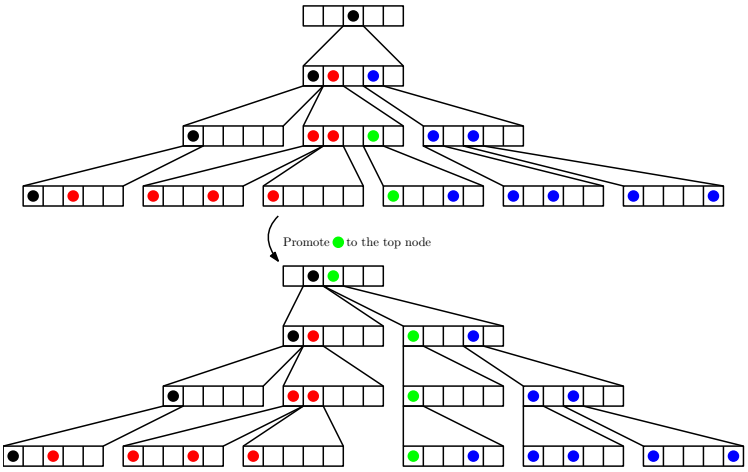


Figure 3 The black representative's subtree passed the threshold of the Balance Property and the weighted median representative, shown in green, is promoted. The subtrees of representatives less and greater than the median are copied and possibly shifted if they need a new root representative. Empty subtrees are omitted.

Algorithm 2 Promote an element x into a node u

```
1:  function PROMOTE(x, u)
2:      v ← node containing the canonical representative for x
3:      remove x from f_v
4:      a ← predecessor of x in f_u
5:      insert a new canonical representative of x after a in f_u
6:      move the entire subtree of any relabeled elements
7:      split the subtree below a's canonical representative into:
8:            - T_1 a subtree of elements < x and >= a
9:            - T_2 a subtree of elements >= x
10:     place T_1 below a's canonical representative
11:     place T_2 below x's canonical representative
12: end function
```

3.2 Promotions

For element e, the *promotion* of $r = r(e)$ from f_u to f_v, where v is a proper ancestor of u, is performed as follows. Let $\hat{r}$ be the representative in the slot in f_v that contains e in its subtree:

1. Insert r', which is a new representative of e, into f_v immediately after $\hat{r}$ in the order f_v is maintaining (this may cause some elements in f_u to change their slots).
2. Any element in $T_{s(\hat{r})}$ that is after e (inclusive) and its representatives in $T_{s(\hat{r})}$ are moved into identical locations in the subtree of $T_{s(r')}$.
3. The previous step partitions some file maintenance instances into two pieces. Each such instance respaces the representatives it contains according to the split operation of Lemma 4. Notice that if the dummy representative is part of one piece, the data structure adds a new dummy representative in the other piece. This new dummy representative is a representative of e.

Examples of promotions are shown in Figures 2 and 3.

▶ **Lemma 5.** *Promotions preserve the Order and Representative Properties.*

Proof. Before the promotion, the Order Property held and so an inorder traversal encountered the elements with label less than $\ell_T(e)$, then e, and then the elements with label greater than $\ell_T(e)$. After the promotion, the inorder traversal will traverse $T_{s(\hat{r})}$ which contains the elements less than e and then $T_{s(r')}$ which contains e followed by the elements greater than e. The two new subtrees preserved the original inorder traversal of their contained elements. So the inorder traversal before and after the promotion traverses the elements in T in the exact same order and the Order Property holds. The last step of a promotion ensures that the representatives of e still behave properly with respect to the Representative Property. Since we split along the path of elements less than and greater than e, no other path of representatives was altered and the Representative Property still holds. ◀

The only operations we perform on T are insertions of elements at leaves and promotions. Both of these preserve the Order and Representative Properties. Violations of the Capacity, Balance, and Counter Properties are fixed by promoting certain representatives. When a node u with parent v violates the Capacity Property, promote the median representative in f_u (which must be a canonical representative) into f_v. When the subtree of $s(r)$ becomes too heavy and violates the Balance Property, promote the weighted median canonical representative in the subtree of $s(r)$ into the node that contains r. When $c(e)$ passes the threshold of the Counter Property, promote the canonical representative of e into the parent of its current node, and reset $c(e)$ to be zero. Figure 3 shows a promotion due to a violation of the Balance Property and Figure 2 shows a promotion due to a violation of the Capacity Property.

Promoting a representative may introduce new property violations. For example, suppose f_u for some internal node u contains $\kappa_1\gamma$ representatives and its parent v has exactly $\kappa_1\gamma - 1$ representatives. Promoting the median representative from f_u to f_v will cause the f_v to violate the Capacity Property.

The very rough pseudocode in Algorithms 1 and 2 describes the high level steps for insertions and promotions. We describe exactly how violations are processed next.

Property violations. Since several properties may be violated at the same time, we employ the following prioritization for fixing these violations. We process the property violations by alternating between processing all violations of the Capacity and Balance Properties in a highest first fashion and then processing a single violation of the Counter Property. Algorithm 3 shows this procedure in pseudocode.

It is not yet clear that this processing terminates. We address this in Section 4. Whenever the initial insertion or a promotion causes an element to be relabeled, we increment the corresponding relabel counter.

During the processing of violations, a given relabel counter may be increased well past the bound in the Counter Property. But our potential argument only allows us to charge for relabelings that occur when the counter is at or below the threshold. To keep from relabeling the corresponding element each time an above-threshold counter is pushed even higher during a single (recursive) house numbering insertion, we perform the invariant violation processing on a logical copy of the data structure and only relabel elements with the final label. While a relabel counter may be incremented many times, any element is only relabelled at most once per insert operation.

Algorithm 3 Process the violations in the tree

```
1:  function PROCESS_VIOLATIONS
2:      while there is a violated property do
3:          while there is a violation of the capacity or balance properties do
4:              process the highest capacity or balance violation
5:              (capacity violations have priority)
6:          end while
7:          if there is a violation of the counter property then
8:              process one violation of the counter property
9:          end if
10:     end while
11: end function
```

4 Bounding the Height and Complexities

Let $H(n)$ denote an upper bound on the maximum possible height of a canonical representative in our house-numbering data structure after n elements are inserted. (We shall bound $H(n)$ as a function of γ in Lemma 9.) Then we can directly bound the length of labels at $H(n) \cdot \lceil \log(4\kappa_1\gamma + 1) \rceil$ bits. In particular, if κ_1 and γ are constants, we will prove that $H(n) = O(\log n)$ and hence the labels use $O(\log n)$ bits. Moreover, since we guarantee the Counter Property, each element e can be relabeled at most $\kappa_2\, cost(\kappa_1\gamma)$ times before $r(e)$ is moved up a level. So the maximum number of times that an element is relabeled is $O(\kappa_2 H(n)) = O(\kappa_2 \log n)$, assuming $\kappa_1\gamma$ is a constant and $H(n) = O(\log n)$.

The intuition behind our height analysis is as follows. Each insertion causes $cost(\kappa_1\gamma)$ representatives to be relabeled. Thus we need roughly κ_2 insertions to trigger enough relabels that a single representative could be promoted by the Counter Property. In other words, at most a $1/\kappa_2$ fraction of representatives are promoted due to insertions of elements and the Counter Property. This argument extends up the tree; promotions into height-h nodes can cause at most a $1/\kappa_2$ fraction of representatives to be promoted from height h. If this were the only effect, we would see $(1/\kappa_2)^h$ representatives promoted to height h.

This challenge turns out to be even more complex, since each promotion into a height-h node u also causes the elements in subtrees of any locally relabeled representatives in u to be completely relabeled. The Balance Property helps us to bound the total weight of representatives in these subtrees by $\kappa_3\gamma^h$. By increasing κ_2 enough, we effectively amortize the high number of relabelings due to moving a subtree against the geometrically decreasing number of promotions to that height, i.e., about $\kappa_3\gamma^h/\kappa_2^h$ per insertion. Since there are some "feedback" effects that arise from the interaction of fixing property violations, the analysis must proceed with care.

Before we turn to bounding the height, we prove a useful lemma.

▶ **Lemma 6.** *If all of the properties hold before an insertion, the processing of the resulting violations will never promote a representative into a node that:*

- *violates the Capacity Property or*
- *contains a representative whose subtree violates the Balance Property.*

Proof. Call a promotion into such a node an *invalid promotion*. We claim that in addition to never performing an invalid promotion, the violation processing maintains the property that violations of the Capacity and Balance Properties each occur at most once in each level of T. We call this the *Once Per Level Property*. When a representative is inserted or promoted

into u: the usage of u is increased, the weight of every subtree of every representative on the path to the root is increased, and the relabel counters of any relabeled elements are increased. So an insertion or promotion will only introduce violations of the Capacity Property at u and violations of the Balance Property along the path from u to the root (and some other violations of the Counter Property). For example in Figures 2 and 3, each promotion can only create Counter Property violations lower in the subtree while it may introduce Capacity and Balance Property violations at the root. Hence the initial insertion or promotion from processing a Counter Property violation in a tree with no violations of the Capacity or Balance Properties is valid and will maintain the Once Per Level Property.

When the Once Per Level Property holds, there is some highest violation of each type. There is a highest node violating the Capacity Property and a highest node containing a representative violating the Balance Property. Let u be the higher of these two nodes. If both nodes have equal height, then let u be the highest node violating the Capacity Property. We consider the cases when u violates the Capacity Property or when u does not violate the Capacity Property and violates the Balance Property.

In the first case, f_u violates the Capacity Property by containing $\kappa_1\gamma$ representatives. Because the parent of u is not violating either of the two properties, promoting the median of f_u is valid. That promotion may introduce violations at the parent of u or higher, but they will only be in levels of the tree where there were previously no violations. The violations that were either at u or below will be unaffected by the promotion (except for the one being processed). Therefore the Once Per Level Property still holds after the violation is processed.

In the second case, f_u does not violate the Capacity Property but it does have one representative violating the Balance Property. Because no other representative in f_u violates the Balance Property due to the Once Per Level Property, processing the violation is valid. By promoting the weighted median descendant into u, only f_u can be newly in violation of the Capacity Property and only representatives in ancestors of u can be newly in violation of the Balance Property. Both of these types of new violations are introduced at levels that did not contain a violation of that type before. The splitting of the subtree below into two pieces can only eliminate violations in the levels below u. So after validly processing this violation, the Once Per Level Property holds.

In either case, processing a violation does not make an invalid promotion and maintains the Once Per Level Property. Thus, the invariant processing never promotes a representative into a node violating the Capacity Property or containing a representative whose subtree violates the Balance Property. ◀

Potential argument. To formalize the intuition outlined in the beginning of this section, we analyze our data structure using the following three potential functions, each of which corresponds to one of our properties:

- $\Phi_{fmds} = \pi_1 \sum_{internal\ nodes\ u} \max\left(2\gamma^{height(u)} \cdot usage(f_u) - \kappa_1\gamma^{height(u)+1}, 0\right)$
- $\Phi_{counters} = \sum_e w(r(e))c(e)$
- $\Phi_{tree} = \pi_2 \sum_u \max\left(2\sum_{node\ v\in T(u)} \sum_{canonical\ representative\ r\in f_v} w(r) - \kappa_3\gamma^{height(u)}, 0\right)$

The total potential, $\Phi(T)$, is the sum of these three potential functions, that is $\Phi = \Phi_{fmds} + \Phi_{counters} + \Phi_{tree}$. Each potential function corresponds to one of our three properties and guarantees that when a property is violated we have sufficient potential to "pay" for the promotion.

The next few lemmas show how the potential functions work with the properties. The change in Φ due to a processing a violation can be separated into the two phases of a

promotion. First, there is the decrease in potential when the promoted element's relabel counter is reset and the node containing the canonical representative of the element is split. Second, there is the increase in potential due to the insertion of the canonical representative of the element into a higher up node which results in relabeling many other elements, increasing that node's usage, and increasing the weight of every subtree containing the higher up node. Lemma 7 gives an upper bound on the increase in potential due to either an insertion or the second part of a promotion. Lemma 8 gives a lower bound on the decrease in potential due to the first part of a promotion. In conjunction these two Lemmas show that as long as the maximum height $H(n)$ is small, the lower bound on the decrease in potential is greater than the upper bound on the increase in potential. So a promotion results in a net decrease in potential and only insertions of new elements at the leaves increase the potential. Finally Lemma 9 contrasts the amount of potential gained from these insertions with the amount of potential needed to promote one representative to a height of $\log_\gamma n$. Because the former is strictly smaller, the height of the tree must be $H(n) < \log_\gamma n$.

▶ **Lemma 7.** *During a promotion, the insertion of a representative into a file maintenance data structure at height h increases $\Phi(T)$ by at most $(2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2\, height(T))\gamma^h$. Moreover, the insertion of an element increases $\Phi(T)$ by at most $2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2\, height(T)$.*

Proof. Placing a canonical representative into a node u at height h causes the potential functions to change as follows:

- $\Delta(\Phi_{fmds}) \leq 2\pi_1\gamma^h$, because f_u had its size increased by 1
- $\Delta(\Phi_{counters}) \leq \kappa_3\gamma^h\, cost(\kappa_1\gamma)$, because $cost(\kappa_1\gamma)$ representatives in f_u are relabeled, each causing subtrees with total weight at most $\kappa_3\gamma^h$ to be relabeled.
- $\Delta(\Phi_{tree}) \leq 2\pi_2\, height(T)\gamma^h$, because each representative on the path to the root has the potential of its subtree increased by at most γ^h

The bound on $\Delta(\Phi_{counters})$ is in an amortized sense and is due to Lemma 6. This is because by Lemma 6 we never promote a representative into a node that is violating the Capacity Property, so there are at most $cost(\kappa_1\gamma)$ relabels, or into a node violating the Balance Property, so incrementing the relabel counters of each subtree costs at most $\kappa_3\gamma^h$ potential.

In total these sum up to $(2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2\, height(T))\gamma^h$ which upper bounds the increase in all three potential functions. ◀

▶ **Lemma 8.** *If $height(T) \leq H(n)$, there exist settings of π_i's and κ_i's such that promoting a representative to fix a violation does not increase the total potential and $\kappa_2 = O(\gamma H(n))$.*

Proof. Depending on which violation caused the promotion, we must analyze the decrease in potential differently to account for the potential increase from Lemma 7 of $(2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2 H(n))\,\gamma^h$ where h is again the height of the node the promoted element is moved into.

- If a Capacity Property violation was processed, then Φ_{fmds} decreased by at least $\pi_1\kappa_1\gamma^h$.
- If a Counter Property violation was processed, then $\Phi_{counters}$ decreased by at least $\gamma^{h-1}\kappa_2\, cost(\kappa_1\gamma)$ due to the potential from $c(e)$.
- If a Balance Property violation was processed, then Φ_{tree} decreased by more than $\pi_2\kappa_3\gamma^h$ because of the subtree containing r.

To ensure the potential available is always at least the potential cost $\pi_1\kappa_1$, $\frac{\kappa_2\, cost(\kappa_1\gamma)}{\gamma}$, and $\pi_2\kappa_3$ must all be greater than $2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2 H(n)$. Analyzing the system of inequalities leads to setting $\kappa_1 = 3$, $\kappa_2 = 72\gamma H(n)$, $\kappa_3 = 12H(n)$, $\pi_1 = 24\, cost(3\gamma)H(n)$, and $\pi_2 = 6\, cost(3\gamma)$.

Plugging these values back into the original formula, the potential increases by at most

$$(2(24\, cost(3\gamma)H(n)) + (12H(n))\, cost(3\gamma) + 2(6\, cost(3\gamma))H(n))\gamma^h = 72\, cost(3\gamma)H(n)\gamma^h.$$

On the other hand, the potential decrease is at least

- $(24\, cost(3\gamma)H(n))(3)\gamma^h$ in the case of a Capacity Property violation,
- $\gamma^{h-1}(72\gamma H(n))\, cost(3\gamma)$ in the case of a Counter Property violation, or
- $(6\, cost(3\gamma))(12H(n))\gamma^h$ in the case of a Balance Property violation.

In all three cases, the lower bound of the decrease in potential is equal to $72\, cost(3\gamma)\gamma^h H(n)$ and therefore it is at least the increase in potential due to a promotion. ◀

▶ **Lemma 9.** *For the same setting of π_i's and κ_i's as Lemma 8, and $\gamma \leq n$, after n insertions there are no promotions to height above $\log_\gamma n$.*

Proof. Initially the tree is empty and has height zero. By Lemma 8, setting $\log_\gamma n = H(n)$, until $height(T)$ exceeds $H(n)$ promoting a representative does not increase the potential. Thus, while the height bound holds the only mechanism for increasing the potential is by inserting a new element. By Lemma 7, the increase in potential from inserting an element is at most $72\, cost(3\gamma)\log_\gamma n$ and so after n insertions, the potential of the entire data structure is at most $72\, cost(3\gamma)n\log n$.

To complete the proof, we observe that a representative can only be promoted to height h if the total potential in the data structure is at least $72\, cost(3\gamma)\gamma^h \log_\gamma n$. Specifically, the proof of Lemma 8 shows that the potential has to decrease by at least this amount when performing the promotion, so the potential has to exist before the promotion. In order to reach a height of at least $\log_\gamma n + 1$, we would need at least $72\, cost(3\gamma)\gamma^{\log_\gamma n+1}\log_\gamma n > 72\, cost(3\gamma)n\log n$ potential. Thus, a height $\log_\gamma n$ structure cannot have enough potential. ◀

▶ **Theorem 10.** *There exists a house numbering data structure with complexity $\langle O(\gamma\log^2 n), \log_\gamma n \cdot \lceil\log(12\gamma+1)\rceil\rangle$.*

Proof. By Lemma 9, after n insertions there are no promotions into nodes at height higher than $\log_\gamma n$, so a tree of this height suffices. Thus, each label uses $\log_\gamma n$ "digits", where each digit uses $\lceil\log(12\gamma+1)\rceil$ bits, for a total of $\log_\gamma n \cdot \lceil\log(12\gamma+1)\rceil$ bits per label. For the relabel bound, by the Counter Property, each canonical representative is promoted at most $\kappa_2\, cost(3\gamma) = O(\gamma H(n))\, cost(3\gamma) = O(\gamma\log_\gamma n \cdot \log^2\gamma)$ times. Summing on all possible levels and applying Lemma 4, each element is only relabeled $O(\gamma\log^2 n)$ times. ◀

Theorem 1 is a special case of Theorem 10 obtained by setting $\gamma = 2$.

4.1 Achieving $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$

Proof of Theorem 3. Setting $\gamma = n^\epsilon$ in Theorem 10, the number of bits used is $1/\epsilon\lceil\log(12n^\epsilon + 1)\rceil = \log n + O(1/\epsilon)$. The maximum relabel bound becomes $O(n^\epsilon\log^2(n^\epsilon)) = O(n^\epsilon\log^2 n)$. That is when $\gamma = n^\epsilon$, it is an $\langle O(n^\epsilon\log^2 n), \log n + O(1/\epsilon)\rangle$ house numbering data structure. This bound is improved to $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$ by using the same solution with some constant $\epsilon' < \epsilon$. ◀

5 Conclusion

The house numbering problem is an interesting variant of the very well studied file maintenance and online list labelling problems. It poses some unique challenges that previous techniques do not solve. Our two data structures are able to come near optimal for the problem, but an $\langle O(\log n), O(\log n)\rangle$ house numbering data structure remains as an open problem.

References

1 Amihood Amir, Martin Farach, Ramana M Idury, Johannes A Lapoutre, and Alejandro A Schaffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.

2 Amihood Amir, Gianni Franceschini, Roberto Grossi, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Managing unbounded-length keys in comparison-driven data structures with applications to online indexing. *SIAM J. Comput.*, 43(4):1396–1416, 2014.

3 Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash-memory algorithms. In Yossi Azar and Thomas Erlebach, editors, *European Symp. on Algorithms (ESA)*, volume 4168 of *LNCS*, pages 100–111. Springer, 2006. doi:10.1007/11841036_12.

4 M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

5 M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 3(2):115–136, 2004.

6 Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In Rolf H. Möhring and Rajeev Raman, editors, *Euro. Symp. on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 152–164. Springer, 2002. doi:10.1007/3-540-45749-6_17.

7 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, Tsvi Kopelowitz, and Pablo Montes. File maintenance: When in doubt, change the layout! In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA)*, pages 1503–1522, 2017.

8 R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003. doi:10.1109/JPROC.2003.811702.

9 Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013.

10 Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Annual Symposium on Discrete Algorithms (SODA)*, pages 39–48, 2002. URL: http://www.brics.dk/~gerth/Papers/soda02.ps.gz.

11 Jan Bulánek, Michal Koucký, and Michael E. Saks. Tight lower bounds for the online labeling problem. *SIAM J. Comput.*, 44(6):1765–1797, 2015.

12 Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *5th Conf. on Innovative Data Systems Research (CIDR)*, pages 21–31, 2011. URL: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper3.pdf.

13 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005. doi:10.1137/S0097539700370539.

14 P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *19th ACM Symp. on Theory of Computing (STOC)*, pages 365–372, 1987. doi:10.1145/28395.28434.

15 Paul F. Dietz. Fully persistent arrays (extended array). In *Algorithms and Data Structures, Workshop WADS'89, Ottawa, Canada, August 17-19, 1989, Proceedings*, pages 67–74, 1989.

16 Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 78–88, 1991.

17 Yuval Emek and Amos Korman. New bounds for the controller problem. *Distributed Computing*, 24(3-4):177–186, 2011.

18 David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, 44(5):669–696, 1997.

19 David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Paweł Pszona. Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket. In Joachim Gudmundsson and Jyrki Katajainen, editors, *13th Int. Symp. on Experimental Algorithms (SEA)*, volume 8504 of *LNCS*, pages 162–173, Cham, 2014. Springer.

20 David Eppstein, Michael T Goodrich, and Jonathan Z Sun. Skip quadtrees: Dynamic data structures for multidimensional point sets. *International Journal of Computational Geometry & Applications*, 18(01n02):131–160, 2008.

21 Sandy Irani, Moni Naor, and Ronitt Rubinfeld. On the time and space complexity of computation using write-once memory or is pen really much worse than pencil? *Mathematical Systems Theory*, 25(2):141–159, 1992. doi:10.1007/BF02835833.

22 Alon Itai, Alan G Konheim, and Michael Rodeh. *A sparse table implementation of priority queues*. Springer, 1981.

23 Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *IEEE Symp. on Found. of Comp. Sci. (FOCS)*, pages 283–292, 2012. doi:10.1109/FOCS.2012.79.

24 Tsvi Kopelowitz, Gregory Kucherov, Yakov Nekrich, and Tatiana A. Starikovskaya. Cross-document pattern matching. *J. Discrete Algorithms*, 24:40–47, 2014.

25 P. Pavan, R. Bez, P. Olivo, and E. Zanoni. Flash memory cells—an overview. *Proceedings of the IEEE*, 85(8):1248–1271, 1997. doi:10.1109/5.622505.

26 Dan E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 251–260, 1986.

27 D.E. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proc. 14th Annual Symposium on Theory of Computing (STOC)*, pages 114–121, 1982.

28 H.-S.P. Wong, S. Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, B. Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010. doi:10.1109/JPROC.2010.2070050.

Temporal Clustering*†

Tamal K. Dey[1], Alfred Rossi[2], and Anastasios Sidiropoulos[3]

1 **Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA**
dey.8@osu.edu
2 **Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA**
rossi.49@osu.edu
3 **Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA**
sidiropoulos.1@osu.edu

Abstract

We study the problem of clustering sequences of unlabeled point sets taken from a common metric space. Such scenarios arise naturally in applications where a system or process is observed in distinct time intervals, such as biological surveys and contagious disease surveillance. In this more general setting existing algorithms for classical (i.e. static) clustering problems are not applicable anymore.

We propose a set of optimization problems which we collectively refer to as *temporal clustering*. The quality of a solution to a temporal clustering instance can be quantified using three parameters: the number of clusters k, the spatial clustering cost r, and the maximum cluster displacement δ between consecutive time steps. We consider spatial clustering costs which generalize the well-studied k-center, discrete k-median, and discrete k-means objectives of classical clustering problems. We develop new algorithms that achieve trade-offs between the three objectives k, r, and δ. Our upper bounds are complemented by inapproximability results.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms, I.5.3 Clustering

Keywords and phrases clustering, multi-objective optimization, dynamic metric spaces, moving point sets, approximation algorithms, hardness of approximation.

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.34

1 Introduction

Clustering points in a metric space is a fundamental problem that can be used to express a plethora of tasks in machine learning, statistics, and engineering, and has been studied extensively both in theory and in practice [4, 8, 13, 19, 20, 21, 23, 24, 26, 27, 29, 31]. Typically, the input consists of a set P of points in some metric space and the goal is to compute a partition of P minimizing a certain objective, such as the number of clusters given a constraint on their diameters.

We study the problem of clustering *sequences* of *unlabeled* point sets taken from a common metric space. Our goal is to cluster the points in each 'snapshot' so that the cluster assignments remain coherent across successive snapshots (across time). We formulate the

* This work was partially supported by the NSF grants CCF 1318595, CCF 1423230, DMS 1547357, and NSF award CAREER 1453472.
† A full version of the paper is available at http://arxiv.org/abs/1704.05964.

problem in terms of tracking the *centers* of the clusters that may merge and split over time while satisfying certain constraints. Such instances are common in the study of time-evolving processes and phenomena under discrete observation. As an example consider a hypothetical study which aims to track the spread of a certain genetic mutation in plants. Here, data collection efforts center on annual field surveys in which a technician collects and catalogs samples. The location and number of mutation positive specimens change from year to year. Clustering such spaces is clearly a generalization of classical (static) clustering, which we refer to as *temporal clustering*. In this dynamic variant of the problem, apart from the number of clusters and their radii, we also wish to minimize the extent by which each cluster moves between consecutive snapshots.

Related work. Clustering of moving point sets has been studied in the context of *kinetic clustering* [2, 6, 17, 18, 16, 14, 1]. In that setting points have identities (labels) which are fixed throughout their motion, the trajectories of the points are known beforehand, and the goal is to design a data structure which can efficiently compute a near-optimal clustering for any given time step. In our setting, since the points are not labeled there is, *a priori*, no explicit motion. Instead we are given a sequence of unlabeled points in a metric space and are required to assign the points of each to a limited number of temporally coherent clusters. Motion emerges as a consequence of cluster assignment. Consequently, kinetic clustering algorithms cannot be used in our setting. Another related problem concerns clustering time series under the Fréchet distance [11], with the clusters being constrained to move along polygonal trajectories of bounded complexity. This constraint is used to avoid overfitting, and is conceptually similar to our requirement that the clusters remain close between snapshots.

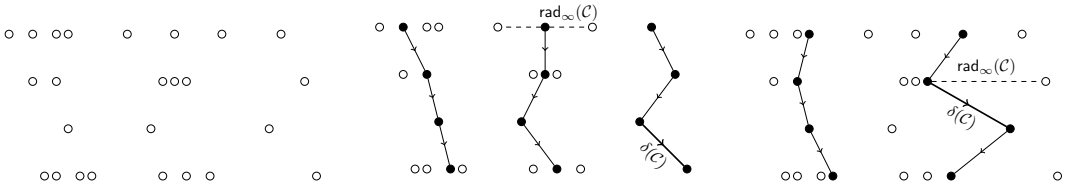
1.1 Problem formulations

Let us now formally define the algorithmic problems that we study in this paper. Perhaps surprisingly, very little is known for temporal clustering problems. There are of course different optimization problems that one could define; here we propose what we believe are the most natural ones.

We first define how the input to a temporal clustering problem is described. Let $M = (X, d)$ be a metric space. Let $P(1), \ldots, P(t)$ be a sequence of t finite, non-empty metric subspaces (points) of M. We refer to individual elements of this sequence (the 'snapshots') as *levels*, and collectively to P as a *temporal-sampling of* M of *length* t. The *size* of P is the total number of points over all levels, that is $\sum_{i\in[t]} |P(i)|$. Let $\{\tau(i)\}_{i=1}^{t}$ be a sequence of points such that $\tau(i) \in P(i)$ is a single point. We say that τ is a *trajectory* of P, and we let $\mathcal{T}(P)$ denote the set of all possible trajectories of P. For some $\mathcal{C} \subseteq \mathcal{T}(P)$, we denote by $\mathcal{C}(i)$ the set of points of the trajectories in $\mathcal{C}$ which lie in $P(i)$. In other words, $\mathcal{C}(i) = \bigcup_{\tau\in\mathcal{C}} \tau(i)$. The set of trajectories $\mathcal{C}$ induces a clustering on each level $P(i)$ by assigning each $p \in P(i)$ to the trajectory $\tau \in \mathcal{C}$ that minimizes $d(p, \tau(i))$. We refer to the points of $\mathcal{C}(i)$ as the *centers* of level i. Intuitively, this formulation allows points in different levels of P which are assigned to the same trajectory to be part of the same cluster; see Figure 1. Further, observe that trajectories may overlap allowing clusters to merge and split implicitly; see Figure 3a. We refer to $\mathcal{C}$ as a *temporal-clustering* of P.

We now formalize the clustering objectives. Our approach is to treat temporal clustering as a multi-objective optimization problem where we try to find a collection of trajectories such that their induced clustering ensures three conditions:

(i) points in the same cluster remain near between successive levels (*locality*),



(a) P **(b)** A temporal 3-clustering **(c)** A temporal 2-clustering

Figure 1 (1a) A temporal-sampling P of length 4 where $P(i) \subset \mathbb{R}$ is drawn horizontally. Each level of P is depicted as a row starting from $P(1)$ at the top. (1b) The temporal-sampling P shown with a clustering $\mathcal{C}$, consisting of 3 clusters. The centers of each of 3 trajectories are depicted as filled circles in each level. Arrows are drawn between $\tau_j(i)$ and $\tau_j(i+1)$ for each trajectory τ_j, $j \in \{1,2,3\}$. In the first level, a pair of points which achieve the spatial cost are joined to their respective cluster centers by a dashed edge. The arrow between the pair of centers which achieves maximum displacement is shown in bold. (1c) The temporal-sampling P shown with 2 clusters.

(ii) the restriction of the clustering to any single level fits the shape of the data (*spatial constraint*), and

(iii) we do not return excessively many clusters (*complexity*).

To measure how far some trajectory τ jumps, we define its *displacement*, denoted by $\delta(\tau)$, to be $\delta(\tau) = \max_{i \in [t-1]} d(\tau(i), \tau(i+1))$. We also define the displacement of $\mathcal{C}$ to be $\delta(\mathcal{C}) = \max_{\tau \in \mathcal{C}} \delta(\tau)$. Finally, we consider three different objectives for the spatial cost, which correspond to generalization of the k-center, k-median, and k-means respectively. The first one, corresponding to k-center, is the maximum over all levels of the maximum cluster radius; formally $\mathsf{rad}_\infty(\mathcal{C}) = \max_{i \in [t]} \max_{p \in P(i)} d(p, \mathcal{C}(i))$, where $d(p, \mathcal{C}(i)) = \min_{\tau \in \mathcal{C}} d(p, \tau(i))$. The second and third spatial cost objectives, which corresponding to discrete k-median, and discrete k-means (respectively), are defined to be $\mathsf{rad}_1(\mathcal{C}) = \max_{i \in [t]} \sum_{p \in P(i)} d(p, \mathcal{C}(i))$, and $\mathsf{rad}_2(\mathcal{C}) = \max_{i \in [t]} \sum_{p \in P(i)} d(p, \mathcal{C}(i))^2$.

▶ **Definition 1.** Let $r \in \mathbb{R}_{\geq 0}$, $\delta \in \mathbb{R}_{\geq 0}$. We say that a set of trajectories $\mathcal{C} \subseteq \mathcal{T}(P)$ is a *temporal* (k,r,δ)*-clustering* of P if $\mathsf{rad}_\infty(\mathcal{C}) \leq r$, $\delta(\mathcal{C}) \leq \delta$, and $|\mathcal{C}| \leq k$. (See Figure 1 for an example.) We further define *temporal* (k,r,δ)*-median-clustering* and (k,r,δ)*-means-clustering* analogously by replacing rad_∞ by rad_1 and rad_2 respectively.

We now formally define the optimization problems that we study. In the case of static clustering, a natural objective is to minimize the maximum cluster radius, subject to the constraint that only k clusters are used; this is the classical k-Center problem [23]. Another natural objective in the static case is to minimize the number of clusters subject to the constraint that the radius of each cluster is at most r, for some given $r > 0$; this is the r-Dominating Set problem [22]. Our definition of temporal clustering includes the temporal analogues of k-Center and r-Dominating Set as special cases.

▶ **Definition 2** (Temporal (k,r,δ)-Clustering problem)**.** An instance of the Temporal (k,r,δ)-Clustering problem is a tuple (M, P, k, r, δ), where M is a metric space, P is a temporal-sampling of M, $k \in \mathbb{N}$, $r \in \mathbb{R}_{\geq 0}$, and $\delta \in \mathbb{R}_{\geq 0}$. The goal is to decide whether P admits a temporal (k,r,δ)-clustering.

▶ **Definition 3** (Temporal (k,r,δ)-Clustering approximation)**.** Given an instance of the Temporal (k,r,δ)-Clustering problem consisting of a tuple (M, P, k, r, δ), a (α,β,γ)-*approximation* is an algorithm which either returns a temporal $(\alpha k, \beta r, \gamma\delta)$-clustering of P, or correctly decides that no temporal (k,r,δ)-clustering exists. In general α, β, and γ can be functions of the input.

We analogously define the Temporal (k,r,δ)-Median Clustering problem and approximation, and the Temporal (k,r,δ)-Means Clustering problem and approximation by replacing in Definitions 2 and 3 $(\cdot,\cdot,\cdot)$-clustering by $(\cdot,\cdot,\cdot)$-median-clustering and $(\cdot,\cdot,\cdot)$-means-clustering respectively.

1.2 Our contribution

To the best of our knowledge, this is the first study of the above models of temporal clustering. Our main contributions consist of polynomial-time approximation algorithms for several temporal clustering variants, and hardness of approximation results for others.

Temporal clustering. We begin by discussing our results on Temporal (k,r,δ)-Clustering. We first consider the problem of minimizing r and δ while keeping k fixed. This is a generalization of the static k-Center problem. We present a polynomial-time $(1, 2, 1+2\varepsilon)$-approximation algorithm where $\varepsilon = r/\delta$ using a different method. More specifically, our result is obtained via a reduction to a network flow problem. We show that the problem is NP-hard to approximate to within polynomial factors even if we increase the radius by a polynomial factor. Formally, we show that it is NP-hard to obtain a $(1, \mathsf{poly}(n), \mathsf{poly}(n))$-approximation.

Next we consider the problem of minimizing the number of clusters k, while fixing r and δ. This is a generalization of the static r-Dominating Set problem. We obtain a polynomial-time $(\ln n, 1, 1)$-approximation algorithm. For the static case, the polynomial-time $(\ln n)$-approximation algorithm follows by a reduction to the Set-Cover problem, and is known to be best-possible [10, 30, 12]. However, in the temporal case, this reduction produces an instance of Set-Cover of exponential size. Thus, it does not directly imply a polynomial-time algorithm for Temporal r-Dominating Set. We bypass this obstacle by showing how to run the greedy algorithm for Set-Cover on this exponentially large instance in polynomial-time, without explicitly computing the Set-Cover instance. We also argue that $(\ln n, 1, 1)$-approximation is best possible by observing that $((1-\varepsilon)\ln n, 2-\varepsilon', \cdot)$-approximation is NP-hard for any $\varepsilon, \varepsilon' > 0$.

We further present a result that can be thought of as a trade-off between the above two settings by allowing both the number of clusters and the radius to increase. More precisely, we obtain a polynomial-time $(2, 2, 1+\varepsilon)$-approximation algorithm where $\varepsilon = r/\delta$. Interestingly, we can show that obtaining a $(1.005, 2-\varepsilon, \mathsf{poly}(n))$-approximation is NP-hard.

The following summarizes the above approximation algorithms.

▶ **Theorem 4.** Temporal (k,r,δ)-Clustering *admits the following algorithms:*
1. $(1,2,1+2\varepsilon)$*-approximation where* $\varepsilon = r/\delta$,
2. $(\ln(n),1,1)$*-approximation,*
3. $(2,2,1+\varepsilon)$*-approximation where* $\varepsilon = r/\delta$,

where n is the size of the temporal-sampling. Moreover, the running time of all of these algorithms is $O(n^3)$.

We prove Theorems 1, 2, 3 in Sections 2.1, 2.2, 2.3, respectively.

It is important that the approximation in displacements for Theorem 1 and Theorem 3 takes into account the factor $\varepsilon = r/\delta$ if a polynomial time algorithm is aimed for. This is because our inapproximability results as summarized below show that the problem is NP-hard otherwise.

▶ **Theorem 5.** *The status of* Temporal (k,r,δ)-Clustering *with temporal-samplings of size n is as follows:*

1. *There exist universal constants* $c > 0$, $c' > 0$ *such that* $(1,cn^{s(1-\varepsilon)},c'n^{(1-s)(1-\varepsilon)})$*-approximation is* NP*-hard for any* $\varepsilon, s \in \mathbb{R}$ *where* $\varepsilon > 0$ *and* $s \in [0,1]$.
2. $((1-\varepsilon)\ln(n),2-\varepsilon',\cdot)$*-approximation is* NP*-hard for any fixed* $\varepsilon > 0$, $\varepsilon' > 0$.
3. *There exists a universal constant* c *such that* $(1.00579,2-\varepsilon',cn^{1-\varepsilon})$*-approximation is* NP*-hard for any fixed* $\varepsilon > 0$, $\varepsilon' > 0$.

Moreover, items 1 and 3 remain NP*-hard even for temporal-samplings in 2-dimensional Euclidean space.*

Due to space constraints, we defer extended discussion of Theorem 1, Theorem 2, and Theorem 3 to the full version of the paper [9].

Temporal median clustering. We next discuss our result on the Temporal (k,r,δ)-Median Clustering problem. The static k-Median problem admits an $O(1)$-approximation via local search [5, 28]. In Section 2.4 we show that the local search approach fails in the temporal case, even on temporal samplings of length two. We present an algorithm that achieves a trade-off between the number of clusters and the spatial cost. The result is obtained via a greedy algorithm, which is similar to the one used for the k-Set Cover problem. The result is summarized in the following theorem.

▶ **Theorem 6.** *For any fixed* $\varepsilon > 0$, *there exists an* $(O(\log(n\Delta/\varepsilon)),\ 1+\varepsilon, 1)$*-median-approximation algorithm with running time* $\mathsf{poly}(n, \log(\Delta/\varepsilon))$, *on an instance of size* n *and a metric space of spread* Δ.

The result is obtained by iteratively selecting a trajectory which minimizes a certain potential function. The proof uses submodularity and monotonicity of the potential function. These properties remain true if the potential function is modified by replacing $d(p,\mathcal{C}(i))$ with $d(p,\mathcal{C}(i))^2$, and thus an identical theorem holds for Temporal k-Means.

We complement the above algorithm by showing the following hardness result.

▶ **Theorem 7.** *The status of* Temporal (k,r,δ)-Median Clustering *with temporal-samplings of size* n *is as follows:*

1. *There exist universal constants* c_r, c_δ *such that* $(1,c_r n^{s(1-\varepsilon)},c_\delta n^{(1-s)(1-\varepsilon)})$*-approximation for* Temporal k-Median *is* NP*-hard for any* $\varepsilon, s \in \mathbb{R}$ *where* $\varepsilon > 0$ *and* $s \in [0,1]$.
2. *Let* c, s *be the constants from Theorem 4.6 (3) in [7]. Let* $0 \le f < c - s$. *Then* $(\frac{3-(s+f)}{3-c}, 1+c_r f, c_\delta n^{1-\varepsilon})$*-approximation is* NP*-hard for any fixed* $\varepsilon > 0$ *and some constants* c_r, c_δ.

Moreover, item 1 remains hard even for temporal-samplings from 2-dimensional Euclidean space.

The clustering instances used in the proofs of Theorem 1 and Theorem 2 involve clusterings which use only a constant number of points per cluster, thus the same constructions suffice to prove hardness of Temporal (k,r,δ)-Means Clustering with only slight modification of the distances. See the discussion in the full version [9].

Additional notation and preliminaries. Let $r > 0$. An r-net in some metric space (X,d) is some maximal $Y \subseteq X$, such that for any $x, y \in Y$, with $x \neq y$, we have $d(x,y) > r$. Let P be a temporal-sampling of length t in some metric space (X,d). Let $V(P,i) = \bigcup_{x \in P(i)} \{(i,x)\}$ for all $i \in [t]$. For any trajectory τ, and for any $r \ge 0$, the *tube* around τ of *radius* r, denoted by $\mathsf{tube}(\tau,r)$, is defined to be $\mathsf{tube}(\tau,r) = \bigcup_{i \in [t]} \{(i,x) \in V(P,i) \mid x \in \mathsf{ball}(\tau(i),r)\}$, where for $x \in X$, $r \in \mathbb{R}_{\ge}0$, we use the notation $\mathsf{ball}(x,r)$ to denote a closed ball of radius r. Let $\delta \in \mathbb{R}_{\ge 0}$. The directed graph $G_\delta(P)$ has as vertices $V(P,i)$ for all $i \in [t]$. For any $i \in [t-1]$ there is an edge between $p \in V(P,i)$ and $q \in V(P,i+1)$ whenever $d(p,q) \le \delta$ (see Figure 2).

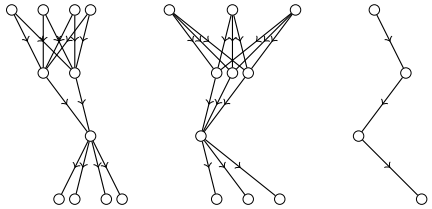


Figure 2 The graph $G_\delta(P)$ for P from the previous diagram and some δ. Points which are within a distance of δ in adjacent levels are connected by a directed edge which points toward the higher indexed level.

2 Algorithms

2.1 Exact number of clusters: (1,2,1 + 2ε)-approximation

In this section, we consider the problem of computing a temporal clustering by relaxing the radius and the displacement, while keeping the number of clusters exact. This is a temporal analogue of the k-Center problem. We first present a polynomial time $(1,2,1+2\varepsilon)$-approximation where $\varepsilon = r/\delta$. In the full version [9], we complement this with an inapproximability result.

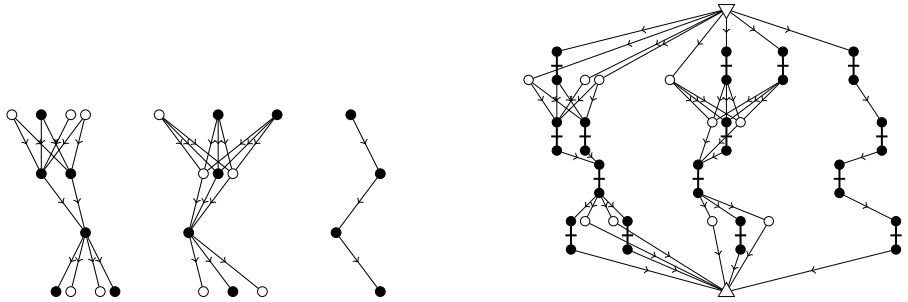
An auxiliary network flow problem. The high-level idea of the polynomial time algorithm is to use a reduction to a specific network flow problem. Specifically, we seek a minimum flow which satisfies lower bound constraints along certain edges. This is the so-called minimum flow, or minimum feasible flow problem [3, 15]. We now formally define this flow network. For each $i \in [t]$, let $C(i) \subseteq P(i)$. Let $\gamma > 0$. We construct a flow network, denoted by $N_\gamma(P, C)$ where C is the sequence of centers $C(i)$ for $i \in [t]$. We start with the graph $G_\gamma(P)$. In level i, we replace each vertex $v = (i, c)$ for $c \in C(i)$ by a pair of vertices $\mathsf{tail}(v)$ and $\mathsf{head}(v)$, and we connect them by an edge $(\mathsf{tail}(v), \mathsf{head}(v))$. For vertices $v = (i, p)$ where $p \in P(i) \setminus C(i)$ we define $\mathsf{tail}(v) = \mathsf{head}(v) = v$. Now for *any* vertex v, all incoming edges to v become incoming edges to $\mathsf{tail}(v)$, and all outgoing edges from v become outgoing edges from $\mathsf{head}(v)$. We add a source vertex s and a sink vertex s'. For all $p \in P(1)$, we add an edge from s to $\mathsf{tail}((1, p))$. Similarly, for all $p \in P(t)$, we add an edges from $\mathsf{head}((t, p))$ to s'. We set the capacity of each edge to be ∞. Finally, we set a lower bound of 1 to the capacity of every edge $(\mathsf{tail}(v), \mathsf{head}(v))$, for all $v = (i, c)$, $c \in C(i)$, $i \in [t]$ (see Figure 3b).

Algorithm. We first compute a net at every level of the temporal-sampling and then we reduce the problem of computing a temporal clustering to a flow instance, using the network flow defined above. By computing an integral flow and decomposing it into paths, we obtain a collection of trajectories. The lower bound constraints ensure that all net points are covered; this allows us to show that all points are covered by the tubular neighborhoods of the trajectories. Formally, the algorithm consists of the following steps:

Step 1: Computing nets. For each $i \in [t]$, compute a $2r$-net $C(i)$ of $P(i)$. If for some $i \in [t]$, $|C(i)| > k$, then return nil.

Step 2: Constructing a flow instance. We construct the minimum flow instance $N_{2r+\delta}(P, C)$.

Step 3: Computing a collection of trajectories. If the flow instance $N_{2r+\delta}(P, C)$ is not feasible, then return nil. Otherwise, find a minimum integral flow F in $N_{2r+\delta}(P, C)$, satisfying all the lower bound constraints. Decompose F into a collection of paths, each



(a) The graph $G_\gamma(P)$. **(b)** The flow network $N_\gamma(P, C)$.

Figure 3 (3a) The graph $G_\gamma(P)$ for $P(i) \subset \mathbb{R}$ and some $\gamma > 0$. The vertices in $C(i)$, $i \in [4]$, are indicated with filled circles. (3b) The flow network $N_\gamma(P, C)$ corresponding to $G_\gamma(P)$. Every node from C has been split into an edge.

carrying a unit of flow. The restriction of each path in G is a trajectory. Output the set of all these trajectories.

Throughout the rest of this section let P be a temporal-sampling. We now show that if there exists a temporal (k,r,δ)-clustering, then the above algorithm outputs a temporal $(k, 2r, (1 + 2\varepsilon)\delta)$-clustering where $\varepsilon = r/\delta$.

▶ **Lemma 8.** *Suppose that P admits a temporal (k,r,δ)-clustering, $\mathcal{Q}$. For each $i \in [t]$ let $Q(i)$ denote the level i centers of $\mathcal{Q}$, and let $C(i)$ be a $2r$-net of $P(i)$. Then the map $\pi_i : C(i) \to Q(i)$ which sends each $2r$-net center to a nearest center in $Q(i)$ is injective.*

Proof. First, observe that for each $c \in C(i)$, $d(c, \pi_i(c)) \leq r$ because r-balls centered at the points in $Q(i)$ cover $P(i)$ and hence $C(i)$. For injectivity of π_i, observe that, $\pi_i(c) \neq \pi_i(c')$ for $c \neq c'$ because otherwise the inequality $d(c, c') \leq d(c, \pi_i(c)) + d(c', \pi_i(c')) \leq 2r$ holds violating the property that $C(i)$ is a $2r$-net. ◀

Since for each $i \in [t]$, the map π_i is injective, it follows that $|C(i)| \leq |Q(i)| \leq k$. So, we have the following immediate Corollary.

▶ **Corollary 9.** *If P admits a temporal (k,r,δ)-clustering then for any $i \in [t]$, any $2r$-net $C(i)$ of $P(i)$ has $|C(i)| \leq k$.*

▶ **Lemma 10.** *If P admits a temporal (k,r,δ)-clustering then for any level-wise $2r$-net C, the flow instance $N_{2r+\delta}(P, C)$ admits a feasible flow of value k.*

Proof. Fix a temporal (k,r,δ)-clustering $\mathcal{Q}$ and let τ denote one of its k trajectories. The graph $G_{2r+\delta}(P)$ contains a path corresponding to τ as the distance between any pair of consecutive points in P is at most δ. For each i, let $\pi_i : C(i) \to Q(i)$ denote a map which sends each $2r$-center of $C(i)$ to a nearest center in $Q(i)$. We modify τ to produce some path τ' in $G_{2r+\delta}(P)$ as follows: for every level i such that $\tau(i) = \pi_i(c_i)$ for some net-point $c_i \in C(i)$ we let $\tau'(i) = c_i$, otherwise we set $\tau'(i) = \tau(i)$. We observe that in the worst case the distance between consecutive points, say $u = \tau'(i)$ and $v = \tau'(i + 1)$, is at most $2r + \delta$ because of the following inequality (see Figure 4) $d(u, v) \leq d(u, \tau(i)) + d(\tau(i), \tau(i + 1)) + d(\tau(i + 1), v) \leq r + \delta + r$. It follows that τ' is indeed a path in $G_{2r+\delta}(P)$. Further, by the injectivity of each map π_i (Lemma 8) which is used in deforming τ to τ', we have that for every net point, there exists some τ' that contains it. In other words, all net points $C(i)$ are covered by the paths τ'. For each optimal trajectory τ, let τ'' be the path in $N_{2r+\delta}(P, C)$ obtained from τ'

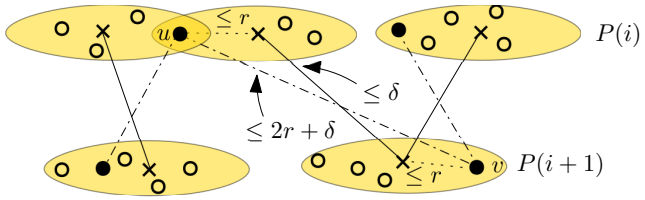


Figure 4 The crosses, filled circles, and empty circles are optimal centers, net points, and other points respectively. The paths (τ) in optimal solution and the deformed paths(τ') are indicated with solid and dotted edges respectively.

by connecting s to the first vertex in τ', and the last vertex in τ' to t. By routing a unit of flow in $N_{2r+\delta}(P, C)$ along each such τ'' we obtain a flow of value at most k that meets all the demands along the edges corresponding to net points C, concluding the proof. ◀

▶ **Lemma 11.** *Given k, r, δ, and a temporal-sampling P, with $|P| = n$, there exists an $O(n^3)$-time algorithm that either correctly decides P does not admit a temporal (k,r,δ)-clustering, or outputs some temporal $(k,2r,2r+\delta)$-clustering.*

Proof. Lemmas 9 and 10 imply that if a temporal (k,r,δ)-clustering exists, then the algorithm does not return nil, and thus outputs a set T of at most k trajectories. Let $\mathcal{C}$ be the temporal clustering corresponding to T. Each trajectory in T corresponds to a path in $G_{2r+\delta}(P)$, thus has displacement at most $2r+\delta$. Therefore $\delta(\mathcal{C}) \leq 2r+\delta$. Since F is a feasible flow, it follows that all lower bound constraints in $N_{2r+\delta}(P, C)$ are satisfied. Thus for all $i \in [t]$, for all $c \in C(i)$, there exists at least one unit of flow along the edge $(\mathsf{tail}(v), \mathsf{head}(v))$ corresponding to the vertex $v = (i, c)$; it follows that there exists some trajectory containing c in level i. Since for all $i \in [t]$, $C(i)$ is a $2r$-net of $P(i)$, it follows that $P(i) \subseteq \bigcup_{c \in C(i)} \mathsf{ball}(c, 2r)$. Thus $\bigcup_{i\in[t]} V(P, i) \subseteq \bigcup_{\tau\in T} \mathsf{tube}(\tau, 2r)$, which implies that $\mathsf{rad}_\infty(\mathcal{C}) \leq 2r$. We thus obtain that $\mathcal{C}$ is a temporal $(k,2r,2r+\delta)$-clustering. Finally, we bound the running time. Computing the $2r$-nets over all levels, checking their sizes can be done in $O(nk)$ time. Building $G_{2r+\delta}(P)$ and $N_{2r+\delta}(P, C)$ can be done in $O(n^2)$ time. Finding an integral solution to $N_{2r+\delta}(P, C)$ takes $O(n^3)$ time using the algorithm of Gabow and Tarjan [15]. Decomposing the resulting flow takes $O(n^3)$ time. We conclude that the entire procedure completes in $O(n^3)$ time. ◀

Writing $\varepsilon = r/\delta$, we immediately obtain Theorem 1 from Lemma 11.

2.2 Exact radius and displacement: $(\ln(n),1,1)$-approximation

In this section we consider the case where the number of clusters is allowed to be approximated in analogy to the static r-Dominating Set problem. We present a polynomial-time $(\ln(n),1,1)$-approximation algorithm. In the full version [9], we argue that this result is tight in the sense that obtaining a $((1-\varepsilon)\ln(n),1,1)$-approximation is NP-hard for any fixed $\varepsilon > 0$.

Let P be a temporal-sampling of length t. For any $\delta \geq 0$, we denote by $\mathcal{T}_\delta(P)$ the set of all trajectories of displacement at most δ. Given an instance of the Temporal (k,r,δ)-Clustering problem consisting of a tuple (M, P, k, r, δ), the high level idea is to express the problem as an instance of Set-Cover. Recall that an instance of Set-Cover consists of a pair $(U, \mathcal{S})$, where U is a set, and $\mathcal{S}$ is a collection of subsets of U. The goal is to find some $\mathcal{S}' \subseteq \mathcal{S}$, minimizing $|\mathcal{S}'|$, such that $U \subseteq \bigcup_{X\in\mathcal{S}'} X$, if such $\mathcal{S}'$ exists. We set $U = \bigcup_{i\in[t]} V(P, i)$, and $\mathcal{S} = \bigcup_{\tau\in\mathcal{T}_\delta(P)} \{\mathsf{tube}(\tau, r)\}$. We will show that a solution to the Set-Cover instance $(U, \mathcal{S})$ can be used to obtain a temporal $(\ln(n)k,r,\delta)$-clustering. Note

that $\mathcal{S}$ can have cardinality exponential in the size of the input. However, as we shall see, we can still obtain an approximate solution for $(U, \mathcal{S})$ in polynomial-time.

We first establish that any $\alpha(n)$-approximate solution to $(U, \mathcal{S})$ can be converted, in polynomial-time, to a temporal $(\alpha(n)k,r,\delta)$-clustering. Let s_{OPT} denote the minimum cardinality of any feasible solution for $(U, \mathcal{S})$ when it exists. Similarly, let k_{OPT} denote the smallest value of k' such that P admits a temporal (k',r,δ)-clustering.

▶ **Lemma 12.** $k_{\text{OPT}} = s_{\text{OPT}}$.

The proof of Lemma 12 is deferred to the full version [9]. We next establish the following result which allows us to run the greedy algorithm for Set-Cover on the instance $(U, \mathcal{S})$ in polynomial-time, even though $|\mathcal{S}|$ can be exponentially large.

▶ **Lemma 13.** *Let $\mathcal{S}' \subsetneq \mathcal{S}$. There exists an $O(n^2)$ time algorithm which computes some $X \in \mathcal{S} \setminus \mathcal{S}'$, maximizing $\left|X \cap \left(U \setminus \bigcup_{Y \in \mathcal{S}'} Y\right)\right|$. Moreover, the algorithm outputs some trajectory $\tau \in \mathcal{T}_\delta(P)$, such that $X = \{\mathsf{tube}(\tau, r)\}$.*

The proof of Lemma 13 is deferred to the full version [9]. We are now ready to prove Theorem 2.

Proof of Theorem 2. Recall that the classical greedy algorithm for Set-Cover computes a solution $\mathcal{S}' \subseteq \mathcal{S}$, if one exists, as follows: Initially, we set $\mathcal{S}' = \varnothing$. At every iteration, we pick some $X \in \mathcal{S} \setminus \mathcal{S}'$ such that $\left|X \cap \left(U \setminus \bigcup_{Y \in \mathcal{S}'} Y\right)\right|$ is maximized, and we add X to $\mathcal{S}$. The algorithm stops when either U is covered by $\mathcal{S}$, or when no further progress can be made, i.e. when $\left|X \cap \left(U \setminus \bigcup_{Y \in \mathcal{S}'} Y\right)\right| = 0$; in the latter case, the instance $(U, \mathcal{S})$ is infeasible. It is well-known that this algorithm achieves an approximation ratio of $\ln n$ for Set-Cover [25]. Now if $(U, \mathcal{S})$ is infeasible the above procedure detects this and terminates. Otherwise, let $\mathcal{S}' \subseteq \mathcal{S}$ be the feasible solution found by repeatedly using the procedure described in Lemma 13. The corresponding trajectories returned by this procedure form a temporal (k',r,δ)-clustering of P, for some $k' = |\mathcal{S}'| \leq \ln n \cdot s_{\text{OPT}}$. By Lemma 12 it follows that $k' \leq \ln n \cdot k_{\text{OPT}} \leq \ln n \cdot k$. Thus we obtain an $(\ln(n),1,1)$-approximation. Finally, to bound the running time note that in the worst case, the total number of calls to the procedure in Lemma 13 is n since at every step we cover at least uncovered point. The theorem now follows by the fact that each call takes $O(n^2)$ time. ◀

2.3 Approximating all parameters: $(2,2,1+\varepsilon)$-approximation

So far we have constrained either the number of clusters or the radius and the displacement to be exact. We now describe an algorithm that relaxes all three parameters simultaneously. We present a polynomial-time $(2, 2, 1 + \varepsilon)$-approximation algorithm where $\varepsilon = r/\delta$. We complement this solution in the full version [9] by showing that it is NP-hard to obtain a $(1.005, 2 - \varepsilon, \mathsf{poly}(n))$-approximation for any $\varepsilon > 0$.

▶ **Lemma 14.** *If P admits a temporal (k,r,δ)-clustering then for any level-wise $2r$-net C, the flow instance $N_{r+\delta}(P, C)$ admits a feasible flow of value $2k$.*

Proof. Fix a temporal (k,r,δ)-clustering $\mathcal{C} = \{\tau_i\}_{i=1}^k$. We inductively define a sequence $\mathcal{Q}_0, \ldots, \mathcal{Q}_t$, where for each $i \in \{0, \ldots, t\}$, $\mathcal{Q}_i$ is a multiset of paths in $G_{r+\delta}(P)$. We set $\mathcal{Q}_0 = \{\sigma_1^1, \sigma_1^2, \ldots, \sigma_k^1, \sigma_k^2\}$, where for each $j \in [k]$, we have $\sigma_j^1 = \sigma_j^2 = \tau_j$. Next, we inductively define $\mathcal{Q}_i$, for some $i \in \{1, \ldots, t\}$. Starting with $\mathcal{Q}_i = \mathcal{Q}_{i-1}$, we proceed to modify $\mathcal{Q}_i$. By induction, it follows that the paths σ_j^1, σ_j^2, and τ_j share the same suffix at levels $i, \ldots, t$. Thus, $\tau_j(i) \in \sigma_j^1$ and $\tau_j(i) \in \sigma_j^2$. Now, for the modification, we consider each $c \in C(i)$, and proceed

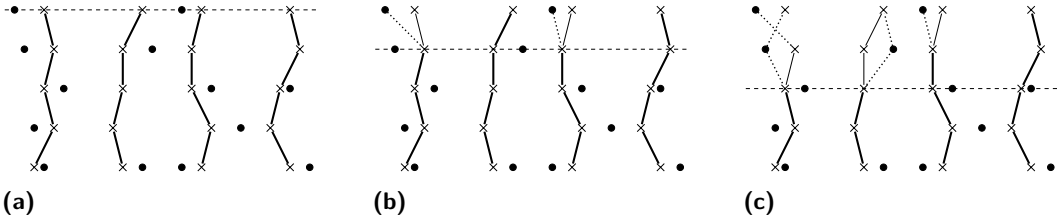


Figure 5 An example of the inductive construction of the multisets of paths $\mathcal{Q}_i$, for $i = 0$ (Figure 5a), $i = 1$ (Figure 5b), and $i = 2$ (Figure 5c). Dotted lines show where a trajectory has been rounded to a net point. Thin and thick solid lines indicate where one or two trajectories are coincident to an optimal trajectory, respectively. Initially (Figure 5a), $\mathcal{Q}_0$ consists of $2k$ trajectories $\sigma_j^1 = \sigma_j^2 = \tau_j$, for the trajectories of some optimal solution $\tau_1, \ldots, \tau_k$. At step i, for any $j \in [k]$ such that $\tau_j(i)$ is within a distance of r from some net point, c, we obtain $\mathcal{Q}_i$ by replacing $\tau_j(i)$ with c in either σ_j^1 or σ_j^2, depending on the parity of i.

as follows (see Figure 5 for an illustration). Since $\mathcal{C}$ is a valid temporal (k,r,δ)-clustering, it follows from Lemma 8 that there exists an injective map π_i from $C(i)$ to the set $\tau_1(i), \ldots, \tau_k(i)$ so that $\pi_i(c) = \tau_j(i)$ for some $j \in [k]$ and $d(\tau_j(i), c) \leq r$. We consider the following two cases:

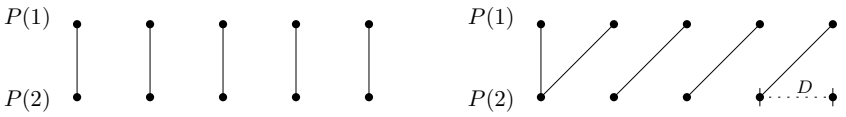
Case 1: If i is odd and $\tau_j(i) = \pi_i(c)$ for some $c \in C(i)$, then we modify σ_j^1 by replacing the vertex $\tau_j(i)$ with c.

Case 2: If i is even and $\tau_j(i) = \pi_i(c)$ for some $c \in C(i)$, then we modify σ_j^2 by replacing the vertex $\tau_j(i)$ with c.

We next argue that the result is indeed a path in $G_{r+\delta}(P)$. Suppose that in the above step, we modify the path σ_j^ℓ, for some $\ell \in \{1, 2\}$ so that $\sigma_j^\ell(i) = c$. It follows by induction on i that the path σ_j^ℓ was not modified when constructing $\mathcal{Q}_{i-1}$; thus $\sigma_j^\ell(i-1) = \tau_j(i-1)$. Since $\delta(\tau_j) \leq r$, it follows by the triangle inequality that $d(\sigma_j^\ell(i-1), \sigma_j^\ell(i)) = d(\tau_j(i-1), c) \leq d(\tau_j(i-1), \tau_j(i)) + d(\tau_j(i), c) \leq \delta + r$. It follows that $\delta(\sigma_j^\ell) \leq r + \delta$, which implies that each element of $\mathcal{Q}_i$ is indeed a path in $G_{r+\delta}(P)$. This completes the inductive definition of the multisets $\mathcal{Q}_0, \ldots, \mathcal{Q}_t$. It is immediate by induction that for each $i \in [t]$, for each $c \in C(i)$, there exist some path $\sigma \in \mathcal{Q}_t$ that visits c. We next transform the collection $\mathcal{Q}_t$ into a flow F in $N_{r+\delta}(P, C)$. For each path $\sigma \in \mathcal{Q}_t$, we obtain a path in the network $N_{r+\delta}(P, C)$ starting from the source s, then replacing for each $i \in [t]$, each $c \in C(i) \cap \sigma$ by the edge $(\mathsf{tail}(v), \mathsf{head}(v))$, for $v = (i, c)$, then terminating at the sink s'; we route a unit of flow along the resulting path. Since for each $i \in [t]$, there exists some path in $\mathcal{Q}_t$ visiting each $c \in C(i)$, it follows that all lower-bound constraints in $N_{r+\delta}(P, C)$ are satisfied by F. Since $\mathcal{Q}_t$ contains $2k$ paths, it follows that the value of the resulting flow is $2k$, as required. ◀

We are now ready to prove Theorem 3.

Proof of Theorem 3. For each $i \in [t]$, compute a $2r$-net $C(i)$ of $P(i)$, and construct the flow network $N_{r+\delta}(P, C)$. Compute a minimum flow F in $N_{r+\delta}(P, C)$ satisfying all lower-bound constraints. If $N_{r+\delta}(P, C)$ is infeasible (i.e. if there is no flow satisfying all lower bound constraints), or if the the value of the minimum flow in $N_{r+\delta}(P, C)$ is greater than $2k$, it follows by Lemma 14 that P does not admit a temporal (k,r,δ)-clustering. Thus, in this case the algorithm terminates. Otherwise, we compute a minimum flow in $N_{r+\delta}(P, C)$. Since all capacities and lower-bound constraints in $N_{r+\delta}(P, C)$ are integers, it follows that F can be taken to be integral. We decompose F into a collection of at most $2k$ paths, each carrying a unit of flow. Arguing as in Lemma 11 we have that the restriction of these paths on $G_{r+\delta}(P)$ is a set of trajectories that induces a valid temporal ($2k$,$2r$,$r + \delta$)-clustering of P.



(a) The clustering $\mathcal{C}^*$. **(b)** The clustering $\mathcal{C}$.

Figure 6 An example demonstrating that local search fails. Consider a temporal-sampling P of length 2 where $P(1) = P(2)$ consists of a sequence of 5 points where successive points are separated by D. (6a) A temporal $(5, r, \delta)$-median-clustering, $\mathcal{C}^*$, for any $r, \delta \in \mathbb{R}_{\geq 0}$ with $\mathsf{rad}_1(\mathcal{C}^*) = 0$. (6b) A temporal $(5, D, \delta)$-median-clustering, $\mathcal{C}$, for any $D \leq \delta < 2D$. Note that swapping any trajectory in $\mathcal{C}$ with one in $\mathcal{T}_\delta(P)$ is non-improving. The clustering $\mathcal{C}$ is therefore a local minimum of local search, yet the ratio $\mathsf{rad}_1(\mathcal{C})/\mathsf{rad}_1(\mathcal{C}^*)$ remains unbounded.

This provides a $(2,2,1+\varepsilon)$-approximation where $\varepsilon = r/\delta$. Finally, the running time is easily seen as $O(n^3)$ by the same argument that appears in Lemma 11, concluding the proof. ◀

2.4 Approximation algorithm for temporal median clustering

In this section we consider variants of Temporal Clustering which evaluate the spatial cost of clustering by taking the level-wise maximum of discrete k-median and discrete k-means objectives. A natural question is whether or not the problem admits a $O(1)$-approximation via local search, as in static case [5, 28]. In Figure 6 we show that the local search approach fails, even on temporal samplings of length two. Instead, the result is obtained by iteratively selecting a trajectory which most improves a certain potential function. The result in this section is presented for the Temporal (k,r,δ)-Median Clustering problem, and follows by submodularity and monotonicity of the potential function. These properties remain if $d(p, \mathcal{C}(i))$ is replaced with the $d(p, \mathcal{C}(i))^2$, and thus holds identically for Temporal k-Means.

We now present an approximation algorithm for the Temporal (k,r,δ)-Median Clustering problem. Let $\mathcal{I} = (M, P, k, r, \delta)$ be an input to the problem, where P is a temporal-sampling of length t. Let n denote the size of the P. Let also Δ denote the spread of $M = (X, d)$. That is, $\Delta = \frac{\operatorname{diam}(M)}{\inf_{p,q \in X}\{d(p,q) : d(p,q) > 0\}}$. Since we only consider finite metric spaces, and since the single point case is trivial, w.l.o.g. we may assume that the diameter of M is Δ and minimum interpoint distance in M is 1. For a set of trajectories $\mathcal{C}$ we define $\mathsf{cost}(i; \mathcal{C}) = \sum_{p \in P(i)} d(p, \mathcal{C}(i))$. We also define $W(\mathcal{C}) = \sum_{i=1}^{t} \max\{0, \mathsf{cost}(i; \mathcal{C}) - r\}$. Intuitively, the quantity $W(\mathcal{C})$ measures how far the solution $\mathcal{C}$ is from the optimum; in particular, if $W(\mathcal{C}) = 0$ then the spatial cost is within the desired bound.

▶ **Lemma 15.** *The set function $-W$ is submodular.*

Proof. Since the sum of submodular functions is submodular, it is enough to show that $-\max\{0, \mathsf{cost}(i; \mathcal{C}) - r\} = \min\{0, -\mathsf{cost}(i; \mathcal{C}) + r\}$ is submodular. Thus it suffices to show that $-\mathsf{cost}(i; \mathcal{C})$ is submodular, and thus it suffices to show that $-d(p, \mathcal{C}(i))$, for all $p \in P(i)$, which is immediate since $d(p, \mathcal{C}(i)) = \min_{\tau \in \mathcal{C}} d(p, \tau(i))$. ◀

Algorithm. Our goal is to compute some set of trajectories $\mathcal{C}$ such that $W(\mathcal{C})$ is sufficiently small, while minimizing $|\mathcal{C}|$. The algorithm consists of the following steps:
Step 1. Let $\mathcal{C}_0$ be a set containing a single arbitrary trajectory.
Step 2. For any $i \in [L]$, let τ_i be a minimizer of $W(\mathcal{C}_{i-1} \cup \{\tau_i\})$. Set $\mathcal{C}_i = \mathcal{C}_{i-1} \cup \{\tau_i\}$.
Step 3. Return $\mathcal{C}_L$.

The parameter $L > 0$ will be determined later. The following Lemma bounds the running time of Step 2.

▶ **Lemma 16.** *Given a clustering $\mathcal{C}$, we can find τ minimizing $W(\mathcal{C}\cup\{\tau\})$, in time* $\mathsf{poly}(|\mathcal{C}|, n)$.

The above Lemma can be done via dynamic programming. The proof is essentially the same as in Lemma 13 and is thus omitted. We next show that for some value of L, the algorithm computes a low cost solution. To that end, we argue that with each iteration of the main loop, $W(\mathcal{C}_i)$ decreases significantly.

▶ **Lemma 17.** *If $\mathcal{I}$ admits a temporal (k, r, δ)-median-clustering, then for any $i \in \{1, \ldots, L\}$, there exists some feasible trajectory σ_i such that $W(\mathcal{C}_{i-1} \cup \{\sigma_i\}) \leq (1 - 1/k) \cdot W(\mathcal{C}_{i-1})$.*

Proof. Let $\mathcal{C}^* = \{\tau_1^*, \ldots, \tau_{k'}^*\}$ be a set of at most k trajectories that yields a (k, r, δ)-median temporal clustering. W.l.o.g. we may assume that $k' = k$. Let $K_0 = W(\mathcal{C}_{i-1})$, and for any $j \in [k]$, let $K_j = W(\mathcal{C}_{i-1} \cup \{\tau_1^*, \ldots, \tau_j^*\})$. Since $\mathcal{C}^*$ is a (k, r, δ)-median temporal clustering, it follows that $W(\mathcal{C}_{i-1}) = K_0 \geq K_1 \geq \ldots \geq K_k = 0$. For any $j \in [k]$, we also define $K'_j = W(\mathcal{C}_{i-1} \cup \{\tau_j^*\})$. By Lemma 15 we have that for all $j \in [k]$, $W(\mathcal{C}_{i-1}) - W(\mathcal{C}_{i-1} \cup \{\tau_j^*\}) \geq W(\mathcal{C}_{i-1} \cup \{\tau_1^*, \ldots, \tau_{j-1}^*\}) - W(\mathcal{C}_{i-1} \cup \{\tau_1^*, \ldots, \tau_j^*\})$. That is, $K_0 - K'_j \geq K_{j-1} - K_j$. Let $\ell = \arg\max_{j \in [k]} \{K_0 - K'_j\}$. It follows that $K_0 - K'_\ell = \max_{j \in [k]} \{K_0 - K'_j\} \geq \max_{j \in [k]} \{K_{j-1} - K_j\} \geq \frac{1}{k} \sum_{j=1}^{k} (K_{j-1} - K_j) = (K_0 - K_k)/k = K_0/k$. Let $\sigma_i = \tau_\ell^*$. It immediately follows that $W(\mathcal{C}_{i-1} \cup \{\sigma_i\}) = K'_\ell \leq (1 - 1/k) \cdot K_0 = (1 - 1/k) \cdot W(\mathcal{C}_{i-1})$, concluding the proof. ◀

We are now ready to prove Theorem 6.

Proof of Theorem 6. We first note that if $r = 0$, then a solution with k trajectories can be computed, if one exists, as follows: Since $r = 0$, it follows that every level of P has at most k points. We construct the flow network instance $N_\delta(P, P)$, as in Section 2.1. It is immediate that the flow instance is feasible iff there exists a solution with k trajectories. We may thus assume that $r > 0$. Since the minimum distance in M is 1, it follows that $r \geq 1$. In a generic step $1 \leq i \leq L$, let τ_i denote the trajectory returned by the dynamic program of Lemma 16, which minimizes $W(\mathcal{C} \cup \{\tau_i\})$. By Lemma 17, if $\mathcal{I}$ admits a temporal (k, r, δ)-median-clustering, then there exists some trajectory σ_i such that $W(\mathcal{C}_{i-1} \cup \{\sigma_i\}) \leq (1 - 1/k) \cdot W(\mathcal{C}_{i-1})$. Thus $W(\mathcal{C}_i) = W(\mathcal{C}_{i-1} \cup \{\tau_i\}) \leq W(\mathcal{C}_{i-1} \cup \{\sigma_i\}) \leq (1 - 1/k) \cdot W(\mathcal{C}_{i-1}) \leq (1 - 1/k)^i \cdot W(\mathcal{C}_0)$. Since the diameter of M is Δ, we get $W(\mathcal{C}_0) \leq \Delta \sum_{i \in [t]} |P(i)| = \Delta n$. Setting $L = k \ln(n\Delta/\varepsilon) = O(k \log(n\Delta/\varepsilon))$, we obtain $W(\mathcal{C}_L) \leq (1 - 1/k)^L n\Delta \leq \varepsilon \leq \varepsilon r$. Thus $\max_{i \in [t]} \max\{0, \mathsf{cost}(i; \mathcal{C}_L) - r\} \leq \sum_{i=1}^{t} \max\{0, \mathsf{cost}(i; \mathcal{C}_L) - r\} \leq \varepsilon r$, which implies $\mathsf{rad}_1(\mathcal{C}_L) = \max_{i \in [t]} \mathsf{cost}(i; \mathcal{C}) \leq (1 + \varepsilon) r$. It follows that either $\mathcal{C}_L$ is a $(L, 1 + \varepsilon, 1)$-approximation, or $\mathcal{I}$ does not admit a (k, r, δ)-median-clustering. Finally, the running time follows by the fact that we perform L iterations of the main loop; each in time bounded by Lemma 16. ◀

3 Inapproximability and Conclusion

We now briefly state our inapproximability results. Due to space constraints we defer all proofs to the full version [9]. There, we show that it is NP-hard to obtain a $(1, \mathsf{poly}(n), \mathsf{poly}(n))$-approximation, complementing Theorem 1. Further, we show that the problem remains hard to approximate, even for an inexact number of clusters where the points are taken from a nice metric space. Specifically, we prove that $(1.005, 2 - \varepsilon, \mathsf{poly}(n))$-approximation is NP-hard for points sampled from 2-dimensional Euclidean space. We show that the $(\ln n, 1, 1)$-approximation (Theorem 2) is best possible by observing that $((1 - \varepsilon) \ln n, 2 - \varepsilon', \cdot)$-approximation is NP-hard, though the construction involves a somewhat unnatural metric space. Finally, we adapt the hardness results for $(1, \mathsf{poly}(n), \mathsf{poly}(n))$-approximation and $(1.005, 2 - \varepsilon, \mathsf{poly}(n))$-approximation to Temporal k-Median/Means.

Conclusion. Our results show that many instances of temporal clustering are hard to approximate. On the other hand, our polynomial time approximations show that sometimes if we allow approximations in terms of parameters like r/δ or the spread Δ, the approximation becomes tractable. We wish to better understand the boundary between these cases. Another direction comes from altering the model; an alternative formulation could allow centers from the ambient metric space. We plan to investigate this model in future research.

References

1 Mohammad Ali Abam and Mark de Berg. Kinetic spanners in rd. In *Proceedings of the Twenty-fifth Annual Symposium on Computational Geometry*, SCG'09, pages 43–50, New York, NY, USA, 2009. ACM. doi:10.1145/1542362.1542371.

2 Pankaj K. Agarwal, Leonidas J. Guibas, Herbert Edelsbrunner, Jeff Erickson, Michael Isard, Sariel Har-Peled, John Hershberger, Christian S. Jensen, Lydia E. Kavraki, Patrice Koehl, Ming C. Lin, Dinesh Manocha, Dimitris N. Metaxas, Brian Mirtich, David M. Mount, S. Muthukrishnan, Dinesh K. Pai, Elisha Sacks, Jack Snoeyink, Subhash Suri, and Ouri Wolfson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002. doi:10.1145/592642.592647.

3 Alfred V. Aho and David Lee. Efficient algorithms for constructing testing sets, covering paths, and minimum flows. *AT&T Bell Laboratories Tech. Memo*, 159, 1987.

4 David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

5 Vijay Arya, Naveen Garg, Rohit Khandekar, Adam Meyerson, Kamesh Munagala, and Vinayaka Pandit. Local search heuristics for k-median and facility location problems. *SIAM Journal on computing*, 33(3):544–562, 2004.

6 Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'97, pages 747–756, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=314161.314435.

7 Mihir Bellare, Oded Goldreich, and Madhu Sudan. Free bits, pcps, and nonapproximability—towards tight results. *SIAM J. Comput.*, 27(3):804–915, June 1998. doi:10.1137/S0097539796302531.

8 Sergio Cabello, Panos Giannopoulos, Christian Knauer, Dániel Marx, and Günter Rote. Geometric clustering: fixed-parameter tractability and lower bounds with respect to the dimension. *ACM Transactions on Algorithms (TALG)*, 7(4):43, 2011.

9 Tamal K. Dey, Alfred Rossi, and Anastasios Sidiropoulos. Temporal clustering. *CoRR*, abs/1704.05964, 2017. URL: http://arxiv.org/abs/1704.05964.

10 Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC'14, pages 624–633, New York, NY, USA, 2014. ACM. doi:10.1145/2591796.2591884.

11 Anne Driemel, Amer Krivošija, and Christian Sohler. Clustering time series under the Fréachet distance. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 766–785. SIAM, 2016.

12 Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

13 Edward W Forgy. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, 21:768–769, 1965.

14 Sorelle A. Friedler and David M. Mount. Approximation algorithm for the kinetic robust k-center problem. *Comput. Geom. Theory Appl.*, 43(6-7):572–586, August 2010. doi:10.1016/j.comgeo.2010.01.001.

15 Harold N. Gabow and Robert Endre Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989. doi:10.1137/0218069.

16 Jie Gao, Leonidas J. Guibas, and An Nguyen. Deformable spanners and applications. *Comput. Geom. Theory Appl.*, 35(1-2):2–19, August 2006. doi:10.1016/j.comgeo.2005.10.001.

17 Leonidas J. Guibas. Kinetic data structures: A state of the art report. In *Proceedings of the Third Workshop on the Algorithmic Foundations of Robotics on Robotics : The Algorithmic Perspective: The Algorithmic Perspective*, WAFR'98, pages 191–209, Natick, MA, USA, 1998. A. K. Peters, Ltd. URL: http://dl.acm.org/citation.cfm?id=298960.299007.

18 Sariel Har-Peled. Clustering motion. *Discrete & Computational Geometry*, 31(4):545–565, 2004.

19 Sariel Har-Peled and Akash Kushal. Smaller coresets for k-median and k-means clustering. *Discrete & Computational Geometry*, 37(1):3–19, 2007. doi:10.1007/s00454-006-1271-x.

20 Sariel Har-Peled and Soham Mazumdar. On coresets for k-means and k-median clustering. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 291–300. ACM, 2004.

21 Sariel Har-Peled and Bardia Sadri. How fast is the k-means method? *Algorithmica*, 41(3):185–202, 2005.

22 Teresa W Haynes, Stephen Hedetniemi, and Peter Slater. *Fundamentals of domination in graphs*. CRC Press, 1998.

23 Dorit S Hochbaum and David B Shmoys. A best possible heuristic for the k-center problem. *Mathematics of operations research*, 10(2):180–184, 1985.

24 Anil K Jain and Richard C Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

25 David S Johnson. Approximation algorithms for combinatorial problems. *Journal of computer and system sciences*, 9(3):256–278, 1974.

26 Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. A local search approximation algorithm for k-means clustering. In *Proceedings of the eighteenth annual symposium on Computational geometry*, pages 10–18. ACM, 2002.

27 Stavros G Kolliopoulos and Satish Rao. A nearly linear-time approximation scheme for the euclidean k-median problem. *SIAM Journal on Computing*, 37(3):757–782, 2007.

28 Madhukar R Korupolu, C Greg Plaxton, and Rajmohan Rajaraman. Analysis of a local search heuristic for facility location problems. *Journal of algorithms*, 37(1):146–188, 2000.

29 Stuart P Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

30 Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM (JACM)*, 41(5):960–981, 1994.

31 Mikkel Thorup. Quick k-median, k-center, and facility location for sparse graphs. In *Automata, Languages and Programming*, pages 249–260. Springer, 2001.

Pricing Social Goods*†

Alon Eden[1], Tomer Ezra[2], and Michal Feldman[3]

1 **Tel Aviv University, Israel**
alonarden@gmail.com
2 **Tel Aviv University, Israel**
tomer.ezra@gmail.com
3 **Tel Aviv University, Israel; and**
Microsoft Research, Israel
michal.feldman@cs.tau.ac.il

Abstract

Social goods are goods that grant value not only to their owners but also to the owners' surroundings, be it their families, friends or office mates. The benefit a non-owner derives from the good is affected by many factors, including the type of the good, its availability, and the social status of the non-owner. Depending on the magnitude of the benefit and on the price of the good, a potential buyer might stay away from purchasing the good, hoping to free ride on others' purchases. A revenue-maximizing seller who sells social goods must take these considerations into account when setting prices for the good. The literature on optimal pricing has advanced considerably over the last decade, but little is known about optimal pricing schemes for selling social goods. In this paper, we conduct a systematic study of revenue-maximizing pricing schemes for social goods: we introduce a Bayesian model for this scenario, and devise nearly-optimal pricing schemes for various types of externalities, both for simultaneous sales and for sequential sales.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases Public Goods, Posted Prices, Revenue Maximization, Externalities

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.35

1 Introduction

Many goods exhibit a positive externality not only on their owner, but also on other parties. For instance, a coffee machine purchased by an employee benefits all of her office mates, and essentially reduces the probability of another coffee machine to be purchased. Examples of these kinds of goods are abundant: A high-schooler who has many friends with cars that can drive him around might be less tempted to buy a new car. A reputable store might draw large customer traffic and benefit other stores in the shopping mall. Therefore, an aggressive advertising campaign carried out by such a store might reduce the likelihood of another store running a campaign in parallel. In all of these scenarios the externalities depend on the type of good, on the social status of the party with whom the good is shared, and on the set of parties who own the good. In the coffee machine example, the machine is typically used by all the individuals sharing the office space. In the shopping mall, some types of stores (*e.g.*, fast food restaurants) might benefit from any traffic in the shopping mall, whereas more specialized stores may benefit from ad campaigns that draw costumers interested in a similar

* A full version of the paper is available at https://arxiv.org/abs/1706.10009.
† This work was partially supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement number 337122.

kind of product (*e.g.*, `Staples` may attract costumers similar to those interested in `Office Depot` products). The benefit of a high school student depends on his social status and on the set of friends who own a car.

Because of the abundance of goods that exhibit externalities similar to the ones in the examples above, their study is of great applicability. We term these goods *social goods*. When selling social goods, a seller must take into account the types of buyers in the market and the benefit they derive from other sets of buyers purchasing the good. Our main goal is to study how to sell goods in a way that approximately maximizes the seller's revenue in the presence of externalities.

To study this problem, we consider a setting with a single type of good, of unlimited supply, and a set of n agents; each agent $i \in [n]$ has a non-negative valuation v_i for purchasing the good, drawn independently from a distribution F_i. We denote the product distribution by $\mathcal{F} = \times_{i\in[n]} F_i$. Unless stated otherwise, we assume the F_i's are regular.[1]

If an agent does not purchase the good, but the good is purchased by others, then this agent derives only a fraction of her value, depending on the set of agents and the type of externality the good exhibits on the agent. This type of externality is captured in our model by an *externality function* $x_i : 2^{[n]} \to [0,1]$, where $x_i(S)$ denotes the fraction of v_i an agent i derives when the good is purchased by the set of agents S. We assume that x_i is publicly known (as it captures the agent's externalities), monotonically non-decreasing and normalized; i.e., $x_i(\emptyset) = 0$, for every $T \subseteq S$, $x_i(T) \leq x_i(S)$, and $x_i(S) = 1$ whenever $i \in S$. We consider three structures of the function x_i, corresponding to three types of externalities of social goods.

(a) *Full externalities* (commonly known as "public goods"): in this scenario all agents derive their entire value if the good is purchased by any agent. Therefore, $x_i(S) = 1$ if and only if $S \neq \emptyset$. This model captures goods that are non-excludable, such as a coffee machine in a shared office. A special case of this scenario, where valuations are independently and identically distributed, has been studied in [10].

(b) *Status-based externalities*: in this scenario, agent i's "social status" is captured by some *discount factor* $w_i \in [0,1]$, which corresponds to the fraction of the value an agent i derives from a good when purchased by another party. That is,

$$x_i(S) = \begin{cases} 1 & i \in S, \\ w_i & i \notin S \text{ and } S \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

This model captures settings that exhibit asymmetry with respect to the benefit different agents derive from goods they do not own (*e.g.*, a fast food restaurant or a popular high-school student in the above examples).

(c) *Availability-based externalities*: in this scenario, the availability of a good increases as more agents purchase a good, and therefore, an agent derives a larger fraction of her value as more agents purchase a good. This is captured by the following externality function.

$$x_i(S) = \begin{cases} 1 & i \in S, \\ w(|S|) & i \notin S. \end{cases} \tag{2}$$

Here, $w : \{0, \ldots, n-1\} \to [0,1]$ is a monotonically non-decreasing function with $w(0) = 0$. Examples of such scenarios include objects that are often shared by neighbors (e.g., snow blowers, lawn mowers), office supplies, etc.

---

[1] This means that the virtual valuation function $\phi(v) = v - \frac{1-F(v)}{f(v)}$ is non-decreasing.

Notice that the full externalities scenario is a special case of both the social-status (where $w_i = 1$ for every i) and the availability (where $w(k) = 1$ for every $k > 0$) models.

Our focus is on posted-price mechanisms, which exhibit many desired properties: they are simple, distributed, straightforward, and strategyproof. Our goal is to maximize the revenue extracted by the seller. We distinguish between discriminatory and non-discriminatory prices. Naturally, using discriminatory prices can often lead to higher revenue for the seller [18, 15]. Price discrimination is commonly used in the US [12], but user studies reveal that many users believe that this practice is illegal, and consider these acts to be an invasion of privacy [5]. Therefore, offering non-discriminatory prices may be critical for maintaining the seller's reputation. We show scenarios in which setting the same price for all users produces (almost) as much revenue as engaging in price discrimination.

We consider two natural sale models: (a) a *simultaneous sale*, where the seller simultaneously sets take-it-or-leave-it prices for all agents, after which agents play a simultaneous Bayesian game, and each agent decides whether or not to buy at the price offered to her; and (b) a *sequential sale*, in which the agents arrive sequentially, and each one is offered a take-it-or-leave-it price upon arrival. In this case, the price and the agent's decision may depend on the set of agents that purchased the good before the arrival of the current agent. We distinguish between *adaptive* and *non-adaptive* pricing schemes, which differ in whether the price can depend upon the set of agents who purchased the good prior to the agent's arrival.

In both simultaneous and sequential sales, assuming that agent i is offered a take-it-or-leave-it price p_i, and that the good is eventually purchased by a set $S \subseteq [n]$ of agents, the utility of agent i is:

$$u_i(S, p_i) = \begin{cases} v_i - p_i & \text{if } i \in S, \\ v_i \cdot x_i(S) & \text{if } i \notin S. \end{cases} \tag{3}$$

As shown in Section 2, a set of prices induces equilibria of the game (multiple equilibria in the simultaneous model, and a single one in the sequential model). Every equilibrium is characterized by a set of *threshold* strategies for the agents, where an agent buys the good if and only if her value exceeds the threshold.

1.1 Our contribution

We provide results for the three aforementioned models. In this section, we provide informal statements of our results. The exact bounds we achieve are summarized in Table 1. Due to space limitation, some of the formal statements and proofs are deferred to the full version.

(a) Full externalities

Theorem (informal): There exist poly-time algorithms for computing pricing schemes for settings with full externalities that give a constant factor approximation to the optimal pricing scheme, for both simultaneous and sequential sales. Moreover, this result can be achieved using non-discriminatory prices, despite asymmetry among buyers.

To derive this result, we first analyze the equilibria in simultaneous and sequential models. We show a surprising equivalence between the revenue attainable in the best equilibrium at simultaneous and sequential sales, albeit induced by different prices. A corollary of this equivalence is that the optimal attainable revenue at a sequential sale does not depend on

the order of agents. Furthermore, we observe that in both simultaneous and sequential sales, the revenue attainable is upper bounded by the optimal revenue from selling a single *private* good (i.e., a good that grants value only to their owners)[2].

We proceed as follows. For simultaneous sales, we establish a method for transforming prices for the sale of a single private good *in expectation* into prices for selling public goods, which preserve the revenue up to a constant factor in every equilibrium. Since selling a single good in expectation yields at least as much revenue as selling a single good deterministically, this implies a near-optimal pricing scheme for simultaneous sales of public goods.

For sequential sales, we use the theory of prophet inequalities. Consider prices that induce thresholds that are equal to the prices that emerge from the prophet inequalities. We show that such prices obtain at least half of the revenue obtained from the prophet inequalities prices in the private good model. We use this connection to obtain a pricing scheme that gives 4-approximation to the revenue of the optimal sequential sale of public goods.

Finally, we show how to compute nearly-optimal *non-discriminatory* prices, even for asymmetric agents, in both the simultaneous and sequential models.

(b) Status-based externalities

Theorem (informal): There exist poly-time algorithms for computing pricing schemes for settings with status-based externalities that give a constant factor approximation to the optimal pricing scheme, for both simultaneous and sequential sales.[3]

For sequential sales, we devise a *non-adaptive* pricing scheme, while the benchmark is the optimal adaptive pricing scheme. To obtain this result, we first show that a seller who is restricted to set only two prices per agent can extract as much revenue as one who can present exponentially many prices. We then show that the optimal revenue in this simpler case can be decomposed into two components: a private component (monotonically decreasing in the agents' discount factors) and a public component (monotonically increasing in the discount factors). The private component can be approximated by simulating n private sales, setting thresholds equal to the monopoly prices. The public component can be approximated by similar techniques to the ones introduced for public goods. Therefore, the better of the two mechanisms extracts a constant fraction of the optimal revenue. A similar decomposition technique is established for the case of simultaneous sales. Our result for the sequential case is essentially a reduction: given prices that yield a c-approximation for the optimal sequential sale in the full externalities model, one can find prices that $(c+2)$-approximate the optimal sequential sale in the status-based externalities model.

(c) Availability-based externalities

Theorem (informal): There exists a poly-time algorithm for computing a pricing scheme for sequential sales with availability-based externalities, that gives a logarithmic factor approximation (with regard to the number of buyers) to the optimal pricing scheme.

In this case, both the pricing scheme and the benchmark set a pricing function for each agent, which depends on the number of agents who have purchased the good before the arrival of the agent. To obtain this result, we decompose the revenue into n components.

---

[2] A similar argument was used in [10] for the special case of simultaneous sales where valuations are identically distributed.

[3] We note that no non-discriminatory prices can achieve a constant approximation in this model. Indeed, the case of private digital goods is a special case of this model, with $w_i = 0$ for every i.

Table 1 Summary of our results. The columns correspond to sale models, whereas the rows correspond to externality types. The rows are further divided to sales using discriminatory and non-discriminatory prices. All the unreferenced results appear in the full version.

		Simultaneous		Sequential	
		disc.	non-disc.	disc.	non-disc.
Full (Public goods)	i.i.d.	$\geq 4/e$	4	–	4
	non i.i.d.	5.83 Thm. 3.5	$4e$	4	$4e$
Status-based		6.83	$\Omega(\log n)$	6 Cor. 4.4	$\Omega(\log n)$
availability-based		–	–	$O(\log n)$	–
network-based		$\Omega\left(n^{1-\epsilon}\right)$	–	$\Omega\left(n^{1-\epsilon}\right)$	–

Component $k = 1, \ldots, n$ is upper bounded by the optimal revenue obtainable by selling k identical private goods, scaled by $w(k) - w(k-1)$. We then partition the components into buckets, and compute prices based on the sequential posted pricing scheme developed by Chawla et al. [8] for selling private goods.

General externalities. Given the near-optimal pricing schemes above, one may be tempted to infer that every social goods scenario is amenable to a near-optimal pricing scheme. We complement our positive results with the following hardness result, refuting this hope. We consider a natural family of social goods proposed by Feldman et al. [10]: network-based externalities. In this model, externalities are represented by a graph, and an agent derives her entire value when a neighboring agent buys a good. We show that there is no poly-time algorithm to compute prices that give a non-trivial approximation to the optimal posted-price mechanism. This negative result holds for both the simultaneous and sequential models. We show that even in very restricted cases (i.e., where agents' valuations are independently and uniformly distributed on $[0, 1]$ in the simultaneous case, and agents' valuations are fixed in the sequential case), it is NP-hard to find prices that approximate the optimal posted-price mechanism to within a factor of $n^{1-\epsilon}$. A $\Theta(n)$ approximation can be trivially achieved by offering the good only to the agent maximizing the monopolist revenue. We note that this negative result rules out other natural externality structures.[4]

Irregular distributions. Although our results are stated and proved for regular distributions, some of our results extend to irregular distributions. Namely, we establish near optimal pricing schemes for sequential and simultaneous sales under full externalities and status-based externalities. The results of non-discriminatory prices do not extend to irregular distributions since the anonymous pricing devised in [4] do not perform well for irregular distributions. (there exist irregular distributions that there is no anonymous price that give constant approximation.)

Organization. Due to space limitations, some of the results and proofs are deferred to the full version[5]. In the extended abstract, we state two of our main results along with their proof ideas. The provided proofs give the flavor of the techniques that seem to be useful in studying pricing mechanisms for social goods.

[4] Some examples include: (a) for every pair of agents i, j, agent i can borrow the good from agent j with some probability w_{ij}. Thus, $x_i(S) = 1 - \prod_{j \in S, j \neq i}(1 - w_{ij})$; and (b) for every pair of agents i, j, $x_i(S) = \max_{j \in S, j \neq i} w_{ij}$.

[5] The full version appears in `https://arxiv.org/abs/1706.10009`.

The extended abstract is organized as follows. In Section 2 we describe the simultaneous and sequential sale models. In Section 3 we study the case of full externalities: In Section 3.1 we establish useful properties of equilibria, and in Section 3.2 we devise a near-optimal pricing scheme for simultaneous sales. The following are deferred to the full version:

(a) a near-optimal pricing scheme for sequential sales,
(b) a non-discriminatory pricing scheme, and
(c) lower bounds for simultaneous sales.

In Section 4 we present our near-optimal pricing scheme for sequential sales under status-based externalities. The near-optimal pricing scheme for simultaneous sales is deferred to the full version. The case of availability-based externalities is deferred to the full version in its entirety. The same is true for the hardness results for general externalities, as well as a discussion about the irregular case.

1.2 Related work

The most famous and well studied instance of social goods is public goods, when all agents derive their full value whenever a good is purchased. The study of public goods was initiated by Samuelson [19], who observed that private provisioning of public goods is not necessarily efficient; see also [17] for an overview.

The closest work to ours is that of Feldman et al. [10]. For their positive results, they consider a special case of our full externalities model — in their model agents arrive simultaneously with valuations that are drawn independently and identically from a known distribution. Our work extends this work in several dimensions. First, we consider more realistic forms of externalities that go beyond public goods. Second, we consider settings where agent valuations are drawn from non-identical distributions. Third, we provide results for settings where agents arrive either sequentially or simultaneously. Finally, some of our results extend to irregular distributions.

A line of work similar in flavor to ours, yet inherently different, is that of revenue maximization in the presence of positive externalities [1, 11, 13, 2, 6]. In this line of work, an agent's value for the good increases as more agents purchase the goods, but only if the agent purchased the good as well. Therefore, an agent is more likely to purchase the good as more agents purchase it. This is in stark contrast to our setting, where agents are less inclined to buy a good as more agents do.

Finally, there is a rich body of literature on the design of posted price mechanisms for the sale of private goods (where agents do not derive value from goods they do not own). See Chapter 4 in [14] for a textbook treatment. A sample of the work can be found in [8, 4, 16, 9, 7]. An overview of some results that are directly referred to in this work is given in the full version.

2 Models and preliminaries

Simultaneous sales model. We view a simultaneous sale game as the following two-stage game. First, the seller posts a price vector $\mathbf{p} = (p_1, \ldots, p_n)$ to the agents (agent i is offered to purchase an item at price p_i). Subsequently, the agents play a simultaneous Bayesian game. In this model, we assume that the probability distribution of every agent is atomless.[6]

[6] Meaning that for every q there exists p for which $F_i(p) = q$.

Agents wish to maximize their expected utility. Given a price p_i, agent i buys the good if her expected utility from buying, $v_i - p_i$, exceeds the utility from not buying, $v_i \cdot \mathrm{E}_{S \not\ni i}[x_i(S)]$ (where $\mathrm{E}_{S \not\ni i}$ is shorthand for $\mathrm{E}_{S: i \notin S}$). Therefore, an agent buys if and only if $v_i \geq \frac{p_i}{1-\mathrm{E}_{S \not\ni i}[x_i(S)]} =: T_i$. The strategy of every agent i is therefore defined by a threshold T_i. Denote by $\mathbf{T} = (T_1, \ldots, T_n)$ a strategy profile, given by a vector of thresholds. A strategy profile $\mathbf{T}$ induces a probability distribution over the set S of agents that purchase the good; denote this distribution by $\mu_{\mathbf{T}}$, and the distribution $\mu_{\mathbf{T}}$ conditioned on i not being in the set of purchasing agents by $\mu_{\mathbf{T}}^{-i}$. A Nash equilibrium is characterized by a threshold vector $\mathbf{T}$ such that:

$$T_i = \frac{p_i}{1 - \mathrm{E}_{S \sim \mu_{\mathbf{T}}^{-i}}[x_i(S)]} \quad \forall i \in [n]. \tag{4}$$

The following theorem establishes the existence of Nash equilibria via a fixed point argument.

▶ **Theorem 2.1.** *In the simultaneous model, for any set of externality functions $\{x_i\}_{i\in[n]}$, for any set of atomless distributions $\mathcal{F}$, and for any price vector $\mathbf{p}$, there exists an equilibrium $\mathbf{T}$.*

One of the challenges in our model stems from the fact that a single price vector may induce multiple equilibria. Consider the simple setting of a single public good and two agents, Alice and Bob, where F_{Alice} and F_{Bob} are both uniform on $[0, 1]$, and the seller sets a non discriminatory price of $1/2$. Applying the equilibrium condition in Eq. (4), we get that every tuple $(T_{\text{Alice}}, T_{\text{Bob}}) \in [0, 1]^2$ satisfying $T_{\text{Alice}} \cdot T_{\text{Bob}} = 1/2$ forms an equilibrium strategy.[7] Therefore, in this case, there is a continuum of equilibria. It is not hard to see, however, that a set of thresholds $\mathbf{T}$ can be the consequence of only a single price vector, which can be derived via Eq. (4). This is cast in the following observation:

▶ **Observation 2.2.** *In the simultaneous model, a given price vector can induce multiple equilibria, but any given equilibrium $\mathbf{T}$ can be induced by a single price vector $\mathbf{p}$.*

Let $Eq(\mathcal{F}, \mathbf{p})$ denote the set of equilibria induced by a price vector $\mathbf{p}$, given a product distribution $\mathcal{F}$. For a given price vector $\mathbf{p}$ and an equilibrium $\mathbf{T} \in Eq(\mathcal{F}, \mathbf{p})$, let $\mathcal{R}_{\text{sim}}(\mathcal{F}, \mathbf{p}, \mathbf{T}) = \sum_i p_i \cdot (1 - F_i(T_i))$ denote the seller's expected revenue. Given a price vector $\mathbf{p}$, we define

$$\overline{\mathcal{R}}_{\text{sim}}(\mathcal{F}, \mathbf{p}) = \max_{\mathbf{T} \in Eq(\mathcal{F}, \mathbf{p})} \mathcal{R}_{\text{sim}}(\mathcal{F}, \mathbf{p}, \mathbf{T}) \quad \text{and} \quad \underline{\mathcal{R}}_{\text{sim}}(\mathcal{F}, \mathbf{p}) = \min_{\mathbf{T} \in Eq(\mathcal{F}, \mathbf{p})} \mathcal{R}_{\text{sim}}(\mathcal{F}, \mathbf{p}, \mathbf{T})$$

to be the revenue obtained in the respective best and worst equilibrium induced by $\mathbf{p}$. We refer to these revenues as the *optimistic* and *pessimistic* revenues, respectively.

The strongest approximation results one can hope for are ones that consider the pessimistic revenue obtained by our pricing scheme against an optimistic benchmark. This is exactly the approach we take. In particular, our benchmark is the revenue obtained by the best pricing, assuming the best equilibrium induced by every pricing. We denote the benchmark by $\mathcal{R}^*_{\text{sim}}(\mathcal{F}) = \max_{\mathbf{p}^*} \overline{\mathcal{R}}_{\text{sim}}(\mathcal{F}, \mathbf{p}^*)$. The performance of a price vector $\mathbf{p}$ is measured by the worst equilibrium induced by $\mathbf{p}$; i.e., $\underline{\mathcal{R}}_{\text{sim}}(\mathcal{F}, \mathbf{p})$. Our goal is to calculate a price vector $\mathbf{p}$ that minimizes the ratio between the former and the latter expressions.

[7] For a comprehensive discussion regarding the equilibrium condition in the public goods model, see Eq.(5) in Section 3.

Sequential sales model. In the sequential sales model, n agents arrive one by one according to an order $\sigma : [n] \to [n]$, where agent i is the $\sigma(i)$th agent to arrive. For ease of notation, we assume that agent i is the ith agent to arrive, unless explicitly stated otherwise. In sequential sales, the price set by the seller for agent i can depend on the set of agents who have purchased the good prior to agent i's arrival. Thus, it can be viewed as a function $p_i : 2^{[i-1]} \to \mathbb{R}^+$.[8] The subgame perfect equilibrium in this auction is unique and can be found by a (possibly exponential) backward induction. An agent who receives a price buys if and only if her utility from buying exceeds her expected derived value from not buying conditioned on the set of agents that purchased the good prior to her arrival. Of course, this might impose a different threshold for every scenario which might lead to an exponential strategy space for the agents and an exponential time to compute each threshold in the strategy of an agent. As we discuss in the following sections, we devise pricing schemes in which the seller has a simple nearly optimal pricing scheme which leads to a simple strategy space and a poly-time threshold computation.

3 Pricing goods with full externalities (public goods)

3.1 Equilibrium and revenue equivalence

In this section we focus on the case where all agents derive their entire value from a good if purchased by any agent. We first characterize the equilibrium condition for a simultaneous sale. Given an equilibrium $\mathbf{T} = (T_1, \ldots, T_n)$, the expected value agent i derives from other agents is $\mathrm{E}_{S \sim \mu_{\mathbf{T}}^{-i}}[x_i(S)] = 1 * \Pr[\text{some agent } j \neq i \text{ buys}] = 1 - \Pr[\text{no agent } j \neq i \text{ buys}] = 1 - \prod_{j \neq i} F_j(T_j)$. Plugging this expression into Eq. (4) yields the following equilibrium condition:

$$T_i = \frac{p_i}{\prod_{j \neq i} F_j(T_j)} \quad \text{for all } i. \tag{5}$$

For a given price vector $\mathbf{p}$ and an equilibrium $\mathbf{T} \in Eq(\mathcal{F}, \mathbf{p})$, the expected revenue is

$$\mathcal{R}_{\text{sim}}(\mathcal{F}, \mathbf{p}, \mathbf{T}) = \sum_i p_i \left(1 - F_i(T_i)\right) \overset{(5)}{=} \sum_i T_i \cdot \left(\prod_{j \neq i} F_j(T_j)\right) \cdot \left(1 - F_i(T_i)\right). \tag{6}$$

We turn to describe the equilibrium in the sequential sales model. In this case, whenever an agent buys an item, no subsequent agent will ever buy an item. Therefore, we can assume without loss of generality that the seller sets a single price per agent. Let $\mathbf{p} = (p_1, \ldots, p_n)$ denote the vector of offered prices.

We now show how to compute the unique subgame perfect equilibrium of the game. When the last agent (agent n) is offered a price, her best strategy is to buy if her value exceeds the price; i.e., $T_n = p_n$. When agent $i = n-1, \ldots, 1$ is offered a price, she faces the following tradeoff: if she buys, her utility is $v_i - p_i$. If she does not buy, her utility is $v_i \left(1 - \prod_{j>i} \Pr[j \text{ does not buy}]\right) = v_i \left(1 - \prod_{j>i} F_j(T_j)\right)$. Consequently, the unique

[8] Indeed, there are cases where the seller can gain higher revenue by setting such prices (an explicit example for availability-based externalities is given in the full version).

equilibrium $\mathbf{T}$ is given by[9][10]

$$T_i = \frac{p_i}{\prod_{j>i} F_j(T_j)} \quad \forall i \in [n]. \tag{7}$$

Given a product distribution $\mathcal{F}$, a price vector $\mathbf{p}$, and an arrival order σ, let $\mathbf{T}_{\mathcal{F}}(\sigma, \mathbf{p})$ be the function that returns the unique equilibrium. Since every price vector $\mathbf{p}$ defines a unique strategy vector $\mathbf{T}$, the expected revenue from agent i is also uniquely defined, and can be calculated by

$$\begin{aligned}\prod_{j<i} \Pr\left[j \text{ does not buy}\right] \cdot p_i \cdot (1 - F_i(T_i)) &\overset{(7)}{=} \left(\prod_{j<i} F_j(T_j)\right) \cdot T_i \cdot \left(\prod_{j>i} F_j(T_j)\right) \cdot (1 - F_i(T_i)) \\ &= T_i \cdot \left(\prod_{j\neq i} F_j(T_j)\right) \cdot (1 - F_i(T_i)).\end{aligned}$$

Therefore, the expected revenue from all agents can be written as

$$\mathcal{R}_{\text{seq}}(\mathcal{F}, \sigma, \mathbf{p}, \mathbf{T} = \mathbf{T}_{\mathcal{F}}(\sigma, \mathbf{p})) = \sum_i T_i \cdot \left(\prod_{j\neq i} F_j(T_j)\right) \cdot (1 - F_i(T_i)). \tag{8}$$

Given an arrival order σ, let $\mathcal{R}^*_{\text{seq}}(\mathcal{F}, \sigma) = \max_{\mathbf{p}} \mathcal{R}_{\text{seq}}(\mathcal{F}, \sigma, \mathbf{p}, \mathbf{T} = \mathbf{T}_{\mathcal{F}}(\sigma, \mathbf{p}))$ denote the highest revenue a seller can obtain. We note that given a threshold vector $\mathbf{T}$ and an arrival order σ, there is also a unique price vector that produces this threshold vector $\mathbf{T}$, which can be calculated by (7), thus $\mathbf{T}_{\mathcal{F}}(\sigma, \cdot)$ is a bijection. This is cast in the following observation.

▶ **Observation 3.1.** *Fix an arrival order. An equilibrium strategy vector* $\mathbf{T}$ *is uniquely determined by a price vector* $\mathbf{p}$*, and a price vector* $\mathbf{p}$ *is uniquely determined by a strategy vector* $\mathbf{T}$*.*

Theorem 3.2 establishes revenue equivalence in simultaneous and sequential sales.

▶ **Theorem 3.2.** *For every product distribution* $\mathcal{F}$ *and for every order of arrival* σ *in the sequential model, we have that* $\mathcal{R}^*_{\text{seq}}(\mathcal{F}, \sigma) = \mathcal{R}^*_{\text{sim}}(\mathcal{F})$*.*

It immediately follows that the optimal revenue is independent of the arrival order.

▶ **Corollary 3.3.** *For every two arrival orders* σ, σ'*,* $\mathcal{R}^*_{\text{seq}}(\mathcal{F}, \sigma) = \mathcal{R}^*_{\text{seq}}(\mathcal{F}, \sigma')$*.*

In the sequel, we use $\mathcal{R}^*_{\text{seq}}(\mathcal{F})$ to denote the optimal revenue in the sequential model.

We next draw a connection between selling public goods and selling a single private good. This connection is later used in proving approximation results for mechanisms for the sale of public goods. Let $Myer(\mathcal{F})$ denotes the optimal revenue a seller can obtain by selling a single private good to a set of agents drawn from $\mathcal{F}$ (*i.e.*, the revenue obtained by Myerson's optimal auction). Using similar arguments to ones used in [10], we have the following:

▶ **Lemma 3.4.** *For every product distribution* $\mathcal{F}$*,* $\mathcal{R}^*_{\text{seq}}(\mathcal{F}) \leq Myer(\mathcal{F})$ *(and therefore,* $\mathcal{R}^*_{\text{sim}}(\mathcal{F}) \leq Myer(\mathcal{F})$ *by Theorem 3.2).*

[9] Unlike the simultaneous model, an equilibrium exists for non atomless distributions, whenever tie-breaking is done consistently by agents. That is, agents always take the same action when their value is equal to their threshold.

[10] For n, we let $\prod_{j>n} F_j(T_j) = 1$.

3.2 Near optimal simultaneous sale

In our construction, we use the *ex-ante relaxation* (EAR) [3, 4] for selling a private good. The EAR relaxes the feasibility constraint, so that instead of selling at most one item *ex post*, this constraint holds only in expectation. Since the feasible region increases, the revenue of an optimal mechanism for this case can only be higher than Myerson's optimal mechanism. Combined with Lemma 3.4, it suffices to provide a pricing scheme for our setting that approximates the revenue of the EAR. As it turns out, when agents' values are drawn from regular distributions, the optimal mechanism for the ex-ante setting is a posted price mechanism. These prices can be computed in polynomial time by a convex programming formulation [14].

We use these prices to determine prices for the sale of public goods. To do so, we partition the agents into *valuable* and *non-valuable* agents, based on their contribution to the revenue of the EAR. All the revenue obtained in our pricing scheme comes from the valuable agents. Their prices are set so that if there exists a valuable agent that buys with low probability, the equilibrium condition guarantees that other agents buy with a sufficiently high probability.

▶ **Theorem 3.5.** *For social goods with full externalities and for any regular product distribution $\mathcal{F}$, there exists a poly-time algorithm that computes prices $\mathbf{p}$ for which $\underline{\mathcal{R}}_{\text{sim}}(\mathcal{F}, \mathbf{p}) \geq \mathcal{R}^*_{\text{sim}}(\mathcal{F})/5.83$.*

Proof. Let $\hat{p} = (\hat{p}_1, \ldots, \hat{p}_n)$ be the posted prices that maximize the revenue in the EAR, and let $\mathcal{R} = \sum_i \hat{p}_i(1 - F_i(\hat{p}_i))$ be the optimal revenue of the EAR. As mentioned above, $\mathcal{R} \geq Myer(\mathcal{F})$. Let $c_1, c_2 > 1$ be two parameters, to be determined later. We partition the agents into two groups as follows. Let $B = \{i \in [n] : \hat{p}_i \geq \mathcal{R}/c_1\}$ and $S = [n] \setminus B$. For every agent i we set

$$p_i = \begin{cases} \hat{p}_i/c_2 & i \in B \\ \infty & i \in S \end{cases}.$$

The revenue from the agents in S in the optimal EAR mechanism is bounded by $\sum_{i \in S} \hat{p}_i \cdot \Pr[i \text{ buys}] \leq \frac{\mathcal{R}}{c_1} \sum_{i \in S}(1 - F_i(\hat{p}_i)) \leq \frac{\mathcal{R}}{c_1}$, where the last inequality stems from the fact that the EAR sells at most 1 item in expectation. Therefore, the revenue extracted from agents in B in the EAR is

$$\sum_{i \in B} \hat{p}_i \cdot (1 - F_i(\hat{p}_i)) \geq \mathcal{R} - \frac{\mathcal{R}}{c_1} = (1 - 1/c_1)\,\mathcal{R}. \tag{9}$$

Let $\mathbf{T}$ be an equilibrium induced by the price vector $\mathbf{p} = (p_1, \ldots, p_n)$. We consider two cases:

Case 1: $T_i \leq \hat{p}_i$ for every $i \in B$. In this case,

$$\begin{aligned} \mathcal{R}_{\text{sim}}(\mathcal{F}, \mathbf{p}, \mathbf{T}) &= \sum_i p_i \cdot (1 - F_i(T_i)) = \sum_{i \in B} p_i \cdot (1 - F_i(T_i)) \\ &\geq \sum_{i \in B} \frac{\hat{p}_i}{c_2} \cdot (1 - F_i(\hat{p}_i)) \overset{(9)}{\geq} \left(\frac{1 - 1/c_1}{c_2}\right) \mathcal{R}, \end{aligned}$$

where the first inequality follows from case 1 and the monotonicity of F_i.

Case 2: There exists $i \in B$ such that $T_i > \hat{p}_i$. For such an agent i,

$$\frac{\hat{p}_i/c_2}{\prod_{j\neq i} F_j(T_j)} = \frac{p_i}{\prod_{j\neq i} F_j(T_j)} \overset{(5)}{=} T_i > \hat{p}_i \quad \Rightarrow \quad \prod_j F_j(T_j) \leq \prod_{j\neq i} F_j(T_j) \leq \frac{1}{c_2}. \tag{10}$$

Let $p_{\min} = \min_i p_i$. The expected revenue in this case is at least

$$p_{\min} \cdot \Pr[\text{at least one agent buys}] \quad \geq \quad \frac{\mathcal{R}}{c_1 c_2}\left(1 - \prod_j F_j(T_j)\right) \overset{(10)}{\geq} \left(\frac{1 - 1/c_2}{c_1 c_2}\right)\mathcal{R},$$

where the first inequality follows from the fact that all prices are at least $\frac{\mathcal{R}}{c_1 c_2}$.

Therefore, we get an approximation factor of $\min\left\{\left(\frac{1-1/c_1}{c_2}\right), \left(\frac{1-1/c_2}{c_1 c_2}\right)\right\}$. Setting $c_1 = \sqrt{2}$ and $c_2 = 1 + \frac{1}{\sqrt{2}}$ optimizes the approximation ratio and gives revenue of at least a $\frac{1}{3+2\sqrt{2}}$ fraction of $\mathcal{R}$. Since $\mathcal{R} \geq Myer(\mathcal{F}) \geq \mathcal{R}^*_{\text{sim}}(\mathcal{F})$ (by Lemma 3.4), we get that $\underline{\mathcal{R}}_{\text{sim}}(\mathcal{F}, \mathbf{p}) \geq \frac{\mathcal{R}^*_{\text{sim}}(\mathcal{F})}{3+2\sqrt{2}} \approx \frac{\mathcal{R}^*_{\text{sim}}(\mathcal{F})}{5.83}$. ◀

▶ Remark. An approximation ratio of 8 is given in [10] for the special case of i.i.d. distributions. The last theorem improves the approximation ratio to 5.83 even for the more general case of non-identical distributions. Moreover, in the full version we give a non-discriminatory pricing that gives 4 approximation for the case of identical distributions. We also show that no pricing scheme can give better approximation than $4/e$, even for identical distributions.

4 Near optimal sequential sale under status-based externalities

Recall that in this setting, every agent is associated with a discount factor $w_i \in [0, 1]$, Let $\mathbf{w} = (w_1, \ldots, w_n)$. We devise a non-adaptive pricing scheme (*i.e.*, where an agent's price does not depend on the previous purchases) that approximates the revenue of the optimal adaptive pricing scheme.[11] In our scheme, every agent is assigned with a single price.

Let $\mathbf{p}^0$ and $\mathbf{p}^{>0}$ be the price vectors posted by the seller who uses two price vectors, where p^0_i (resp., $p^{>0}_i$) is the price offered to agent i when no agent (resp., at least one agent) has purchased a good prior to i's arrival. Let $\mathbf{p} = (\mathbf{p}^0, \mathbf{p}^{>0})$. In the full version, we show that we it is without loss of generality to restrict attention to two price vectors.

In contrast to the full externalities settings, agent i may have two different thresholds in the equilibrium — one for the case where no agent bought a good before she arrives, denoted by T^0_i, and one for the case where at least one agent buys the good, denoted by $T^{>0}_i$. For every agent i, if some agent bought the good before she arrived, she faces the following trade-off — if she buys the good, her utility is $v_i - p^{>0}_i$; otherwise, her utility is $w_i \cdot v_i$. Therefore, the threshold satisfies the following equation:

$$T^{>0}_i - p^{>0}_i = w_i \cdot T^{>0}_i \Rightarrow p^{>0}_i = (1 - w_i) \cdot T^{>0}_i. \tag{11}$$

If no agent bought the good before before agent i arrived, then[12]

$$T^0_i - p^0_i = w_i \cdot T^0_i \cdot \Pr[\text{Agent } j > i \text{ buys a good}] = w_i \cdot T^0_i \cdot \left(1 - \prod_{j>i} F_j(T^0_j)\right)$$

$$\Rightarrow p^0_i = (1 - w_i) \cdot T^0_i + w_i \cdot T^0_i \cdot \prod_{j>i} F_j(T^0_j). \tag{12}$$

[11] Which sets a price for the current agent depending on the set of agents that purchased the good prior her arrival.

[12] For the case of $i = n$, the RHS product is naturally defined to be 1, and therefore $T^0_n = p^0_n$.

For every agent i and pricing $\mathbf{p} = (\mathbf{p}^0, \mathbf{p}^{>0})$, let $q_i^0 = q_i^0(\mathbf{p})$ (resp., $q_i^{>0} = q_i^{>0}(\mathbf{p})$) denote the probability that no agent (resp., at least one agent) has bought a good before agent i arrived. The revenue can now be written as

$$\begin{aligned}
\mathcal{R}(\mathbf{p}) &= \sum_i \left(q_i^0 \cdot p_i^0 \cdot (1 - F_i(T_i^0)) + q_i^{>0} \cdot p_i^{>0} \cdot (1 - F_i(T_i^{>0}))\right) \\
&\overset{(12)}{=} \sum_i q_i^0 \cdot \left((1-w_i) \cdot T_i^0 + w_i \cdot T_i^0 \cdot \prod_{j>i} F_j(T_j^0)\right) \cdot (1 - F_i(T_i^0)) \\
&\quad + \sum_i q_i^{>0} \cdot p_i^{>0} \cdot (1 - F_i(T_i^{>0})) \\
&\overset{(11)}{=} \sum_i q_i^0 \cdot (1-w_i) \cdot T_i^0 \cdot (1 - F_i(T_i^0)) + \sum_i q_i^0 \cdot w_i \cdot T_i^0 \cdot \left(\prod_{j>i} F_j(T_j^0)\right) \cdot (1 - F_i(T_i^0)) \\
&\quad + \sum_i q_i^{>0} \cdot T_i^{>0} \cdot (1-w_i) \cdot (1 - F_i(T_i^{>0})).
\end{aligned}$$

By removing factors smaller than 1 ($q_i^0, q_i^{>0}, w_i$) in the last expression, we get

$$\begin{aligned}
\mathcal{R}(\mathbf{p}) &\leq \sum_i (1-w_i) \cdot T_i^0 \cdot (1 - F_i(T_i^0)) + \sum_i \left(\prod_{j<i} F_j(T_j^0)\right) \cdot T_i^0 \cdot \left(\prod_{j>i} F_j(T_j^0)\right) \cdot (1 - F_i(T_i^0)) \\
&\quad + \sum_i T_i^{>0} \cdot (1-w_i) \cdot (1 - F_i(T_i^{>0})) \\
&= \sum_i (1-w_i) \cdot T_i^0 \cdot (1 - F_i(T_i^0)) + \sum_i (1-w_i) \cdot T_i^{>0} \cdot (1 - F_i(T_i^{>0})) \\
&\quad + \sum_i T_i^0 \cdot \left(\prod_{j \neq i} F_j(T_j^0)\right) \cdot (1 - F_i(T_i^0)).
\end{aligned} \tag{13}$$

Given a thresholds vector $\mathbf{T} = (T_1, T_2, \ldots, T_n)$, we define $\mathcal{R}_1(\mathbf{T}, \mathbf{w}) = \sum_i (1-w_i) \cdot T_i \cdot (1 - F_i(T_i))$ and $\mathcal{R}_2(\mathbf{T}) = \sum_i T_i \cdot \left(\prod_{j \neq i} F_j(T_j)\right) \cdot (1 - F_i(T_i))$. It follows from Eq. (13) that

$$\max_{\mathbf{p}} \mathcal{R}(\mathbf{p}) \leq 2 \max_{\mathbf{T}} \mathcal{R}_1(\mathbf{T}, \mathbf{w}) + \max_{\mathbf{T}} \mathcal{R}_2(\mathbf{T}). \tag{14}$$

That is, the RHS sum in Eq. (14) is an upper bound on the optimal revenue that can be obtained. $\mathcal{R}_1(\mathbf{T}, \mathbf{w})$ can be viewed as the private component of the revenue, which becomes more significant as w_i's get smaller, while $\mathcal{R}_2(\mathbf{T})$ can be viewed as the public component, which becomes more significant as w_i's grow. Notice that $\max_{\mathbf{T}} \mathcal{R}_2(\mathbf{T})$ is exactly $\mathcal{R}^*_{\text{seq}}(\mathcal{F})$, where $\mathcal{R}^*_{\text{seq}}(\mathcal{F})$ is the optimal posted prices revenue in a sequential sale in the full externalities model, as defined in Section 3.

The following lemmas (4.1 and 4.2) show that it is possible to find prices that approximate $\max_{\mathbf{T}} \mathcal{R}_1(\mathbf{T}, \mathbf{w})$ and prices that approximate $\max_{\mathbf{T}} \mathcal{R}_2(\mathbf{T})$. In fact, they show a stronger result, namely that for each of the terms in the sum, there exists a single price vector $\mathbf{p} = \mathbf{p}^0 = \mathbf{p}^{>0}$ that approximates it.

▶ **Lemma 4.1.** *There exists a poly-time algorithm for computing prices* $\mathbf{p}$ *such that* $\mathcal{R}(\mathbf{p}) \geq \max_{\mathbf{T}} \mathcal{R}_1(\mathbf{T}, \mathbf{w})$.

The following allows us to reduce the problem of finding "good" prices in the status-based externalities model to finding "good" prices in the full externalities model.

▶ **Lemma 4.2.** *Given prices* $\mathbf{p}'$*, there exist poly-time computable prices* $\mathbf{p}$ *such that* $\mathcal{R}(\mathbf{p}) \geq \mathcal{R}_{\text{seq}}(\mathcal{F}, \mathbf{p}')$.

We now present the main result of this section:

▶ **Theorem 4.3.** *Given a c-approximation pricing for sequential sales in the full externalities model, there exists a poly-time computable pricing that guarantees a $(c+2)$-approximation for the optimal sequential sales in the model of status-based externalities.*

Proof. Since $\max_{\mathbf{T}} \mathcal{R}_2(\mathbf{T}) = \mathcal{R}^*_{\text{seq}}(\mathcal{F})$, if one can find prices that c-approximate the optimal prices in the full externalities model, by Lemma 4.2, one can compute prices that c-approximate $\max_{\mathbf{T}} \mathcal{R}_2(\mathbf{T})$ in the status-based externalities model.

Let $\mathbf{p}_1$ and $\mathbf{p}_2$ be the sets of prices for which $\mathcal{R}(\mathbf{p}_1) \geq \max_{\mathbf{T}} \mathcal{R}_1(\mathbf{T}, \mathbf{w})$ and $c \cdot \mathcal{R}(\mathbf{p}_2) \geq \max_{\mathbf{T}} \mathcal{R}_2(\mathbf{T})$, respectively. These prices can be computed in poly time by Lemmas 4.1 and 4.2. We have that

$$\begin{aligned} \max_{\mathbf{p}} \mathcal{R}(\mathbf{p}) &\overset{(14)}{\leq} 2 \max_{\mathbf{T}} \mathcal{R}_1(\mathbf{T}, \mathbf{w}) + \max_{\mathbf{T}} \mathcal{R}_2(\mathbf{T}) \\ &\leq 2 \cdot \mathcal{R}(\mathbf{p}_1) + c \cdot \mathcal{R}(\mathbf{p}_2) \\ &\leq (c+2) \cdot \max\{\mathcal{R}(\mathbf{p}_1), \mathcal{R}(\mathbf{p}_2)\}. \end{aligned}$$ ◀

The following corollary follows from Theorem 4.3 and by the existence of a 4-approximation pricing for sequential sales in the full externalities model, as shown in the full version.

▶ **Corollary 4.4.** *For goods that exhibit status-based externalities, there exists a poly-time algorithm for computing prices that give a 6-approximation to the optimal pricing scheme.*

References

1. Nima AhmadiPourAnari, Shayan Ehsani, Mohammad Ghodsi, Nima Haghpanah, Nicole Immorlica, Hamid Mahini, and Vahab Mirrokni. Equilibrium pricing with positive externalities. *Theoretical Computer Science*, 476:1–15, 2013.
2. Hessameddin Akhlaghpour, Mohammad Ghodsi, Nima Haghpanah, Vahab S Mirrokni, Hamid Mahini, and Afshin Nikzad. Optimal iterative pricing over social networks. In *International Workshop on Internet and Network Economics*, pages 415–423. Springer, 2010.
3. Saeed Alaei. Bayesian combinatorial auctions: Expanding single buyer mechanisms to many buyers. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 512–521, 2011. doi:10.1109/FOCS.2011.90.
4. Saeed Alaei, Jason D. Hartline, Rad Niazadeh, Emmanouil Pountourakis, and Yang Yuan. Optimal auctions vs. anonymous pricing. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 1446–1463, 2015. doi:10.1109/FOCS.2015.92.
5. Ryan Calo. Digital market manipulation. *Geo. Wash. L. Rev.*, 82:995, 2013.
6. Ozan Candogan, Kostas Bimpikis, and Asuman Ozdaglar. Optimal pricing in the presence of local network effects. In *International Workshop on Internet and Network Economics*, pages 118–132. Springer, 2010.
7. Shuchi Chawla, Jason D. Hartline, and Robert Kleinberg. Algorithmic pricing via virtual valuations. In *Proceedings of the 8th ACM conference on Electronic commerce*, pages 243–251. ACM, 2007.
8. Shuchi Chawla, Jason D. Hartline, David L. Malec, and Balasubramanian Sivan. Multi-parameter mechanism design and sequential posted pricing. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 311–320. ACM, 2010.
9. Michal Feldman, Nick Gravin, and Brendan Lucier. Combinatorial auctions via posted prices. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 123–135. SIAM, 2015.

10 Michal Feldman, David Kempe, Brendan Lucier, and Renato Paes Leme. Pricing public goods for private sale. In *Proceedings of the fourteenth ACM conference on Electronic commerce*, pages 417–434. ACM, 2013.

11 Nima Haghpanah, Nicole Immorlica, Vahab Mirrokni, and Kamesh Munagala. Optimal auctions with positive network externalities. *ACM Transactions on Economics and Computation*, 1(2):13, 2013.

12 Aniko Hannak, Gary Soeller, David Lazer, Alan Mislove, and Christo Wilson. Measuring price discrimination and steering on e-commerce web sites. In *Proceedings of the 2014 conference on internet measurement conference*, pages 305–318. ACM, 2014.

13 Jason Hartline, Vahab Mirrokni, and Mukund Sundararajan. Optimal marketing strategies over social networks. In *Proceedings of the 17th international conference on World Wide Web*, pages 189–198. ACM, 2008.

14 Jason D. Hartline. Mechanism design and approximation, 2016.

15 Jason D. Hartline and Tim Roughgarden. Simple versus optimal mechanisms. In *Proceedings 10th ACM Conference on Electronic Commerce (EC-2009), Stanford, California, USA, July 6–10, 2009*, pages 225–234, 2009. doi:10.1145/1566374.1566407.

16 Robert Kleinberg and Seth Matthew Weinberg. Matroid prophet inequalities. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 123–136. ACM, 2012.

17 Andreu Mas-Colell, Michael Dennis Whinston, Jerry R Green, et al. *Microeconomic theory*, volume 1. Oxford university press New York, 1995.

18 Roger B Myerson. Optimal auction design. *Mathematics of operations research*, 6(1):58–73, 1981.

19 Paul A. Samuelson. The pure theory of public expenditure. *The review of economics and statistics*, pages 387–389, 1954.

Half-Integral Linkages in Highly Connected Directed Graphs*†

Katherine Edwards[1], Irene Muzi[2], and Paul Wollan[3]

1 **Nokia Bell Labs, Murray Hill, NJ, USA**
katherine.edwards@nokia-bell-labs.com
2 **Dept. of Computer Science, University of Rome, "La Sapienza", Rome, Italy**
muzi@di.uniroma1.it
3 **Dept. of Computer Science, University of Rome, "La Sapienza", Rome, Italy**
wollan@di.uniroma1.it

Abstract

We study the half-integral k-Directed Disjoint Paths Problem ($\frac{1}{2}$kDDPP) in highly strongly connected digraphs. The integral kDDPP is NP-complete even when restricted to instances where $k = 2$, and the input graph is L-strongly connected, for any $L \geq 1$. We show that when the integrality condition is relaxed to allow each vertex to be used in two paths, the problem becomes efficiently solvable in highly connected digraphs (even with k as part of the input). Specifically, we show that there is an absolute constant c such that for each $k \geq 2$ there exists $L(k)$ such that $\frac{1}{2}$kDDPP is solvable in time $O(|V(G)|^c)$ for a $L(k)$-strongly connected directed graph G. As the function $L(k)$ grows rather quickly, we also show that $\frac{1}{2}$kDDPP is solvable in time $O(|V(G)|^{f(k)})$ in $(36k^3 + 2k)$-strongly connected directed graphs. We show that for each $\epsilon < 1$, deciding half-integral feasibility of kDDPP instances is NP-complete when k is given as part of the input, even when restricted to graphs with strong connectivity ϵk.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases linkage, directed graph, treewidth

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.36

1 Introduction

Let $k \geq 1$ be a positive integer. An *instance of a directed k-linkage problem* is an ordered tuple (G, S, T) where G is a directed graph and $S = (s_1, \ldots, s_k)$ and $T = (t_1, \ldots, t_k)$ are each ordered sets of k distinct vertices in G. The instance is *integrally feasible* if there exist paths $P_1, \ldots, P_k$ such that P_i is a directed path from s_i to t_i for $1 \leq i \leq k$ and the paths P_i are pairwise vertex disjoint. The paths $P_1, \ldots, P_k$ will be referred to as an *integral solution* to the linkage problem.

The *k-Directed Disjoint Paths Problem* (**kDDPP**) takes as input an instance of a directed k-linkage problem. If the problem is integrally feasible, we output an integral solution and otherwise, return that the problem is not feasible. The kDDPP is notoriously difficult. The problem was shown to be NP-complete even under the restriction that $k = 2$ by Fortune, Hopcroft and Wyllie [4].

* A full version of this article is available at https://arxiv.org/abs/1611.01004.
† Supported by the European Research Council under the European Unions Seventh Framework Programme (FP7/2007-2013)/ERC Grant Agreement no. 279558.

In an attempt to make the kDDPP more tractable, Thomassen [16] asked if the problem would be easier if we assume the graph is highly connected . Define a *separation* in a directed graph G as a pair (A, B) with $A, B \subseteq V(G)$ such that $A \cup B = V(G)$ and where there does not exist an edge (u, v) with $u \in A \setminus B$ and $v \in B \setminus A$. The *order* of the separation (A, B) is $|A \cap B|$. The separation is *trivial* if $A \subseteq B$ or $B \subseteq A$. The graph G is *strongly k-connected* if $|V(G)| \geq k + 1$ and there does not exist a nontrivial separation of order at most $k - 1$. Let $k \geq 1$ and define a directed graph G to be *integrally k-linked* if every linkage problem (G, S, T) is integrally feasible. Thomassen conjectured [16] that there exists a function f such that every $f(k)$-strongly connected digraph G is integrally k-linked. He later answered his own conjecture in the negative [17], showing that no such function $f(k)$ exists. Moreover, he also showed [17] for all $L \geq 1$, the 2DDPP is NP-complete even when restricted to problem instances where the graph is L-strongly connected.

In this article, we relax the kDDPP problem by requiring that a potential solution not use any vertex more than twice. Define a directed k-linkage problem (G, S, T) to be *half-integrally feasible* if $S = (s_1, \ldots, s_k)$, and $T = (t_1, \ldots, t_k)$ and there exist paths $P_1, \ldots, P_k$ such that:

- for all $1 \leq i \leq k$, P_i is a directed path from s_i to t_i, and
- for every vertex $v \in V(G)$, v is contained in at most two distinct paths P_i.

The paths $P_1, \ldots, P_k$ form a *half-integral solution.*

The main result of this article is that the $\frac{1}{2}$kDDPP is polynomial time solvable (even with k as part of the input) when the graph is sufficiently highly connected. Define a graph G to be *half-integrally k-linked* if every k disjoint paths problem (G, S, T) is half-integrally feasible.

▶ **Theorem 1.** *For all integers $k \geq 1$, there exists a value $L(k)$ such that every strongly $L(k)$-connected graph is half-integrally k-linked. Moreover, there exists an absolute constant c such that given an instance (G, S, T) of the $\frac{1}{2}$kDDPP where G is $L(k)$-connected, we can find a solution in time $O(|V(G)|^c)$.*

The assumption that G is highly connected in Theorem 1 cannot be omitted under the usual complexity assumptions.

▶ **Theorem 2.** *For all $\epsilon < 1$, it is NP-complete to determine whether a given kDDPP instance (G, S, T) is half-integrally feasible, even under the assumption that G is ϵk-strongly connected.*

The value for $L(k)$ in Theorem 1 grows extremely quickly. However, when we fix k, we can still efficiently solve the $\frac{1}{2}$kDDPP with a significantly weaker bound on the connectivity than that given in Theorem 1.

▶ **Theorem 3.** *There exists a function f satisfying the following. Let $k \geq 1$ be a positive integer. Given a k-linkage problem (G, S, T) such that G is $(36k^3 + 2k)$-strongly connected, we can determine if the problem is half-integrally feasible and if so, output a half-integral solution, in time $O(|V(G)|^{f(k)})$.*

Given that the kDDPP is NP-complete even in the case $k = 2$, previous work on the problem has focused on various relaxations of the problem. Schrijver [14] showed that for fixed k, the kDDPP is polynomial time solvable when the input graph is assumed to be planar. Later, Cygan et al. [1] improved this result, showing that the kDDPP is fixed parameter tractable with the assumption that the input graph is planar. In their recent series of articles [8, 7, 10] leading to the breakthrough showing the grid theorem holds for directed graphs, Kawarabayashi and Kreutzer and Kawarabayashi et al. showed the following

relaxation of the kDDPP can be efficiently resolved for fixed k. They showed that there exists a polynomial time algorithm which, given an instance $(G, S = (s_1, \ldots, s_k), T = (t_1, \ldots, t_k))$ of the kDDPP, does one of the following:

- find directed paths P_i, $1 \leq i \leq k$, such that P_i links s_i to t_i and for every vertex v of G, v is in at most four distinct P_i, or
- determine that no integral solution to (G, S, T) exists.

In terms of hardness results, Slivkins [15] showed that the kDDPP is $W[1]$-complete even when restricted to acyclic graphs. Kawarabayashi et al. [7] announced that the proof of Slivkins result can be extended to show that the $\frac{1}{2}$kDDPP is also $W[1]$-complete.

There are two primary steps in the proof of Theorem 1. First, we show that any highly connected graph contains a large structure which we can use to connect up the appropriate pairs of vertices. The exact structure we use is a *bramble of depth two*. A bramble is a set of pairwise touching, connected (strongly connected) subgraphs; they are widely studied certificates of large tree-width both in directed and undirected graphs. See Sections 2 and 3 for the exact definitions and further details. The existence of such a bramble of depth two follows immediately from Kawarabayashi and Kreutzer's proof of the grid theorem [9]; however, the algorithm given in [9] only runs in polynomial time for fixed size of the bramble. We show in Section 4 that from appropriate assumptions which will hold both in the proof of Theorem 1 and Theorem 3, we are able to find a large bramble of depth two in time $O(n^c)$ for a graph on n vertices and some absolute constant c.

The second main step in the proof of Theorem 1 is to show how we can use such a bramble of depth two to find the desired solution to a given instance of the $\frac{1}{2}$kDDPP. Define a *linkage* to be a set of pairwise disjoint paths. We show in Section 5 that given an instance (G, S, T) and a large bramble $\mathcal{B}$ of depth two, we can find a smaller, sub-bramble $\mathcal{B}' \subseteq \mathcal{B}$ along with a linkage $\mathcal{P}$ of order k such that every element of $\mathcal{P}$ is a path from an element of S to a distinct subgraph in $\mathcal{B}'$. Moreover, the linkage $\mathcal{P}$ is internally disjoint from $\mathcal{B}'$. At the same time, we find a linkage $\mathcal{Q}$ from distinct subgraphs of $\mathcal{B}'$ to the vertices T. Thus, by linking the appropriate endpoints of $\mathcal{Q}$ and $\mathcal{P}$ in the bramble $\mathcal{B}'$, we are able to find the desired solution to (G, S, T). The fact that the bramble $\mathcal{B}'$ has depth two ensures that the solution we find uses each vertex at most twice. This result is given as Theorem 11; the statement and proof are presented in Section 5.

Linking to a well-behaved structure (the bramble of depth two in the instance above) is a common technique in disjoint path and cycle problems in undirected graphs. See [6, 13] for examples. The main contribution of Theorem 11 is to extend the technique to directed graphs, and in particular, simultaneously find the linkage from S to $\mathcal{B}'$ and the linkage $\mathcal{Q}$ from $\mathcal{B}'$ to T. This is made significantly more difficult in the directed case by the directional nature of separations in directed graphs and the fact that there is no easy way to control how the separations between S and $\mathcal{B}'$ and those between $\mathcal{B}'$ and T cross.

The proofs of Theorems 1 and 3 are given in Section 6. The construction showing NP-completeness in Theorem 2 is given in the full version of this article [3], Section 7. Due to space constraints, some of the more technical proofs are also found in that version; see Sections 4 and 5.2 in particular.

2 Directed tree-width

An *arborescence* is a directed graph R such that R has a vertex r_0, called the *root* of R, with the property that for every vertex $r \in V(R)$ there is a unique directed path from r_0 to r. Thus every arborescence arises from a tree by selecting a root and directing all edges away

from the root. If $r, r' \in V(R)$ we write $r' > r$ if $r' \neq r$ and there exists a directed path in R from r to r'. If $(u, v) \in E(R)$ and $r \in V(R)$, we write $r > (u, v)$ if $r > v$ or $r = v$. Let G be a directed graph and $Z \subseteq V(G)$. A set $S \subseteq V(G) \setminus Z$ is *Z-normal* if there is no directed walk in $G - Z$ with the first and last vertex in S which also contains a vertex of $V(G) \setminus (S \cup Z)$. Note that every Z-normal set is a union of strongly connected components of $G - Z$.

Let G be a directed graph. A *tree decomposition* of G is a triple (R, β, γ), where R is an arborescence, $\beta : V(R) \to 2^{V(G)}$ and $\gamma : E(R) \to 2^{V(G)}$ are functions such that:

1. $\{\beta(r) : r \in V(R)\}$ is a partition of $V(G)$ into non-empty sets and
2. if $e \in E(R)$, then $\{\beta(r) : r \in V(R), r > e\}$ is $\gamma(e)$-normal.

The sets $\beta(r)$ are called the *bags* of the decomposition and the sets $\gamma(e)$ are called the *guards* of the decomposition. For any $r \in V(R)$, we define $\Gamma(r) := \beta(r) \cup \{\gamma(e) : \ e \text{ incident to } r\}$. The *width* of (R, β, γ) is the smallest integer w such that $|\Gamma(r)| \leq w + 1$ for all $r \in V(R)$. The *directed tree-width* of G is the minimum width of a tree decomposition of G.

Johnson, Robertson, Seymour, and Thomas showed that if we assume k and w are fixed positive integers, then we can efficiently resolve the kDDPP when restricted to directed graphs of tree-width at most w [5].

▶ **Theorem 4** ([5], Theorem 4.8)**.** *For all $t \geq 1$, there exists a function f satisfying the following. Let $k \geq 1$, and let (G, S, T) be an k-linkage problem such that the directed tree-width of G is at most t. Then we can determine if (G, S, T) is integrally feasible and if so, output an integral solution, in time $O(|V(G)|^{f(k)})$.*

A simple construction shows that the same result holds to efficiently resolve k-linkage problems half-integrally when k and the tree-width of the graph are fixed. We first define the following operation. To *double* a vertex v in a directed graph G, we create a new vertex v' and add the edges (u, v') for all edges $(u, v) \in E(G)$, the edges (v', u) for all edges $(v, u) \in E(G)$ and the edges (v, v') and (v', v).

▶ **Corollary 5.** *For all $t \geq 1$, there exists a function f satisfying the following. Let $k \geq 1$, and let (G, S, T) be an instance of a k-linkage problem such that the directed tree-width of G is at most t. Given in input (G, S, T) and a directed tree-decomposition of G of width at most t, we can determine if the problem is half-integrally feasible and if so, output a half-integral solution, in time $O(|V(G)|^{f(k)})$.*

Proof. Fix $w \geq 1$ to be a positive integer. Let $(G, S = (s_1, \ldots, s_k), T = (t_1, \ldots, t_k))$ be an instance of a k-linkage problem where G has tree-width at most w. Let G' be the directed graph obtained by doubling every vertex $v \in V(G)$. Define the k-linkage problem $(G', S^* = (s_1^*, \ldots, s_k^*), T^* = (t_1^*, \ldots, t_k^*))$ by letting $s_i^* = s_i$ and $t_i^* = t_i'$ for $1 \leq i \leq k$. Thus, (G, S, T) is half-integrally feasible if and only if (G', S^*, T^*) is integrally feasible. Moreover, any integral solution to (G', S^*, T^*) can be easily converted to a half-integral solution for the original problem (G, S, T).

Let (R, β, γ) be a tree decomposition of G of width w. Observe that (R, β', γ') defined by $\beta'(r) = \{\{v, v'\} : v \in \beta(r)\}$ and $\gamma'(r) = \{\{v, v'\} : v \in \gamma(r)\}$ yields a tree decomposition of G' of width at most $2w$. Thus, by Theorem 4, we can determine if $(G', S^* = (s_1^*, \ldots, s_k^*), T^* = (t_1^*, \ldots, t_k^*))$ is integrally feasible and find an solution when it is, in polynomial time assuming k and w are fixed, proving the claim. ◀

3 Certificates for large directed tree-width

A *bramble* in a directed graph G is a set $\mathcal{B}$ of strongly connected subgraphs $B \subseteq G$ such that if $B, B' \in \mathcal{B}$, then $V(B) \cap V(B') \neq \emptyset$ or there exists edges $e, e' \in E(G)$ such that e links B to B' and e' links B' to B. A cover of $\mathcal{B}$ is a set $X \subseteq V(G)$ such that $V(B) \cap X \neq \emptyset$ for all

$B \in \mathcal{B}$. The *order* of a bramble is the minimum size of a cover of $\mathcal{B}$. The *bramble number*, denoted $bn(G)$, is the maximum order of a bramble in G. The elements of a bramble are called *bags*, and the *size* of a bramble, denoted $|\mathcal{B}|$, is the number of bags it contains.

The bramble number of a directed graph gives a good approximation of the tree-width, as seen by the following theorem of [12] as formulated by [10].

▶ **Theorem 6** ([12],[10]). *There exist constants c, c' such that for all directed graphs G, it holds that*

$$bn(G) \leq c \cdot tw(G) \leq c' \cdot bn(G).$$

Johnson, Robertson, Seymour, and Thomas showed one can efficiently (in fixed-parameter time) either find a large bramble in a directed graph or explicitly find a directed tree-decomposition. Note that the result is not stated algorithmically, but that the algorithm follows from the construction in the proof. Additionally, they looked at an alternate certificate of large tree-width, namely havens, but a haven of order $2t$ immediately gives a bramble of order t by the definitions.

▶ **Theorem 7** ([5], 3.3). *There exist constants c_1, c_2 such that for all t and directed graphs G, we can algorithmically find in time $O(|V(G)|^{c_1})$ either a bramble in G of order t or a tree-decomposition of G of order at most $c_2 t$. Moreover, if we find the bramble, it has at most $|V(G)|^{2t}$ elements.*

A long open question of Johnson, Robertson, Seymour, and Thomas [5] was whether sufficiently large tree-width in a directed graph would force the presence of a large directed grid minor. Let $r \geq 2$ be a positive integer. The *directed r-grid* J_r (or *cylindrical grid*) is the graph defined as follows. Let $C_1, \ldots, C_r$ be directed cycles of length $2r$. Let the vertices of C_i be labeled $v_1^i, \ldots, v_{2r}^i$ for $1 \leq i \leq r$. For $1 \leq i \leq 2r$, i odd, let P_i be the directed path $v_i^1, v_i^2, \ldots, v_i^r$. For $1 \leq i \leq 2r$, i even, let P_i be the directed path $v_i^r, v_i^{r-1}, \ldots, v_i^1$. The directed grid $J_r = \bigcup_1^r C_i \cup \bigcup_1^{2r} P_i$.

In a major recent breakthrough, Kreutzer and Kawarabayashi have confirmed the conjecture of Johnson et al.

▶ **Theorem 8** ([10]). *There is a function $f : \mathbb{N} \to \mathbb{N}$ such that given any directed graph and any fixed constant k, in polynomial time, we can obtain either*

1. *a cylindrical grid of order k as a butterfly minor, or*
2. *a directed tree decomposition of width at most $f(k)$.*

For our purposes, we will use brambles when attempting to solve the $\frac{1}{2}$kDDPP. However, in order to ensure that the paths we find don't use any vertex more than twice, we require the bramble to have *depth* two. Define the *depth* of a bramble $\mathcal{B} = \{B_1, \ldots, B_t\}$ in a directed graph G to be the $\max_{v \in V(G)} |\{i : v \in V(B_i)\}|$; in other words, a bramble has depth at most k for some positive integer k if no vertex is contained in more than k distinct subgraphs in the bramble. Note that if $\mathcal{B}$ has depth k and size t, then it has order at least $\lceil t/k \rceil$.

▶ **Lemma 9.** *For all $t \geq 2$, the directed t-grid contains a model of a bramble $\mathcal{B}$ of size t and depth two.*

Proof. Let the cycles $C_1, \ldots, C_t$, paths $P_1, \ldots, P_{2t}$, and vertex labels v_i^j, $1 \leq i \leq 2t$, $1 \leq j \leq t$, be as in the definition of the directed t-grid. For every l, $1 \leq l \leq t$, and for every i, $1 \leq i \leq 2t$, let $P_i(l)$ be the subpath of P_i with endpoints v_i^1 and v_i^l. For $1 \leq i \leq t-1$, let C_i' be the (unique) cycle in $C_i \cup C_1 \cup P_{2i-1}(i) \cup P_{2i}(i)$ which contains all the vertices v_1^j, $1 \leq j \leq 2k$. Let $C_t' = C_t$. The cycles $C_1', \ldots, C_t'$ form a bramble of depth two and size t, as desired. ◀

4 Finding a bramble of depth two

As described in the introduction, we can actually find a bramble of depth two in time $O(n^c)$ for some absolute constant c without appealing to the full power of the directed grid theorem of [10]. Indeed, we can show that in a graph with large enough directed treewidth, we find what is called a sufficiently large *well-linked* set of vertices in a directed graph, and from that we are able to efficiently find a large bramble of depth two.[1]

▶ **Theorem 10.** *There exists a function f which satisfies the following. Let G be a directed graph on n vertices and $t \geq 1$ a positive integer. Let P be a directed path and $X \subseteq V(P)$ a well-linked set with $|X| \geq f(t)$. Then G contains a bramble $\mathcal{B} = B_1, \ldots, B_t$ of depth two. Moreover, given G, P, and X in input, we can find $\mathcal{B}$ in time $O(n^c)$ for some absolute constant c.*

The proof of Theorem 10 is given in [3], Section 4. The argument in many ways follows Diestel et al.'s proof of Robertson and Seymour's grid theorem (see [2] for the proof) for undirected graphs.

5 Linking in a bramble of depth two

The main result of this section is the following which shows that if we have a sufficiently large bramble of depth two, we can use it to efficiently resolve a given instance of the $\frac{1}{2}$kDDPP under a modest assumption on the connectivity of the graph.

▶ **Theorem 11.** *For all $k \geq 1$, there exists a positive integer t such that if G is a $(36k^3+2k)$-strongly connected directed graph, and G contains a bramble $\mathcal{B}$ of depth two and size t, then for every k-linkage problem instance (G, S, T) is half-integrally feasible. Moreover, given (G, S, T) and the bags of $\mathcal{B}$, we can find a solution in time $O(k^4 n^2)$.*

We begin with some notation. Recall that the doubling of a vertex in a directed graph was defined in Section 2. To *contract* a set of vertices U inducing a strongly connected subgraph of G is to delete U and create a new vertex v, then add edges (w, v) for all edges $(w, u) \in E(G)$ with $u \in U, w \notin U$ and edges (u, w) for all edges $(v, u) \in E(G)$ with $u \in U, w \notin U$.

Let $\mathcal{B}$ be a depth two bramble in a directed graph G and $\mathcal{B}_1 \subseteq \mathcal{B}$. Define the graph $G(\mathcal{B}_1; \mathcal{B})$ as follows: First, let G' be the graph obtained from G by doubling every vertex belonging to two bags of $\mathcal{B}$ and to at least one bag of $\mathcal{B}_1$. For each such vertex v, denote its double by v'. Let $\mathcal{B}'$ be the collection of $|\mathcal{B}_1|$ subsets of $V(G')$ obtained from $\mathcal{B}_1$ by replacing each vertex v belonging to a bag of $\mathcal{B}$ with v' in exactly one of the bags it belongs to. Thus, the elements of $\mathcal{B}'$ are pairwise disjoint and each induces a strongly connected subgraph of G', so $\mathcal{B}'$ is a depth 1 bramble in G'. Let $G(\mathcal{B}_1; \mathcal{B})$ be the graph obtained from G' by contracting each element of $\mathcal{B}'$. Denote by $K_{\mathcal{B}_1}$ the set of contracted vertices in $G(\mathcal{B}_1; \mathcal{B})$; note that the vertices of $K_{\mathcal{B}_1}$ form a bidirected clique. Observe that every double of a vertex of G' gets contracted, so $V(G(\mathcal{B}_1; \mathcal{B})) \setminus K_{\mathcal{B}_1} \subseteq V(G)$. For a vertex $v \in K_{\mathcal{B}_1}$, we write $\mathrm{im}(v)$ for the bag of $\mathcal{B}_1$ corresponding to the vertices contracted to v. We stress that each $\mathrm{im}(v)$ is a bag of $\mathcal{B}_1$; in particular $\mathrm{im}(v) \subseteq V(G)$.

Let S, T be disjoint subsets of the vertices of a directed graph G. A separation (A, B) *separates* S from T if $S \subseteq A$ and $T \subseteq B$. The separation (A, B) *properly separates* S from T if $S \setminus B$ and $T \setminus A$ are both nonempty. For a positive integer α, we say S is *α-connected* to T if every separation separating S from T has order at least α.

[1] A subset $X \subseteq V(G)$ of vertices of a directed graph G is *well-linked* if for any pair of subsets $U_1, U_2 \subseteq X$ with $|U_1| = |U_2|$, there exists a directed U_1 to U_2 linkage of order $|U_1|$.

Let G be a directed graph, and $\mathcal{B}' \subseteq \mathcal{B}$ be brambles of depth two. Let $X \subseteq V(G(\mathcal{B}';\mathcal{B})) \setminus K_{\mathcal{B}'}$. We say an $X - K_{\mathcal{B}'}$ or $K_{\mathcal{B}'} - X$ linkage $P_1, \ldots, P_{|X|}$ is $\mathcal{B}$-*minimal* if none of the paths contains internally a vertex in $K_{\mathcal{B}'}$ or in $\mathrm{im}(v)$ for some $v \in K_{\mathcal{B}} \setminus \cup_i P_i$.

We now give a quick outline of how the proof will proceed. Let us denote $S = (s_1, \ldots, s_k)$ and $T = (t_1, \ldots, t_k)$. Our approach to proving half-integral feasibility is in two steps. We find three sets of paths, one set of k paths linking S to the bramble $\mathcal{B}$, another set linking $\mathcal{B}$ to T, and a third linking the appropriate ends of paths in the first two sets to each other inside of $\mathcal{B}$. To get the first two sets of paths, we take advantage of the high connectivity of the graph. Linking half-integrally inside of the bramble is easy, and its structure allows us to link any pairs of vertices we like half-integrally. We need the union of the three sets of paths to form a half-integral solution, so we will choose the first and second sets each to be (almost) vertex-disjoint, and to intersect the bramble $\mathcal{B}$ in a very limited way. The third set of paths will be half-integral and completely contained in $\mathcal{B}$.

The underlying idea behind our approach to finding the first two sets of paths is to contract each bag of the bramble (after doubling vertices in two bags) and try to apply Menger's theorem. In trying to do this, some issues arise. First, we want the ends of all $2k$ paths to belong to distinct bags of $\mathcal{B}$. More concerningly, contracting the bags of the bramble may destroy the connectivity between the bramble and the terminals S and T. We solve this by throwing away a bounded number of bags from the bramble until we are left with a sub-bramble that is highly connected to S and from T. In Subsection 5.1, we will show how to find the first two sets of paths (Lemma 12), modulo finding the sub-bramble (Lemma 13), and the third set of paths (Lemma 14). Then we show how to put these pieces together to prove Theorem 11. The proof of Lemma 13 can be found in the full version of this paper [3], Section 5.2.

5.1 Linking into and inside of a depth two bramble

▶ **Lemma 12.** *Let G be a $(36k^3 + 2k)$-strongly connected directed graph and $\mathcal{B}$ be a bramble of depth two and size $> 188k^3$ in G. Let $(G, S = (s_1, \ldots, s_k), T = (t_1, \ldots, t_k))$ be a k-linkage problem instance. Then we can find paths $P_1^s, \ldots, P_k^s, P_1^t, \ldots, P_k^t$ and $\mathcal{B}' \subseteq \mathcal{B}$ satisfying the following:*

A1: *For each i, P_i^s is a directed path from s_i to some vertex s_i', and P_i^t is a directed path from some vertex t_i' to t_i.*

A2: *The vertices $s_1', \ldots, s_k', t_1', \ldots, t_k'$ belong to distinct bags of $\mathcal{B}'$, say $B_1^s, \ldots, B_k^s, B_1^t, \ldots, B_k^t$, respectively.*

A3: *Every vertex belongs to at most two of $P_1^s, \ldots, P_k^s$, and if a vertex v does belong to two paths, say P_i^s and $P_j^s (i \neq j)$, then $v = s_i'$ or $v = s_j'$.*

A4: *Similarly, every vertex belongs to at most two of $P_1^t, \ldots, P_k^t$, and if a vertex v does belong to two paths, say P_i^t and $P_j^t (i \neq j)$, then $v = t_i'$ or $v = t_j'$.*

A5: *For each i, the internal vertices of P_i^s and of P_i^t belong to at most one bag of $\mathcal{B}'$.*

A6: *For each i, j, ℓ all distinct , $P_i^s \cap P_j^t \cap (B_\ell^s \cup B_\ell^t) = \emptyset$.*

A7: *Every vertex belongs to at most two of $P_1^s, \ldots, P_k^s, P_1^t, \ldots, P_k^t$.*

Moreover, given the bags of $\mathcal{B}$, we can find the paths $P_1^s, \ldots, P_k^s, P_1^t, \ldots, P_k^t$ in time $O(k^4 n^2)$.

We will prove the following lemma as an intermediate step to Lemma 12.

▶ **Lemma 13.** *Let G be a $(36k^3 + 2k)$-strongly connected directed graph and $\mathcal{B}$ be a bramble of depth two and size $> 188k^3$ in G. Let (G, S, T) be a k-linkage problem instance. Assume $\mathcal{B}$ is disjoint from $\{s_i, t_i; 1 \leq i \leq k\}$. Then there exist brambles $\mathcal{B}_S$ and $\mathcal{B}_T$ with $\mathcal{B}_T \subseteq \mathcal{B}_S \subseteq \mathcal{B}$ such that S is $(36k^3 + 2k)$-connected to $K_{\mathcal{B}_S}$ in $G(\mathcal{B}_S; \mathcal{B})$ and T is $3k$-connected to $K_{\mathcal{B}_T}$ in $G(\mathcal{B}_T; \mathcal{B}_S)$. Also $|\mathcal{B}_S| - |\mathcal{B}_T| < 36k^3$. Moreover we can find $\mathcal{B}_S$ and $\mathcal{B}_T$ in time $O(k^4 n^2)$.*

The proof of Lemma 13 is found in [3], Section 5.2. But first, let's see how Lemma 13 implies Lemma 12.

Proof of Lemma 12. Consider the brambles $\mathcal{B}_S$ and $\mathcal{B}_T$ given by Lemma 13. Denote by W the vertices in $G(\mathcal{B}_T;\mathcal{B})$ that belong to exactly one bag in $\mathcal{B}_T$ and to two bags in $\mathcal{B}_S$.

▶ **Claim.** *There exist k vertex-disjoint paths $P_1,\dots,P_k$ in $G(\mathcal{B}_T;\mathcal{B})\setminus W$ where P_i links s_i to v_i, for some $v_i \in K_{\mathcal{B}_T}$.*

Suppose not; then by Menger's theorem there exists a separation (A,B) of order $< k$ in $G(\mathcal{B}_T;\mathcal{B})\setminus W$ separating S from $K_{\mathcal{B}_T}$. But then consider the following separation in $G(\mathcal{B}_S)$. Let

$$A' = (A \cap V(G(\mathcal{B}_S;\mathcal{B}))) \cup \{v \in K_{\mathcal{B}_S} : \mathrm{im}(v) \cap A \neq \emptyset\} \cup (K_{\mathcal{B}_S} \setminus K_{\mathcal{B}_T})$$

and

$$B' = (B \cap V(G(\mathcal{B}_S;\mathcal{B}))) \cup \{v \in K_{\mathcal{B}_S} : \mathrm{im}(v) \cap B \neq \emptyset\} \cup (K_{\mathcal{B}_S} \setminus K_{\mathcal{B}_T}).$$

Intuitively, (A',B') is the separation (A,B) viewed in the graph $G(\mathcal{B}_S;\mathcal{B})$, plus we add the vertices of $K_{\mathcal{B}_S}\setminus K_{\mathcal{B}_T}$ to each side. It's easy to check that (A',B') is a separation in $G(\mathcal{B}_S;\mathcal{B})$, since every vertex in $V(G(\mathcal{B}_T;\mathcal{B}))\setminus V(G(\mathcal{B}_S;\mathcal{B}))$ belongs to $\mathrm{im}(v)$ for some $v \in K_{\mathcal{B}_S}$. Also, we have $|A'\cap B'| \le 2|A\cap B| + 36k^3$ because every vertex belongs to at most two bags of $\mathcal{B}_S$ and every vertex in W belongs to one bag of $\mathcal{B}_S\setminus\mathcal{B}_T$. But this contradicts Lemma 13 and proves the claim.

Choose the paths $P_1,\dots,P_k$ so that they are $\mathcal{B}_T$-minimal in $G(\mathcal{B}_T;\mathcal{B})$. Let us now view these as paths in the original graph G: Since $V(G(\mathcal{B}_T;\mathcal{B}))\setminus K_{\mathcal{B}_T} \subseteq V(G)$, each vertex in P_i except v_i is a vertex of G, for each $1\le i\le k$. So choose $s_i' \in \mathrm{im}(v_i)$ such that there exists an edge from the second to last vertex of P_i to s_i'. Then let P_i^s be the path obtained from P_i by replacing v_i with s_i'. Notice that P_i^s is a path in G. The paths $P_1^s,\dots,P_k^s$ are internally disjoint, so they satisfy A3 .

▶ **Claim.** *There exist vertex-disjoint paths $Q_1,\dots,Q_k$ in $G(\mathcal{B}_T;\mathcal{B}_S)\setminus\{v_1,\dots,v_k,s_1',\dots,s_k'\}$ where Q_i links w_i to t_i for some $w_i \in K_{\mathcal{B}_T}$. Moreover, the vertices $v_1,\dots,v_k,w_1,\dots,w_k$ are distinct.*

Suppose not; then by Menger's theorem, in the graph $G(\mathcal{B}_T;\mathcal{B}_S)\setminus\{v_1,\dots,v_k,s_1',\dots,s_k'\}$ there is a separation (A,B) of order $< k$ properly separating $K_{\mathcal{B}_T}$ from T. But then $(A\cup\{v_1,\dots,v_k,s_1',\dots,s_k'\}, B\cup\{v_1,\dots,v_k,s_1',\dots,s_k'\})$ has order $< 3k$ and properly separates $K_{\mathcal{B}_T}$ from T in $G(\mathcal{B}_T;\mathcal{B}_S)$, contradicting Lemma 13. This proves the claim.

We may also choose the paths $Q_1,\dots,Q_k$ to be $\mathcal{B}_T$-minimal in $G(\mathcal{B}_T;\mathcal{B}_S)$. Viewing these paths as paths in G as above (symmetrically), we obtain paths $P_1^t,\dots,P_k^t$, with P_i^t joining t_i' to t_i. These paths satisfy A4 .

Let $\mathcal{B}' = \{\mathrm{im}(v) : v \in \{v_1,\dots,v_k,w_1,\dots,w_k\}\}$. For each i, set $B_i^s = \mathrm{im}(v_i)$ and $B_i^t = \mathrm{im}(w_i)$. We now check that the paths $P_1^s,\dots,P_k^s,P_1^t,\dots,P_k^t$ satisfy the seven assertions in the lemma statement. A1 , A2 , A3 and A4 have already been established.

To see that A5 holds, note that each of $P_1,\dots,P_k$ is internally disjoint from $K_{\mathcal{B}_T}$ in $G(\mathcal{B}_T;\mathcal{B})$. Similarly, the $Q_1,\dots,Q_k$ paths are internally disjoint from $K_{\mathcal{B}_T}$ in $G(\mathcal{B}_T;\mathcal{B}_S)$. Moreover, by the definition of $G(\mathcal{B}_T;\mathcal{B})$ and $G(\mathcal{B}_T;\mathcal{B}_S)$, every vertex not in $K_{\mathcal{B}_T}$ in either of those graphs belongs to at most one bag of $\mathcal{B}_T$ and therefore to at most one bag of $\mathcal{B}'$. It follows that for each i, each internal vertex of P_i^s and P_i^t belongs to at most one bag of $\mathcal{B}'$, proving A5 .

To see A6 , let $1 \leq i, j, \ell \leq k$ be distinct. Suppose for contradiction that some vertex v belongs to $P_i^s \cap P_j^t \cap (B_\ell^s \cup B_\ell^t)$. If v is an internal vertex of either P_i^s or P_j^t then v belongs to only one bag of $\mathcal{B}'$ by A5 . Also, if $v = s_i'$ or t_j' then v belongs to two bags of $\mathcal{B}'$. We deduce that v is an internal vertex of both P_i^s and P_j^t. Since we found P_i in the graph $G(\mathcal{B}_T; \mathcal{B}) \setminus W$, we know $v \notin W$ so v belongs to one bag in $\mathcal{B}_T$ and one bag of $\mathcal{B}_S$. But we found Q_j in the graph $G(\mathcal{B}_T; \mathcal{B}_S)$, so v belongs to one bag of $\mathcal{B}_T$ and two bags of $\mathcal{B}_S$. This is a contradiction, proving A6 .

Finally, let us check A7 . Suppose for contradiction's sake that some vertex $v \in V(G)$ belongs to three paths. By A3 and A4 , we must have $v \in P_i^s \cap P_j^s \cap P_\ell^t$ or $v \in P_i^t \cap P_j^t \cap P_\ell^s$ for some $1 \leq i, j, \ell \leq k$. If $v \in P_i^s \cap P_j^s \cap P_\ell^t$, then by A3 we may assume without loss of generality that $v = s_i'$. But the path Q_ℓ was found in a graph not containing v_i or s_i', so we must have $s_i' \in B_\ell^t \cap B_i^s$. Since $\mathcal{B}'$ is depth two, $v \notin B_j^s$ so v is an internal vertex of P_j^s, contradicting A5 . If $v \in P_i^t \cap P_j^t \cap P_\ell^s$, then without loss of generality $v = t_i' \in B_i^t = \mathrm{im}(w_i)$. By the $\mathcal{B}_T$-minimality of $P_1, \ldots, P_k$, v cannot be an internal vertex of P_ℓ so we have $v = s_\ell' \in B_\ell^s$. Since v belongs to two bags, A5 implies that $v = t_j'$, a contradiction.

It remains to check that we can indeed find these paths in time $O(k^4 n^2)$. Indeed finding the brambles $\mathcal{B}_S$ and $\mathcal{B}_T$ takes time $O(k^4 n^2)$ using Lemma 13. Then, the sets of paths $P_1, \ldots, P_k$ and $Q_1, \ldots, Q_k$ can be found in time $O(n^2)$ according to Menger's Theorem (see [11]), and from these we can easily get $P_1^s, \ldots, P_k^s, P_1^t, \ldots, P_k^t$ in linear time. ◀

The following lemma shows how to solve any linkage problem half-integrally in a depth two bramble, provided the terminals belong to distinct bags.

▶ **Lemma 14.** *For all $k \geq 2$, let G be a directed graph and let $S' = (s_1', \ldots, s_k')$ and $T' = (t_1', \ldots, t_k'))$ be two ordered k-tuples of vertices in G. Suppose $\mathcal{B}$ is a bramble of depth two in G, and $s_1', \ldots, s_k', t_1', \ldots, t_k'$ belong to distinct bags $B_1^s, \ldots, B_k^s, B_1^t, \ldots, B_k^t$, respectively of $\mathcal{B}$. Then there exist paths $P_1, \ldots, P_k$ such that P_i links s_i' to t_i' and, additionally, every vertex of G is in at most two distinct paths P_i. Finally, it also holds that $P_i \subseteq B_i^s \cup B_i^t$ for each i, and we can find the paths $P_1, \ldots, P_k$ in time $O(kn^2)$.*

Proof. For each i, we obtain P_i as follows. By the definition of a bramble, there exist vertices $v_i \in B_i^s$ and $w_i \in B_i^t$ with either $v_i = w_i$ or $(v_i, w_i) \in E(G)$. Since B_i^s and B_i^t are both strongly connected, there exist a directed path from s_i' to v_i contained in B_i^s and a directed path from w_i to t_i' contained in B_i^t. Take P_i to be the concatenation of these two paths. By construction, each P_i belongs to $B_i^s \cup B_i^t$. Further, since the bags $B_1^s, \ldots, B_k^s, B_1^t, \ldots, B_k^t$ are distinct, and every vertex in G belongs to at most two distinct bags, it follows that $P_1, \ldots, P_k$ is the desired collection of paths. Each P_i can be found in time $O(n^2)$, and so the overall running time of $O(kn^2)$ follows. ◀

We can deduce Theorem 11 from Lemmas 12 and 14 as follows.

Proof of Theorem 11. Let $P_1^s, \ldots, P_k^s, P_1^t, \ldots, P_k^t$ and $s_1', \ldots, s_k', t_1', \ldots, t_k'$ and $\mathcal{B}' = B_1^s, \ldots, B_k^s, B_1^t, \ldots, B_k^t$ satisfy A1 - A7 , as given by Lemma 12.

By A2 , G, $S' = (s_1', \ldots, s_k')$ and $T' = (t_1', \ldots, t_k'))$ satisfying the hypothesis of Lemma 14. Let $P_1, \ldots, P_k$ be the paths guaranteed by that lemma.

For each $1 \leq i \leq k$, let $Q_i = P_i^s P_i P_i^t$ be the concatenation of these three paths. Clearly, each Q_i is a directed walk linking s_i to t_i and therefore contains a directed path from s_i to t_i. We just need to check that the k paths are half-integral. Suppose for contradiction's sake that some vertex $v \in Q_i \cap Q_j \cap Q_\ell$ for some $1 \leq i, j, \ell \leq k$ all distinct. By symmetry, we can consider four cases.

Case 1: $v \in P_i \cap P_j$.
Then, by Lemma 14, $v \in (B_i^s \cup B_i^t) \cap (B_j^s \cup B_j^t)$, so v belongs to two bags of $\mathcal{B}'$. Then by A5 v is not an internal vertex of P_ℓ^s or P_ℓ^t, a contradiction.

Case 2: $v \in P_i^s \cap P_j^s \cap P_\ell$.
By A3 in Lemma 12, we may assume $v = s_i'$, so $v \in P_i$. Since $v \in P_\ell$, it follows $v \in B_i^s \cap (B_\ell^s \cup B_\ell^t)$. By A5 , v is not an internal vertex of P_j^s, so $v \in B_j^s$ as well, a contradiction.

Case 3: $v \in P_i^t \cap P_j^t \cap P_\ell$.
By A4 in Lemma 12, we may assume $v = t_i'$, so $v \in P_i$. Again, since $v \in P_\ell$, it follows $v \in B_i^t \cap (B_\ell^s \cup B_\ell^t)$. By A5 , v is not an internal vertex of P_j^t so $v \in B_j^s$ as well, a contradiction.

Case 4: $v \in P_i^s \cap P_j^t \cap P_\ell$.
By Lemma 14 $P_\ell \subseteq (B_\ell^s \cap B_\ell^t)$, but this contradicts A6 . By A6 $v \notin B_\ell^s \cap B_\ell^t$ so $v \notin P_\ell$, a contradiction.

The running time bound of $O(k^4 n^2)$ follows from the bounds given by Lemmas 12, 13 and 14. ◀

6 Proofs of Theorems 1 and 3

Given Theorems 10 and 11, it is now easy to complete the proofs of Theorems 1 and 3. We begin with Theorem 1.

Proof of Theorem 1. Let f be the function from Theorem 10. Let $t = t(k)$ be the value necessary for the size of the bramble in order to apply Theorem 11 and resolve an instance of $\frac{1}{2}$kDDPP.

Let G be an $f(t)$-strongly connected graph on n vertices, and let $(G, S = (s_1, \ldots, s_k), T = (t_1, \ldots, t_k))$ be an instance of the $\frac{1}{2}$kDDPP. We can greedily find a path P with $|V(P)| \geq f(k)$. Note that any subset of at most $f(k)$ vertices is well-linked, and thus, $V(P)$ is a well-linked set. By Theorem 10, we can find in time $O(n^{c_1})$ a bramble $\mathcal{B}$ of size at least t. As $f(t) \geq 36k^3 + 2k$, by Theorem 11, we can find a solution to (G, S, T) in time $O(k^4 n^2)$, completing the proof of the theorem. ◀

For the proof of Theorem 3, we will need two additional results from [9]. Note that in [9], neither statement is algorithmic, but the existence of the algorithm follows immediately from the constructive proof.

▶ **Lemma 15** ([9], 4.3). *Let G be a directed graph on n vertices and $\mathcal{B}$ a bramble in G. Then there is a path P intersecting every element of $\mathcal{B}$ and given G and $\mathcal{B}$ in input, we can find the path P in time $O(|\mathcal{B}|n^2)$.*

▶ **Lemma 16** ([9], 4.4). *Let G be a directed graph graph on n vertices, $\mathcal{B}$ a bramble of order $k(k+2)$ and P a path intersecting every element of $\mathcal{B}$. Then there exists a set $X \subseteq V(P)$ of order $4k$ which is well-linked. Given P, $\mathcal{B}$, and G in input, we can algorithmically find X in time $|\mathcal{B}|n^{O(k)}$.*

Proof of Theorem 3. Let $(G, S = (s_1, \ldots, s_k), T = (t_1, \ldots, t_k))$ be an instance of the $\frac{1}{2}$kDDPP. Let $n = |V(G)|$. Let t be the necessary size of a bramble in order to apply Theorem 11 to resolve an instance of the $\frac{1}{2}$kDDPP. Let f be the function in Theorem 10.

By Theorem 7, we can either find a tree decomposition of G of width at most $c_2((f(t)+2)^2)$ or a bramble $\mathcal{B}$ of order $(f(t)+2)^2$. Given the tree decomposition, by Corollary 5, we can solve (G, S, T) in time $O(n^{f_1(c_2(f(t)+2)^2)})$ for some function f_1.

If instead we find the bramble $\mathcal{B}$, in order to apply Theorem 11, we will have to convert it to a bramble of depth two. By Theorem 7, we may assume that $|\mathcal{B}| \leq n^{2(f(t)+2)^2}$. Thus, in time $n^{O(f(t)^2)}$, we can find a path P intersecting every element of $\mathcal{B}$ by Lemma 15. By Lemma 16, again in time $n^{O(f(t)^2)}$, we can find a well-linked subset $X \subseteq V(P)$ with $|X| \geq f(t)$. Finally, applying Theorem 10, we find can find a bramble $\mathcal{B}'$ of size t and depth two. Finally, by Theorem 11, we can resolve (G, S, T) in time $O(k^4 n^2)$. In total, the algorithm takes time $O(n^{f_2(k)})$ for some function f_2, as desired. ◀

References

1. Marek Cygan, Daniel Marx, Marcin Pilipczuk, and Michal Pilipczuk. The planar directed k-vertex-disjoint paths problem is fixed-parameter tractable. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 197–206, Oct 2013. doi:10.1109/FOCS.2013.29.
2. Reinhard Diestel, Tommy R Jensen, Konstantin Yu Gorbunov, and Carsten Thomassen. Highly connected sets and the excluded grid theorem. *Journal of Combinatorial Theory, Series B*, 75(1):61–73, 1999.
3. Katherine Edwards, Irene Muzi, and Paul Wollan. Half-integral linkages in highly connected directed graphs, 2016. URL: https://arxiv.org/abs/1611.01004.
4. Steven Fortune, John Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10(2):111–121, 1980.
5. Thor Johnson, Neil Robertson, Paul D Seymour, and Robin Thomas. Directed tree-width. *Journal of Combinatorial Theory, Series B*, 82(1):138–154, 2001.
6. Ken-ichi Kawarabayashi. An improved algorithm for finding cycles through elements. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 374–384. Springer, 2008.
7. Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Stephan Kreutzer. An excluded half-integral grid theorem for digraphs and the directed disjoint paths problem. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC'14, pages 70–78, New York, NY, USA, 2014. ACM.
8. Ken-ichi Kawarabayashi and Stephan Kreutzer. An excluded grid theorem for digraphs with forbidden minors. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 72–81. doi:10.1137/1.9781611973402.6.
9. Stephan Kreutzer Ken-Ichi Kawarabayashi. The directed grid theorem, 2014. URL: https://arxiv.org/abs/1411.5681.
10. Stephan Kreutzer Ken-ichi Kawarabayashi. The directed grid theorem. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 655–664. ACM, 2015.
11. Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 1(10):96–115, 1927.
12. Bruce Reed. Introducing directed tree width. *Electronic Notes in Discrete Mathematics*, (3):1–8, 2000.
13. Neil Robertson and Paul D Seymour. Graph minors XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995.
14. Alexander Schrijver. Finding k disjoint paths in a directed planar graph. *SIAM Journal on Computing*, 23(4):780–788, 1994.

15 Aleksandrs Slivkins. Parameterized tractability of edge-disjoint paths on directed acyclic graphs. *SIAM Journal on Discrete Mathematics*, 24(1):146–157, 2010.

16 Carsten Thomassen. 2-linked graphs. *European Journal of Combinatorics*, 1(4):371–378, 1980.

17 Carsten Thomassen. Highly connected non-2-linked digraphs. *Combinatorica*, 11(4):393–395, 1991.

Bounds on the Satisfiability Threshold for Power Law Distributed Random SAT*†

Tobias Friedrich[1], Anton Krohmer[2], Ralf Rothenberger[3], Thomas Sauerwald[4], and Andrew M. Sutton[5]

1 **Hasso Plattner Institute, Potsdam, Germany**
`tobias.friedrich@hpi.de`
2 **Hasso Plattner Institute, Potsdam, Germany**
`anton.krohmer@hpi.de`
3 **Hasso Plattner Institute, Potsdam, Germany**
`ralf.rothenberger@hpi.de`
4 **University of Cambridge, Cambridge, UK**
`thomas.sauerwald@cl.cam.ac.uk`
5 **Hasso Plattner Institute, Potsdam, Germany**
`andrew.sutton@hpi.de`

Abstract

Propositional satisfiability (SAT) is one of the most fundamental problems in computer science. The worst-case hardness of SAT lies at the core of computational complexity theory. The average-case analysis of SAT has triggered the development of sophisticated rigorous and non-rigorous techniques for analyzing random structures.

Despite a long line of research and substantial progress, nearly all theoretical work on random SAT assumes a *uniform* distribution on the variables. In contrast, real-world instances often exhibit large fluctuations in variable occurrence. This can be modeled by a *scale-free* distribution of the variables, which results in distributions closer to industrial SAT instances.

We study random k-SAT on n variables, $m = \Theta(n)$ clauses, and a power law distribution on the variable occurrences with exponent β. We observe a satisfiability threshold at $\beta = (2k-1)/(k-1)$. This threshold is tight in the sense that instances with $\beta \leqslant (2k-1)/(k-1)-\varepsilon$ for any constant $\varepsilon > 0$ are *unsatisfiable* with high probability (w. h. p.). For $\beta \geqslant (2k-1)/(k-1)+\varepsilon$, the picture is reminiscent of the uniform case: instances are *satisfiable* w. h. p. for sufficiently small constant clause-variable ratios m/n; they are *unsatisfiable* above a ratio m/n that depends on β.

1998 ACM Subject Classification G.2.1 Combinatorics

Keywords and phrases satisfiability, random structures, random SAT, power law distribution, scale-freeness, phase transitions

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.37

1 Introduction

Satisfiability of propositional formulas (SAT) is one of the most researched problems in theoretical computer science. SAT is widely used to model practical problems such as bounded

* The full version of this paper is available at `https://arxiv.org/abs/1706.08431`.
† The research leading to these results has received funding from the German Research Foundation (DFG) under grant agreement no. FR 2988 (ADLON).

model checking, hardware and software verification, automated planning and scheduling, and circuit design. Even *large industrial instances* with millions of variables can often be solved very efficiently by modern SAT solvers. The structure of these industrial SAT instances appears to allow a much faster processing than the theoretical worst-case of this NP-complete problem. It is an open and widely discussed question which structural properties make a SAT instance easy to solve for modern SAT solvers.

Random SAT. For modeling typical inputs, we study random propositional formulas. In random satisfiability, we have a distribution over Boolean formulas in conjunctive normal form (CNF). The degree of a variable in a CNF formula is the number of disjunctive clauses in which that variable appears either positively or negatively. Two interesting properties of random models are its *degree distribution* and its *satisfiability threshold*. The degree distribution $F(x)$ of a formula Φ is the fraction of variables that occur more than x times (negated or unnegated). A satisfiability threshold is a critical value around which the probability that a formula is satisfiable changes from 0 to 1.

Uniform random SAT. In the classical uniform random model, the degree distribution is binomial. On uniform random k-SAT, the *satisfiability threshold conjecture* [1] asserts if Φ is a formula drawn uniformly at random from the set of all k-CNF formulas with n variables and m clauses, there exists a real number r_k such that

$$\lim_{n\to\infty} \Pr\{\Phi \text{ is satisfiable}\} = \begin{cases} 1 & m/n < r_k; \\ 0 & m/n > r_k. \end{cases}$$

A well-known result of Friedgut [20] establishes that the transition is sharp, even though its location is not known exactly for all values of k (and may also depend on n). For $k = 2$, the critical threshold is $r_2 = 1$ [13, 16, 22]. Recently, Coja-Oghlan and Panagiotou [15] gave a sharp bound (up to lower order terms) with $r_k = 2^k \log 2 - \frac{1}{2}(1 + \log 2) \pm o_k(1)$. Ding, Sly, and Sun [18] derive an exact representation of the threshold for all $k \geqslant k_0$, where k_0 is a large enough constant. Explicit bounds also exist for low values of k, e.g., $3.52 \leqslant r_3 \leqslant 4.4898$ [23, 24, 17], and numerical estimates using the cavity method from statistical mechanics [28] suggest that $r_3 \approx 4.26$.

Other random SAT models. In the regular random model [10], formulas are constructed at random, but the degree distribution is fixed: each literal appears exactly $\lfloor \frac{km}{2n} \rfloor$ or $\lfloor \frac{km}{2n} \rfloor + 1$ times in the formula. Similarly, Bradonjic and Perkins [11] considered a random geometric k-SAT model in which $2n$ points are placed at random in $[0, 1]^d$. Each point corresponds to a unique literal, and clauses are formed by all k-sets of literals that lie together within a ball of diameter $\Theta(n^{-1/d})$. Again, this model has a binomial variable distribution.

Power law random SAT. Recently, there has been a paradigm shift when modeling real-world data. In many applications, it has been found that certain quantities do not cluster around a specific scale as suggested by a uniform distribution, but are rather inhomogeneous [14, 30]. In particular, the degree distribution in complex networks often follows a power law [29]. This means that the fraction of vertices of degree k is proportional to $k^{-\beta}$, where the constant β depends on the network. To mathematically study the behavior of such networks, random graph models that generate a power law degree distribution have been proposed [9, 26, 2, 31].

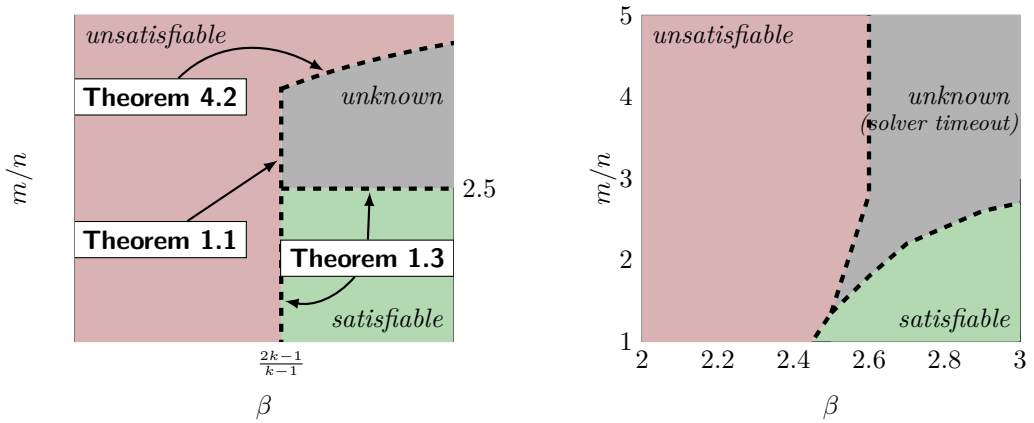


Figure 1 Illustration of our asymptotic results for the power law satisfiability threshold location when $n \to \infty$ (left) compared with empirical results for randomly generated power law 3-SAT formulas on $n = 10^6$ variables checked with the SAT solver MiniSAT (right). The timeout was set to one hour.

While there has been a large amount of research on power law random graphs in the past few years [32], there is little previous work on power law SAT formulas. Nevertheless, the observation that quantities follow a power law in real-world data has also emerged in the context of SAT [10]. As all aforementioned random SAT models assume strongly concentrated degree distributions, it was conjectured that this property might be modeled well by random formulas with a power law degree distribution.

To address this conjecture, and to help close the gap between the structure of uniform random and industrial instances, Ansótegui, Bonet, and Levy [6] recently proposed a power-law random SAT model. This model has been studied experimentally [6, 7, 4, 5], and empirical investigations found that (1) indeed the constraint graphs of many families of industrial instances obey a power-law and (2) SAT solvers that are constructed to specialize on industrial instances perform better on power-law formulas than on uniform random formulas. To complement these experimental findings, we contribute with this paper the first theoretical results on this model.

Our results. We study random k-SAT on n variables and $m = \Theta(n)$ clauses. Each clause contains $k = \Theta(1)$ different, independently sampled variables. Each variable x_i is chosen with non-uniform probability p_i and negated with probability $1/2$. A formal definition can be found in Section 2. We first study sufficient conditions under which the resulting k-SAT instances are *unsatisfiable*. Assume a probability distribution $\vec{p}$ on the variables where p_i is non-decreasing in $i \in \{1, \dots, n\}$. If the k most frequent variables are sufficiently common, we prove in Section 3 the following statement:

▶ **Theorem 1.1.** *Let Φ be a random k-SAT formula with probability distribution $\vec{p}$ on the variables (c.f. Definition 2.1), with $k \geqslant 2$ and $\frac{m}{n} = \Omega(1)$. If $p_{n-k+1} = \Omega((\frac{\log n}{n})^{1/k})$, then Φ is w. h. p. unsatisfiable.*

Our focus are power law distributions with some exponent β. Theorem 1.1 implies that power law random k-SAT formulas with $\beta = \frac{2k-1}{k-1} - \varepsilon$ for an arbitrary constant $\varepsilon > 0$ are unsatisfiable with high probability[1], cf. Corollary 3.1.

[1] We say that an event E holds *w. h. p.*, if there exists a $\delta > 0$ such that $\Pr[E] \geqslant 1 - \mathcal{O}(n^{-\delta})$.

In Section 4 we show that something similar holds for the clause-variable ratio $\frac{m}{n}$, i.e. power law random k-SAT formulas with $\frac{m}{n}$ bigger than some constant are unsatisfiable with high probability. Although this already follows from basic observations, we derive a better bound on the value of the constant.

▶ **Theorem 1.2.** *Let Φ be a random k-SAT formula with probability distribution $\vec{p}$ on the variables (c.f. Definition 2.1), with $k \geqslant 2$ and $r = \frac{m}{n}$. Φ is unsatisfiable w. h. p. if*

$$\left(1 - \tfrac{1}{2^k}\right)^r \left[\prod_{i=1}^{n} \left[2 - \left(1 - \frac{k \cdot p_i}{2^k - 1} \frac{1}{\left(1 - \frac{1}{2}k^2 \|\vec{p}\|_2^2\right)}\right)^m\right]\right]^{\frac{1}{n}} < 1.$$

In Section 5 we prove the following positive result, which complements our picture of the satisfiability landscape:

▶ **Theorem 1.3.** *Let Φ be a random k-SAT formula whose variable probabilities follow a power law distribution (c.f. Definition 2.2). If the power law exponent is $\beta \geqslant \frac{2k-1}{k-1} + \varepsilon$ for an arbitrary $\varepsilon > 0$, Φ is satisfiable with high probability if $\frac{m}{n}$ is a small enough constant.*

Together our main theorems prove that random k-SAT instances whose variables follow power law distributions do not only exhibit a phase transition for some clause-variable ratio $r = \frac{m}{n}$, but also around the power law exponent $\beta = \frac{2k-1}{k-1}$. Figure 1 contains an overview of our results. To prove these statements, we borrow tools developed for the uniform random SAT model. Note, however, that many of their common techniques like the differential equation method seem difficult to apply to non-uniform distributions; as removing a variable results in a more complex rescaling of the rest of the distribution. It is therefore crucial to perform careful operations on the formulas that leave the distribution of variables intact. To this end, we use techniques known from the analysis of power law random graphs.

Clause length. We focus on power law variable distributions but fix the length of every clause to $k \geqslant 2$. Power law models have also been proposed in which clause length is distributed by a power law as well [6, 7]. As long as there is a constant *minimum clause length* $k_{\min} \geqslant 2$, our results can be extended to this case in the following way.

If the clause lengths are distributed as a power law, there will appear $\Theta(n)$ clauses of length $k_{\min}$, and all other clauses are of larger size. In that case, Theorems 1.1 and 1.3 are directly applicable to the linear number of clauses with size $k_{\min}$ (obtaining different hidden constants); and we have that the formula is satisfiable with high probability if $\beta \geqslant \frac{2k_{\min}-1}{k_{\min}-1} + \varepsilon$ and m/n is a small enough constant. On the other hand, the formula is unsatisfiable with high probability, if $\beta \leqslant \frac{2k_{\min}-1}{k_{\min}-1} - \varepsilon$. Consequently, the satisfiability of the formula does (asymptotically) not depend on the second power law.

2 Definition of the Model and Preliminaries

We analyze random k-SAT on n variables and $m = \Theta(n)$ clauses, where $k \geqslant 2$. The constant $r := \frac{m}{n}$ is called *clause-variable ratio* or *constraint density*. We denote by $x_1, \ldots, x_n$ the Boolean variables. A clause is a disjunction of k literals $\ell_1 \vee \ldots \vee \ell_k$, where each literal assumes a (possibly negated) variable. Finally, a formula Φ in conjunctive normal form is a conjunction of clauses $c_1 \wedge \ldots \wedge c_m$. We conveniently interpret a clause c both as a Boolean formula and as a set of literals. Following standard notation, we write $|\ell|$ to refer to the indicator of the variable corresponding to literal ℓ. We say that Φ is satisfiable if there exists an assignment of variables $x_1, \ldots, x_n$ such that the formula evaluates to 1.

▶ **Definition 2.1** (Random k-SAT). Let m, n be given, and consider any probability distribution $\vec{p}$ on n variables with $\sum_{i=1}^{n} p_i = 1$. To construct a random SAT formula Φ, we sample m clauses independently at random. Each clause is sampled as follows:

1. Select k variables independently at random from the distribution $\vec{p}$. Repeat until no variables coincide.
2. Negate each of the k variables independently at random with probability $1/2$.

Observe that by setting $p_i = \frac{1}{n}$ for all i, we obtain again the uniform random SAT model. One can show (see full version [21]) that for power law distributions, the probability to sample a specific clause c is

$$(1 + o(1))\frac{k!}{2^k}\prod_{\ell \in c} p_{|\ell|}. \tag{1}$$

Power law Distributions. In this paper, we are mostly concerned with distributions p_i that follow a power law. To this end, we define two models: A *general* model to capture most power law distributions (which is harder to analyze), and a *concrete* model that gives us one instance of $\vec{p}$ depending only on n that can be used to compute precise leading constants. We use the general model to derive some asymptotic results; and the concrete model to compare with the uniform random SAT model and for the experiments.

Before we define these two models, let us establish the concept of a *weight* w_i of a variable x_i. The weight gives us (roughly) the expected number of times the variable appears in the formula. That is,

$$p_i := \frac{w_i}{\sum_j w_j}.$$

Thus, fixing the weights $\vec{w} = (w_1, \dots, w_n)$ also fixes the probability distribution $\vec{p}$. It is important to distinguish between the initial distribution of variables $\vec{p}$ and modified distributions that may arise as a result of stochastic considerations. For instance, the smallest-weight variable in a clause is clearly not distributed according to $\vec{p}$ (except in 1-SAT). To avoid confusion, we identify a variable with its weight, as the weights stay fixed throughout the analysis. For convenience, we further assume W. l. o. g. that the variables are ordered increasingly by weight, i. e. for $i \leqslant j$ we have $w_i \leqslant w_j$. Note that our definition of power law ensures that for $\beta > 2$, we have $\sum_j w_j = \Theta(n)$.

We are now ready to define the two models.

▶ **Definition 2.2** (General Power Law). Let the weights $\vec{w} := w_1, \dots, w_n$ be given, and let W be a weight selected uniformly at random. We say that $\vec{w}$ follows a power law with exponent β, if $w_1 = \Theta(1)$, $w_n = \Theta(n^{\frac{1}{\beta-1}})$, and for all $w \in [w_1, w_n]$ it holds

$$F(w) := \Pr[W \geqslant w] = \Theta(w^{1-\beta}) \tag{2}$$

Whenever we need the explicit constants bounding the distribution function, we refer to them by α_1, α_2 as in

$$\alpha_1 w^{1-\beta} \leqslant F(w) \leqslant \alpha_2 w^{1-\beta}. \tag{3}$$

We point out that Definition 2.2 assumes a deterministic weight sequence; but it can be easily generalized to also support randomly generated weights.

For the concrete model, we define the weights as follows.

▶ **Definition 2.3** (Concrete Power Law)**.** Given a power law exponent β, we call $\vec{w}$ the concrete power law sequence, if

$$w_{n-i+1} := \left(\tfrac{n}{i}\right)^{\frac{1}{\beta-1}}. \tag{4}$$

One can check that for these concrete weights, it holds $n \cdot F(w) = \lfloor nw^{1-\beta} \rfloor$, so in a sense, they are a canonical choice for producing a power law weight distribution.

It remains to show that using a power law distribution in Definition 2.1 indeed results in a power law distribution of variable occurrences. Ansótegui et al. [7] provide a proof sketch for this fact, we prove it rigorously in the full version [21] of the paper.

▶ **Theorem 2.4.** *Let Φ be a random k-SAT formula that follows an arbitrary power law distribution with exponent β (c.f. Definition 2.2) and $m = \Theta(n)$. Then, there are $d_{\min} = \Theta(w_{\min})$ and $d_{\max} = \Theta(w_{\max})$, such that for all $d_{\min} \leqslant d \leqslant d_{\max}$ w. h. p. it holds that*

$$N_{\geqslant d} = \Theta(n \cdot d^{1-\beta}),$$

where $N_{\geqslant d}$ is the number of variables that appear at least d times in Φ.

To analyze power law distributions, we often make use of the following result of Bringmann, Keusch, and Lengler [12, Lemma B.1], which allows replacing sums by integrals.

▶ **Theorem 2.5** ([12])**.** *Let $f\colon \mathbb{R} \to \mathbb{R}$ be a continuously differentiable function, and let $F^{>}(w) := \Pr[W > w]$. Then, for any $0 \leqslant \underline{w} \leqslant \bar{w}$,*

$$\sum_{i\in[n], \underline{w} \leqslant w_i \leqslant \bar{w}} \tfrac{1}{n} f(w_i) = f(\underline{w}) \cdot F(\underline{w}) - f(\bar{w}) \cdot F^{>}(\bar{w}) + \int_{\underline{w}}^{\bar{w}} f'(w) \cdot F(w)\,\mathrm{d}w.$$

Using this theorem, the following corollary can be shown (see full version [21]):

▶ **Corollary 2.6.** *Let the variables w_i be power law distributed with exponent $\beta > 2$, and define $W_{\geqslant w} := \sum_{i\in[n]\colon w_i \geqslant w} w_i$. Then, $W_{\geqslant w} = \Theta(nw^{2-\beta})$.*

Hence, $\sum_j w_j = W_{\geqslant w_1} = \Theta(n)$ and therefore $p_i = \Theta(\frac{w_i}{n})$. Finally, we denote by V the random variable describing the weight of a SAT variable chosen according to a power law distribution p_i, that is, $\Pr[V = w] = \sum_i p_i \cdot \mathbb{1}[w_i = w]$, where $\mathbb{1}$ denotes the indicator variable of the event. Note that this is not equivalent to W, since there is a subtle difference in the two random processes: W is a random variable drawn uniformly at random from $w_1, \ldots, w_n$, whereas V is a random variable drawn from the same set, but with the non-uniform distribution $p_1, \ldots, p_n$. Hence, by Corollary 2.6,

$$\Pr[V \geqslant w] = \Theta(w^{2-\beta}). \tag{5}$$

3 Small Power Law Exponents are Unsatisfiable

For small power law exponents, one can show that they result in formulas that are unsatisfiable (for large n) for all constant clause-variable ratios. The rationale behind this is that large variables with weight $\Theta(w_n)$ appear polynomially often together in a clause. For constant k, they thus appear in all 2^k configurations (negated and non-negated), making the formula trivially unsatisfiable. Theorem 1.1, already stated in the introduction, gives a sufficient condition on the variable distribution to make a random k-SAT formula unsatisfiable.

▶ **Theorem 1.1.** *Let Φ be a random k-SAT formula with probability distribution $\vec{p}$ on the variables (c.f. Definition 2.1), with $k \geqslant 2$ and $\frac{m}{n} = \Omega(1)$. If $p_{n-k+1} = \Omega((\frac{\log n}{n})^{1/k})$, then Φ is w. h. p. unsatisfiable.*

Proof. Recall that p_i is without loss of generality increasing in i. Consider the k largest variables $n-k+1, \dots, n$. We call $\mathcal{E}_i$ the event that clause i consists of these variables. Then,

$$\Pr[\mathcal{E}_i] = \Omega(p_{n-k+1}^k) = \Omega(\tfrac{\log n}{n}).$$

Since each clause is drawn independently at random, we obtain by a Chernoff bound (see for example Theorem 1.1 in [19]) that with high probability, the total number of clauses consisting of these variables is

$$|\mathcal{E}| := \sum_{i=1}^{m} \mathbb{1}[\mathcal{E}_i] = \Omega(\log n).$$

In other words, the number of clauses in which the k largest variables appear together increases as a logarithm in n. Since in each of these clauses, the literals appear negated or non-negated with constant probability $1/2$, we have that all 2^k possible combinations of negated and non-negated literals appear in the formula with probability at least

$$1 - 2^k \cdot (\tfrac{2^k-1}{2^k})^{|\mathcal{E}|} = 1 - n^{-\Omega(1)}$$

by the union bound. Since all 2^k combinations cannot be satisfied at once, the resulting formula is unsatisfiable. ◀

By applying Theorem 1.1 to a power law distribution on the variables, we obtain the following power law threshold for unsatisfiability.

▶ **Corollary 3.1.** *Let Φ be a random k-SAT formula that follows an arbitrary power law distribution fulfilling Definition 2.2. If the power law exponent is $\beta \leqslant \frac{2k-1}{k-1} - \varepsilon$ for an arbitrary $\varepsilon > 0$, Φ is unsatisfiable with high probability.*

Proof. Observe that from $\beta = \frac{2k-1}{k-1} - \varepsilon$ it follows $k = \frac{\beta-1}{\beta-2} - \varepsilon'$ for some constant ε'. By setting $nF(w) \leqslant k$ we obtain that the largest k variables all have weight $\Theta(w_n) = \Theta(n^{\frac{1}{\beta-1}})$. Consequently, when $\beta > 2$,

$$(p_{n-k})^k = \Theta(n^{-k\frac{\beta-2}{\beta-1}}) = \Theta(n^{-1+\varepsilon'\frac{\beta-2}{\beta-1}}) = \omega(\tfrac{\log n}{n}),$$

and the statement follows from Theorem 1.1. For the case where $\beta \leqslant 2$, one can show using Theorem 2.5 that $\sum_i w_i = \Theta(n^{\frac{1}{\beta-1}})$, and therefore $p_{n-k} = \Omega(1)$. Again, the statement follows from Theorem 1.1. ◀

4 Large Clause-Variable Ratios are Unsatisfiable

It is a well-known result that random SAT on any probability distribution will result in unsatisfiable formulas if the clause-variable ratio is high. This follows from the probabilistic method: The expected number of assignments that satisfy a formula is $2^n(1-2^{-k})^m$. This is independent from the variable distribution as long as each variable is negated with probability $1/2$. Hence, if the clause-variable ratio exceeds $\ln(2)/\ln(\frac{2^k}{2^k-1})$, the resulting formula will be unsatisfiable with high probability. This constant is rather large, however: In the case of $k = 3$ this yields an upper bound on the clause-variable ratio of ≈ 5.191. For the concrete power law distribution in Definition 2.3, the true threshold is much smaller. In fact, it appears to be below the satisfiability threshold for uniform random SAT.

Let us restate the main result, which will be proven with the Single Flip Method [25].

Table 1 Numerical upper bounds on the density threshold obtained from the Single-Flip Method (cf. Theorems 1.2 and 4.2). Empty fields indicate unsatisfiability for *all* constant densities by Theorem 1.1. To the best of our knowledge, the bounds for uniform random SAT with $k \geqslant 4$ are the currently best known numerical upper bounds. For $k = 3$ the best known unconditional numerical upper bound is 4.4898 [17].

	power law distribution with exponent β								uniform dist.
k	**2.2**	**2.3**	**2.4**	**2.5**	**2.6**	**2.7**	**2.8**	**2.9**	
3				3.48	3.71	3.87	3.99	4.08	4.67
4			7.87	8.42	8.78	9.04	9.23	9.37	10.23
5		16.27	17.75	18.64	19.21	19.61	19.90	20.11	21.33
7	67.21	75.74	79.81	82.09	83.49	84.42	85.07	85.54	87.88
10	619.28	662.48	680.93	690.36	695.77	699.12	701.34	702.88	708.94

▶ **Theorem 1.2.** *Let Φ be a random k-SAT formula with probability distribution $\vec{p}$ on the variables (c.f. Definition 2.1), with $k \geqslant 2$ and $r = \frac{m}{n}$. Φ is unsatisfiable w. h. p. if*

$$\left(1 - \tfrac{1}{2^k}\right)^r \left[\prod_{i=1}^{n} \left[2 - \left(1 - \frac{k \cdot p_i}{2^k - 1} \frac{1}{\left(1 - \frac{1}{2}k^2 \|\vec{p}\|_2^2\right)}\right)^m\right]\right]^{\frac{1}{n}} < 1.$$

The following is a corollary from this theorem:

▶ **Corollary 4.1.** *Let Φ be a random k-SAT formula that follows Definition 2.1 with $k \geqslant 2$, $r = \frac{m}{n}$ and $\|\vec{p}\|_2^2 = o(1)$. With high probability, Φ is unsatisfiable if*

$$\left(1 - \tfrac{1}{2^k}\right)^r \left(2 - \exp\left(-\left(\frac{k}{2^k - 1} r\right)(1 + o(1))\right)\right) < 1.$$

The proof can be found in the full version [21] of the paper. Interestingly, the above corollary gives the same inequality as the Single-Flip Method for uniform random SAT [25]. This shows that the uniform distribution resembles a worst-case for this method; and all other distributions can only improve this bound.

If $\vec{p}$ follows a power law distribution as in Definition 2.3, we can derive the following theorem, which gives an upper bound independent of n.

▶ **Theorem 4.2.** *Let Φ be a random k-SAT formula with $k \geqslant 2$ and $r = \frac{m}{n}$ that follows a power law distribution fulfilling Definition 2.3. Let further $N \in \mathbb{N}^+$ be any constant. If the power law exponent is $\beta > 2$, then Φ is w. h. p. unsatisfiable if*

$$\left(\left(1 - \tfrac{1}{2^k}\right)^r 2^{\frac{1}{N}} \prod_{l=1}^{N-1} \left[2 - \exp\left(-(1 + o(1))\, r \frac{k}{2^k - 1} \frac{\beta - 2}{\beta - 1} \left(\frac{N}{l}\right)^{\frac{1}{\beta - 1}}\right)\right]^{\frac{1}{N}}\right) < 1.$$

The bound from this Theorem improves as $N \to \infty$. As this expression is rather terse, we also numerically determine in Table 1 the smallest constant r such that the formula is unsatisfiable. We compare these values to the upper bounds for uniform random SAT obtained from the Single-Flip Method.

In the remainder of this section, we show Theorem 1.2 and defer the proof of Theorem 4.2 to the full version of the paper [21].

▶ **Definition 4.3** (Single-Flip Property). For a random formula Φ a truth assignment A has the *single-flip property* iff A satisfies Φ and every assignment A' obtained from A by flipping exactly one zero to one does *not* satisfy Φ.

Let N_{SF} be the number of truth assignments with the single-flip property for Φ. As argued in [25], such an assignment exists if Φ is satisfiable. From Markov's Inequality, we thus know $\Pr[\Phi \text{ satisfiable}] \leqslant \mathbb{E}[N_{SF}]$.

In the following, we derive a bound on $\mathbb{E}[N_{SF}]$. Using the non-uniform birthday paradox from [3] we can show that the probability of choosing a clause c is at most

$$\frac{k!}{2^k} \cdot \frac{\prod_{\ell \in c} p_{|\ell|}}{1 - \frac{1}{2}k^2 \|\vec{p}\|_2^2}.$$

To bound the number of assignments with the single-flip property, we use the following result.

▶ **Lemma 4.4** ([25]). *The expected number of assignments with the single-flip property is*

$$\mathbb{E}[N_{SF}] = \left(1 - \tfrac{1}{2^k}\right)^m \sum_{\text{assignment } A} \Pr[A \textit{ single-flip} \mid A \textit{ satisfying}].$$

Proof. Note that for a certain truth assignment A, the probability of choosing a clause which is not satisfied by A is $1/2^k$. Therefore, the probability that A is a satisfying assignment for Φ is exactly $\left(1 - \frac{1}{2^k}\right)^m$. ◀

We next bound the probability that a satisfying assignment A has the single-flip property.

▶ **Lemma 4.5.** *For a satisfying assignment $A = (a_1, a_2, \ldots, a_n) \in \{0,1\}^n$ it holds that*

$$\Pr[A \textit{ single-flip} \mid A \textit{ satisfying}] \leqslant \prod_{i:\ a_i = 0} 1 - \left(1 - \frac{k \cdot p_i}{2^k - 1} \frac{1}{\left(1 - \frac{1}{2}k^2 \|\vec{p}\|_2^2\right)}\right)^m.$$

Proof. For a satisfying assignment A to have the single-flip property, all assignments A^i obtained by flipping a bit $a_i = 0$ of A must not satisfy Φ. To fulfill this property for A^i, we have to choose at least one clause which contains $\bar{X}_i$ and $k-1$ other variables with appropriate signs so that A^i does not satisfy the clause. Let $S^i(c)$ denote the event that a clause c is satisfied by A, but not by A^i. Then,

$$\Pr[S^i(c)] = \frac{k! \cdot p_i \sum_{J \in \mathcal{P}_{k-1}([n] \setminus \{i\})} \prod_{j \in J} p_j}{2^k \left(1 - \frac{1}{2}k^2 \|\vec{p}\|_2^2\right)} \leqslant \frac{k \cdot p_i}{2^k \left(1 - \frac{1}{2}k^2 \|\vec{p}\|_2^2\right)}$$

since $\sum_{J \in \mathcal{P}_{k-1}([n] \setminus \{i\})} \prod_{j \in J} p_j \leqslant \frac{\|\vec{p}\|_1^{k-1}}{(k-1)!}$. The probability of choosing a clause not satisfied by A^i under the condition that A is satisfying is then

$$\Pr[S^i(c) \mid A \text{ sat}] = \Pr[S^i(c) \mid A \text{ satisfies } c] \leqslant \frac{k \cdot p_i}{2^k - 1} \frac{1}{\left(1 - \frac{1}{2}k^2 \|\vec{p}\|_2^2\right)}$$

as the probability of choosing a clause which is satisfied by any assignment is exactly $\frac{2^k - 1}{2^k}$. For a fixed assignment A^i we conclude

$$\begin{aligned} \Pr[A^i \text{ unsat} \mid A \text{ sat}] &= 1 - \left(1 - \Pr[S^i(c) \mid A \text{ sat}]\right)^m \\ &\leqslant 1 - \left(1 - \frac{k \cdot p_i}{2^k - 1} \frac{1}{\left(1 - \frac{1}{2}k^2 \|\vec{p}\|_2^2\right)}\right)^m. \end{aligned} \tag{6}$$

Algorithm 1 Clause Shrinking Algorithm

Input: k-SAT formula Φ; weight distribution $\vec{w}$

1: **for all** $c \in \Phi$ **do**
2: $\quad \ell_1 \leftarrow \operatorname{argmin}_{\ell \in c}\{w_{|\ell|}\}$
3: $\quad \ell_2 \leftarrow \operatorname{argmin}_{\ell \in c \setminus \{\ell_1\}}\{w_{|\ell|}\}$
4: $\quad c \leftarrow (\ell_1 \vee \ell_2)$
5: Solve Φ using any polynomial time 2-SAT algorithm

It remains to find the joint probability that all single-flipped assignments A^i for $1 \leqslant i \leqslant n$ with $a_i = 0$ are not satisfying. We show this using a correlation inequality by Farr [27]. The sets of clauses which are not satisfied by the A^i's are pairwise disjoint as each clause in the set for A^i has to contain $\bar{X}_i$, whereas each clause in the set for A^j $(j \neq i)$ can not contain $\bar{X}_i$. In the context of the correlation inequality from [27] we set $V = \{1, 2, \ldots, m\}$, $I = \{i \in \{1, 2, \ldots, n\} \mid a_i = 0\}$, $X_v = i$ iff the v-th clause is satisfied by A, but not by A^i, and $\mathcal{F}_i$ the "increasing" collection of non-empty subsets of V. The application of the Theorem then directly yields

$$\begin{aligned}\Pr[A \text{ single-flip} \mid A \text{ sat}] &= \Pr\Big[\bigcap_{i:\ a_i=0} A^i \text{ unsat} \mid A \text{ sat}\Big] \\ &\leqslant \prod_{i:\ a_i=0}\left[1 - \left(1 - \frac{k \cdot p_i}{2^k - 1}\frac{1}{\left(1 - \frac{1}{2}k^2\|\vec{p}\|_2^2\right)}\right)^m\right]. \qquad \blacktriangleleft\end{aligned}$$

Combining Lemmas 4.4 and 4.5 we get that the expected number of assignments with single-flip property is at most

$$\begin{aligned}\mathbb{E}[N_{SF}] &\leqslant \left(1 - \tfrac{1}{2^k}\right)^m \sum_{I \subseteq \{1,2,\ldots,n\}} \prod_{i \in I}\left[1 - \left(1 - \frac{k \cdot p_i}{2^k - 1}\frac{1}{\left(1 - \frac{1}{2}k^2\|\vec{p}\|_2^2\right)}\right)^m\right] \\ &= \left(1 - \tfrac{1}{2^k}\right)^m \prod_{i=1}^{n}\left[2 - \left(1 - \frac{k \cdot p_i}{2^k - 1}\frac{1}{\left(1 - \frac{1}{2}k^2\|\vec{p}\|_2^2\right)}\right)^m\right].\end{aligned}$$

This establishes Theorem 1.2.

5 Conditions for Satisfiability

In this section, we provide a complementary result to Theorems 1.1 and 4.2 proving that if $\beta \geqslant \frac{2k-1}{k-1} + \varepsilon$ and the clause-variable ratio $r = \frac{m}{n}$ does not exceed some small constant, then a random k-SAT formula with exponent β is satisfiable with high probability. Let us first restate the main result:

▶ **Theorem 1.3.** *Let Φ be a random k-SAT formula whose variable probabilities follow a power law distribution (c.f. Definition 2.2). If the power law exponent is $\beta \geqslant \frac{2k-1}{k-1} + \varepsilon$ for an arbitrary $\varepsilon > 0$, Φ is satisfiable with high probability if $\frac{m}{n}$ is a small enough constant.*

We show this statement by constructing an algorithm that satisfies Φ w. h. p. if the clause-variable ratio is small. Algorithm 1 contains a formal description. The main idea is to shrink all clauses to size 2 by selecting the literals with smallest weight in each clause; and then running any well-known (polynomial time) 2-SAT algorithm (e. g. [8]).

In the following, we seek to establish that Algorithm 1 will find a satisfying assignment (for small constraint densities) with high probability. To this end, we first analyze the probability distribution of a clause c after it has been shrunk.

▶ **Lemma 5.1.** *Let ℓ_1, ℓ_2 be the selected literals of an arbitrary clause $c \in \Phi$ in Algorithm 1. Then,*

$$\Pr[|\ell_1| = i, |\ell_2| = j] + \Pr[|\ell_1| = j, |\ell_2| = i] \leqslant \mathcal{O}(\tfrac{1}{n^2}(w_i w_j)^{1-\frac{1}{2}(k-2)(\beta-2)}).$$

Proof. W. l. o. g., we assume that $w_i \leqslant w_j$. Then, $\Pr[|\ell_1| = j, |\ell_2| = i] = 0$ by the definition of Algorithm 1. For the event $|\ell_1| = i$, $|\ell_2| = j$ to happen, all other $k-2$ literals in the clause must be of larger weight. By Equations (1) and (5),

$$\begin{aligned}\Pr[|\ell_1| = i, |\ell_2| = j] &= \frac{1}{2} \cdot \binom{k}{2} \cdot (1 + o(1)) \cdot p_i \cdot p_j \cdot \Pr[V \geqslant w_j]^{k-2} \\ &= \Theta(\tfrac{1}{n^2}) \cdot w_i w_j^{1-(k-2)(\beta-2)} \\ &\leqslant \mathcal{O}(\tfrac{1}{n^2}) \cdot (w_i w_j)^{1-\frac{1}{2}(k-2)(\beta-2)}.\end{aligned}$$

The last statement holds since $w_i \leqslant w_j$. ◀

Having derived a bound on the probability distribution of a shrunk clause, it is possible to compute the probability that the resulting 2-SAT formula is satisfiable. We use that the clauses are sampled independently. To avoid confusion, we write Φ' and c', whenever we talk about the shrunk formula and clauses. To upper bound the probability of Φ not being satisfiable, we look at so-called *bi-cycles* in Φ'.

▶ **Definition 5.2.** A *bi-cycle* of length l is a sequence of $l+1$ clauses of the form

$$(u, \ell_1), (\bar{\ell}_1, \ell_2), \ldots, (\bar{\ell}_{l-1}, \ell_l), (\bar{\ell}_l, v),$$

where $\ell_1, \ldots, \ell_l$ are literals of distinct variables and $u, v \in \{\ell_1, \ldots, \ell_l, \bar{\ell}_1, \ldots, \bar{\ell}_l\}$.

Chvatal and Reed [13, Theorem 3] show that if the formula Φ' is unsatisfiable, it must contain a bi-cycle. Consequently, by upper bounding the probability that a bi-cycle appears, we immediately obtain an upper bound on the probability that Φ' and henceforth Φ is unsatisfiable.

▶ **Theorem 5.3** (Chvatal and Reed [13])**.** *Let Φ' be any 2-SAT formula. If Φ' contains no bi-cycle, it is satisfiable.*

Before we are able to prove the main Theorem, we need the following auxiliary Lemma, the proof of which can be found in the full version of the paper [21].

▶ **Lemma 5.4.** *Let $\beta = \delta + 1 + \varepsilon$ for some $\varepsilon > 0$. For all $1 \leqslant l \leqslant n$, there is a constant c with*

$$\sum_{\substack{S \subseteq [n]: \\ |S| = l}} \prod_{i \in S} w_i^{\delta} \leqslant n^l \cdot c^l \tfrac{1}{l!}.$$

We are now able to show Theorem 1.3. As discussed above, we do this by upper bounding the probability that a bi-cycle appears in Φ'. To this end, we calculate the expected number of bi-cycles in Φ', observe that it is $\text{poly}(n)^{-1}$, and apply Markov's inequality. This yields that w. h. p., Φ' and thus Φ are satisfiable.

Proof of Theorem 1.3. We calculate the expected number of bi-cycles in Φ'. First, we fix a set $S \subseteq [n]$ of $l \geqslant 2$ variables to appear in a bi-cycle. Let X_B denote the random variable counting how many times a *specific* bi-cycle B with the variables from S appears in F. Then

$$\mathbb{E}[X_B] \leqslant \binom{m}{l+1}(l+1)! \cdot \Pr[u \vee x_1] \Pr[\bar{x}_l \vee v] \cdot \prod_{i=1}^{l-1} \Pr[\bar{x}_i \vee x_{i+1}].$$

The factor $\binom{m}{l+1}(l+1)!$ counts the possible positions of B in F. By Lemma 5.1,

$$\mathbb{E}[X_B] \leqslant m^{l+1} \cdot \left(\tfrac{c_1}{n^2}\right)^{l+1} \cdot \left(w_{|u|} w_{|v|} \prod_{i \in S} w_i^2\right)^{1-\frac{1}{2}(k-2)(\beta-2)}$$

for some suitable constant c_1. Now let X_S denote the random variable counting how many times *any* bi-cycle with the variables from S appears in F. There are $l!$ permutations of the l variables; and 2^l combinations of literals on l variables. Similarly, literals u and v have 4 possible sign combinations. Thus,

$$\mathbb{E}[X_S] \leqslant m^{l+1} \cdot l! \cdot 2^l \cdot \left(\tfrac{c_1}{n^2}\right)^{l+1} \cdot 4 \left(\sum_{i \in S} w_i^{1-\frac{1}{2}(k-2)(\beta-2)}\right)^2 \prod_{i \in S} w_i^{2-(k-2)(\beta-2)}.$$

To estimate the sum, we upper bound $w_i \leqslant w_n$ for all sets up to a certain size l_0, which we will determine later. We set $\delta := 2 - (k-2)(\beta-2)$ and define $\alpha(l)$ as

$$\left(\sum_{i \in S} w_i^{\delta/2}\right)^2 \leqslant \alpha(l) := \begin{cases} \mathcal{O}(l^2), & \text{if } \delta \leqslant 0, \\ l_0^2 \cdot w_n^\delta, & \text{if } \delta > 0 \text{ and } l \leqslant l_0, \\ \mathcal{O}(n^2), & \text{otherwise.} \end{cases}$$

Now let X denote the random variable counting the number of bi-cycles that appear in F.

$$\mathbb{E}[X] \leqslant \sum_{l=2}^{n} 2^{l+2} \cdot m^{l+1} \cdot l! \cdot (\tfrac{c_1}{n^2})^{l+1} \cdot \alpha(l) \sum_{\substack{S \subseteq [n] \\ |S|=l}} \prod_{i \in S} w_i^\delta.$$

Since $\delta + 1 = 2 - (k-2)(\beta-2) + 1 < \beta$ by our assumption $\beta \geqslant \frac{2k-1}{k-1} + \varepsilon$, we can apply Lemma 5.4. Using $r := m/n$, we obtain that the right-hand side is at most

$$\mathbb{E}[X] \leqslant \sum_{l=2}^{n} 2^{l+2} \cdot m^{l+1} \cdot l! \cdot (\tfrac{c_1}{n^2})^{l+1} \cdot \alpha(l) \cdot n^l \cdot c^l \tfrac{1}{l!} \leqslant \tfrac{1}{n} \sum_{l=2}^{n} c_2^l \cdot r^l \cdot \alpha(l), \tag{7}$$

for some suitable constant c_2. Since r is a small enough constant we thus have $c_2 \cdot r < 1$. If $\delta \leqslant 0$, we are finished, since then

$$\tfrac{1}{n} \sum_{l=2}^{n} c_2^l \cdot r^l \cdot \alpha(l) \leqslant \tfrac{1}{n} \sum_{l=2}^{n} (c_2 \cdot r)^l \cdot l^2 \leqslant \mathcal{O}(\tfrac{1}{n}).$$

Otherwise, if $\delta > 0$, we choose $l_0 := -4 \cdot \ln^{-1}(c_2 r) \ln(n)$, which ensures $(r \cdot c_2)^l = \mathcal{O}(n^{-4})$ for all $l > l_0$. For $l = 2, \ldots, l_0$, equation (7) sums up to at most

$$\tfrac{1}{n} \sum_{l=2}^{l_0} (c_2 r)^l \cdot l_0^2 \cdot w_n^\delta = \mathcal{O}(\log^3(n) \cdot n^{1-k\frac{\beta-2}{\beta-1}}),$$

where we substituted $w_n = \Theta(n^{\frac{1}{\beta-1}})$ and $\delta = 2-(k-2)(\beta-2)$. Since $\beta \geqslant \frac{2k-1}{k-1} + \varepsilon$, the exponent $1 - k\frac{\beta-2}{\beta-1} < -\varepsilon'$ is negative, and we thus have

$$\mathbb{E}[X] \leqslant \tfrac{1}{n} \sum_{l=2}^{n} c_2^l r^l \alpha(l) \leqslant \mathcal{O}(\log^3(n) \cdot n^{-\varepsilon'}) + \mathcal{O}(\tfrac{1}{n}),$$

which proves the Theorem by Markov's inequality. ◀

6 Discussion of the Results

In this work, we have shown that with high probability, a power law random k-SAT formula is satisfiable, if $\beta \geqslant \frac{2k-1}{k-1} + \varepsilon$ and the clause-variable ratio is not too large; and that it is unsatisfiable if $\beta \leqslant \frac{2k-1}{k-1} - \varepsilon$, or if the clause-variable ratio is too large. Here, we give a few observations following these results.

First, as explained in Section 1 our results translate directly to the model where clause lengths are power law distributed. This observation might help to explain a phenomenon that arose in [7]: The authors experimentally observed that a random-sat formula with double power law distribution (both variables and clause lengths are drawn from a power law) can be solved extremely fast by MiniSAT. Although the formula was of length $5 \cdot 10^5$, MiniSAT already gave an answer after 4 seconds! Using our results, we are now able to provide a potential explanation for this phenomenon: Disregarding the double power law distribution, the smallest clause length $k_{\min}$ occurring in their generated formulas is one. Thus, there will be $\Theta(n)$ clauses of length one and by Theorem 1.1 the formula is likely unsatisfiable.

Second, we observe a sharp threshold in the sense of Friedgut [20] (for small constraint densities r) for β at the point $\frac{2k-1}{k-1}$. In contrast, it is unclear whether such a sharp threshold exists (and can be analytically derived) for fixed β but variable r. Considering however, that decades of research were dedicated to the same question in the uniform case—an arguably simpler model—it is unlikely that we obtain a satisfying answer any time soon; at least for all k. As in the uniform model, however, it might be more tractable to get sharp thresholds for $k \to \infty$.

References

1 Dimitris Achlioptas, Amin Coja-Oghlan, and Federico Ricci-Tersenghi. On the solution-space geometry of random constraint satisfaction problems. *Random Structures & Algorithms*, 38(3):251–268, 2011.

2 William Aiello, Fan Chung, and Linyuan Lu. A random graph model for power law graphs. *Experimental Mathematics*, 10(1):53–66, 2001.

3 Dan Alistarh, Thomas Sauerwald, and Milan Vojnović. Lock-free algorithms under stochastic schedulers. In *34th Symp. Principles of Distributed Computing (PODC)*, pages 251–260, 2015.

4 Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, and Jordi Levy. The fractal dimension of SAT formulas. In *7th Intl. Joint Conf. Automated Reasoning (IJCAR)*, pages 107–121, 2014.

5 Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, and Jordi Levy. On the classification of industrial SAT families. In *18th Intl. Conf. of the Catalan Association for Artificial Intelligence (CCIA)*, pages 163–172, 2015.

6 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. On the structure of industrial SAT instances. In *15th Intl. Conf. Principles and Practice of Constraint Programming (CP)*, pages 127–141, 2009.

7 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Towards industrial-like random SAT instances. In *21st Intl. Joint Conf. Artificial Intelligence (IJCAI)*, pages 387–392, 2009.

8 Bengt Aspvall, Michael F Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

9 Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

10 Yacine Boufkhad, Olivier Dubois, Yannet Interian, and Bart Selman. Regular random k-SAT: Properties of balanced formulas. *J. Automated Reasoning*, 35(1-3):181–200, 2005.

11 Milan Bradonjic and Will Perkins. On sharp thresholds in random geometric graphs. In *18th Intl. Workshop on Randomization and Computation (RANDOM)*, pages 500–514, 2014.

12 Karl Bringmann, Ralph Keusch, and Johannes Lengler. Geometric inhomogeneous random graphs. *arXiv preprint arXiv:1511.00576*, 2015.

13 Václav Chvatal and Bruce Reed. Mick gets some (the odds are on his side). In *33rd Symp. Foundations of Computer Science (FOCS)*, pages 620–627, 1992.

14 Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

15 Amin Coja-Oghlan and Konstantinos Panagiotou. The asymptotic k-SAT threshold. *Advances in Mathematics*, 288:985–1068, 2016.

16 Wenceslas Fernandez de la Vega. Random 2-SAT: results and problems. *Theoretical Computer Science*, 265(1-2):131–146, 2001.

17 Josep Díaz, Lefteris M. Kirousis, Dieter Mitsche, and Xavier Pérez-Giménez. On the satisfiability threshold of formulas with three literals per clause. *Theoretical Computer Science*, 410(30-32):2920–2934, 2009.

18 Jian Ding, Allan Sly, and Nike Sun. Proof of the satisfiability conjecture for large k. In *47th Symp. Theory of Computing (STOC)*, pages 59–68, 2015.

19 D.P. Dubhashi and A. Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009.

20 Ehud Friedgut. Sharp thresholds of graph properties, and the k-SAT problem. *J. ACM*, 12(4):1017–1054, 1999.

21 Tobias Friedrich, Anton Krohmer, Ralf Rothenberger, Thomas Sauerwald, and Andrew M. Sutton. Bounds on the satisfiability threshold for power law distributed random SAT. *arXiv preprint arXiv:1706.08431*, 2017.

22 Andreas Goerdt. A threshold for unsatisfiability. *J. Computer & System Sciences*, 53(3):469–486, 1996.

23 Mohammad Taghi Hajiaghayi and Gregory B. Sorkin. The satisfiability threshold of random 3-SAT is at least 3.52. Technical Report RC22942, IBM, October 2003.

24 Alexis C. Kaporis, Lefteris M. Kirousis, and Efthimios G. Lalas. The probabilistic analysis of a greedy satisfiability algorithm. *Random Structures & Algorithms*, 28(4):444–480, 2006.

25 Lefteris M Kirousis, Evangelos Kranakis, Danny Krizanc, and Yannis C Stamatiou. Approximating the unsatisfiability threshold of random formulas. *Random Structures & Algorithms*, 12(3):253–269, 1998.

26 Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Phys. Rev. E*, 82:036106, Sep 2010.

27 Colin McDiarmid. On a correlation inequality of Farr. *Combinatorics, Probability & Computing*, 1(02):157–160, 1992.

28 Marc Mézard, Giorgio Parisi, and Riccardo Zecchina. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297(5582):812–815, 2002.

29 M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):pp. 167–256, 2003.

30 Mark EJ Newman. Power laws, pareto distributions and Zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

31 Bo Söderberg. General formalism for inhomogeneous random graphs. *Phys. Rev. E*, 66(6):066121, 2002.

32 Remco van der Hofstad. Random graphs and complex networks. Available at `www.win.tue.nl/~rhofstad/NotesRGCN.pdf`, 2011.

An Encoding for Order-Preserving Matching*

Travis Gagie[1], Giovanni Manzini[2], and Rossano Venturini[3]

1 **School of Computer Science and Telecommunications, Diego Portales University and CeBiB, Santiago, Chile**
travis.gagie@mail.udp.cl

2 **Computer Science Institute, University of Eastern Piedmont, Alessandria, Italy and IIT-CNR, Pisa, Italy**
giovanni.manzini@uniupo.it

3 **Department of Computer Science, University of Pisa, Pisa, Italy and ISTI-CNR, Pisa, Italy**
rossano.venturini@unipi.it

Abstract

Encoding data structures store enough information to answer the queries they are meant to support but not enough to recover their underlying datasets. In this paper we give the first encoding data structure for the challenging problem of order-preserving pattern matching. This problem was introduced only a few years ago but has already attracted significant attention because of its applications in data analysis. Two strings are said to be an order-preserving match if the *relative order* of their characters is the same: e.g., $4, 1, 3, 2$ and $10, 3, 7, 5$ are an order-preserving match. We show how, given a string $S[1..n]$ over an arbitrary alphabet of size σ and a constant $c \geq 1$, we can build an $O(n \log\log n)$-bit encoding such that later, given a pattern $P[1..m]$ with $m \leq \log^c n$, we can return the number of order-preserving occurrences of P in S in $O(m)$ time. Within the same time bound we can also return the starting position of some order-preserving match for P in S (if such a match exists). We prove that our space bound is within a constant factor of optimal if $\log\sigma = \Omega(\log\log n)$; our query time is optimal if $\log\sigma = \Omega(\log n)$. Our space bound contrasts with the $\Omega(n \log n)$ bits needed in the worst case to store S itself, an index for order-preserving pattern matching with no restrictions on the pattern length, or an index for standard pattern matching even with restrictions on the pattern length. Moreover, we can build our encoding knowing only how each character compares to $O(\log^c n)$ neighbouring characters.

1998 ACM Subject Classification E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems, H.3 Information Storage and Retrieval

Keywords and phrases Compact data structures, encodings, order-preserving matching

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.38

1 Introduction

As datasets have grown even faster than computer memories, researchers have designed increasingly space-efficient data structures. We can now store a sequence of n numbers from $\{1, \ldots, \sigma\}$ with $\sigma \leq n$ in about n words, and sometimes $n \log\sigma$ bits or even nH bits, where H is the empirical entropy of the sequence, and still support many powerful queries quickly. If we are interested only in queries of the form "what is the position of the smallest number between

* This work was partially supported by Academy of Finland grant 268324, Fondecyt grant 1171058, PRIN grant 201534HNXC, INdAM-GNCS Project *Efficient algorithms for the analysis of Big Data*.

the ith and jth?", however, we can do even better: regardless of σ or H, we need store only $2n + o(n)$ bits to be able to answer in constant time [21]. Such a data structure, that stores enough information to answer the queries it is meant to support but not enough to recover the underlying dataset, is called an *encoding* [38]. As well as the variant of range-minimum queries mentioned above, there are now efficient encoding data structures for range top-k [14, 24, 27], range selection [34], range majority [35], range maximum-segment-sum [23] and range nearest-larger-value [20] on sequences of numbers, and range-minimum [26] and range nearest-larger-value [30, 31] on two-dimensional arrays of numbers; all of these queries return positions but not values from the sequence or array. Perhaps Orlandi and Venturini's [36] results about sublinear-sized data structures for substring occurrence estimation are the closest to the ones we present in this paper, in that they are more related to pattern matching than range queries: the authors showed how we can store a sequence of n numbers from $\{1, \ldots, \sigma\}$ in fewer than $n \log \sigma$ bits but such that we can estimate quickly how often any pattern occurs in the sequence, with the additive error in our estimate proportional to our compression ratio.

Encoding data structures can offer better space bounds than traditional data structures that store the underlying dataset somehow (even in succinct or compressed form), and possibly even security guarantees: if we can build an encoding data structure using only public information, then we need not worry about it being reverse-engineered to reveal private information. From the theoretical point of view, encoding data structures pose new interesting combinatorial problems and promise to be a challenging field for future research.

In this paper we give the first encoding for *order-preserving pattern matching*, a problem which asks us to search in a text for substrings whose characters have the same relative order as those in a pattern. For example, in $6, 3, 9, 2, 7, 5, 4, 8, 1$, the order-preserving matches of $2, 1, 3$ are $6, 3, 9$ and $5, 4, 8$. Kubica et al. [33] and Kim et al. [32] formally introduced this problem and gave efficient online algorithms for it. Other researchers have continued their investigation, and we briefly survey their results in Section 2. As well as its theoretical interest, this problem has practical applications in data analysis. For example, mining for correlations in large datasets is complicated by amplification or damping – e.g., the euro fluctuating against the dollar may cause the pound to fluctuate similarly a few days later, but to a greater or lesser extent – and if we search only for sequences of values that rise or fall by exactly the same amount at each step we are likely to miss many potentially interesting leads. In such settings, searching for sequences in which only the relative order of the values is constrained to be the same is certainly more robust.

In Section 2 we discuss some previous work on order-preserving pattern matching. In Section 3 we review the algorithmic tools we use in the rest of the paper. In Section 4 we prove our first result showing how, given a string $S[1..n]$ over an arbitrary alphabet $[\sigma]$ and a constant $c \geq 1$, we can store $\mathcal{O}(n \log \log n)$ bits – regardless of σ – such that later, given a pattern $P[1..m]$ with $m < \log^c n$, in $\mathcal{O}(n \log^c n)$ time we can scan our encoding and report all the order-preserving matches of P in S. Our space bound contrasts with the $\Omega(n \log n)$ bits needed in the worst case, when $\log \sigma = \Omega(\log n)$, to store S itself, an index for order-preserving pattern matching with no restriction on the pattern length, or an index for standard pattern matching even with restrictions on the pattern length. (If S is a permutation then we can recover it from an index for unrestricted order-preserving pattern matching, or from an index for standard matching of patterns of length 2, even when they do not report the positions of the matches. Notice this does not contradict Orlandi and Venturini's result, mentioned above, about estimating substring frequency, since that permits additive error.) In fact, we build our representation of S knowing only how each

character compares to $2\log^c n$ neighbouring characters. We show in Section 5 how to adapt and build on this representation to obtain indexed order-preserving pattern matching, instead of scan-based, allowing queries in $\mathcal{O}\big(m\log^3 n\big)$ time but now reporting the position of only one match.

In Section 6 we give our main result showing how to speed up our index using weak prefix search and other algorithmic improvements. The final index is able to *count* the number of occurrences and *return the position* of an order-preserving match (if one exists) in $\mathcal{O}(m)$ time. This query time is optimal if $\log\sigma = \Omega(\log n)$. Finally, in Section 7 we show that our space bound is optimal (up to constant factors) even for data structures that only return whether or not S contains any order-preserving matches.

2 Previous Work

Although recently introduced, order-preserving pattern matching has received considerable attention and has been studied in different settings. For the online problem, where the pattern is given in advance, the first contributions were inspired by the classical Knuth–Morris–Pratt and Boyer–Moore algorithms [5, 12, 32, 33]. The proposed algorithms have guaranteed linear time worst-case complexity or sublinear time average complexity. However, for the online problem the best results in practice are obtained by algorithms based on the concept of filtration, in which some sort of "order-preserving" fingerprint is applied to the text and the pattern [6, 7, 8, 10, 11, 18, 15]. This approach was successfully applied also to the harder problem of matching with errors [8, 25, 28].

There has also been work on indexed order-preserving pattern matching. Crochemore et al. [13] showed how, given a string $S[1..n]$, in $\mathcal{O}\big(n\sqrt{\log n}\big)$ time we can build an $\mathcal{O}(n\log n)$-bit index such that later, given a pattern $P[1..m]$ over an alphabet polynomially bounded in m, we can return the starting positions of all the occ order-preserving matches of P in S in optimal $\mathcal{O}(m+\mathsf{occ})$ time. Their index is a kind of suffix tree, and other researchers [39] are trying to reduce the space bound to $n\log\sigma + o(n\log\sigma)$ bits, where σ is the size of the alphabet of S, by using a kind of Burrows–Wheeler Transform instead (similar to recent work [22] on parameterized pattern matching [1]). Even if they succeed, however, when $\sigma = n^{\Omega(1)}$ the resulting index will still take linear space – i.e., $\Omega(n)$ words or $\Omega(n\log n)$ bits. (It could be interesting to apply the techniques we develop here to parameterized pattern matching, but we leave that as future work.)

In addition to Crochemore et al.'s result, other offline solutions have been proposed combining the idea of fingerprints and indexing. Chhabra et al. [9] showed how to speed up the search by building an FM-index [19] on the binary string expressing whether in the input text each element is smaller or larger than the next one. By expanding this approach, Decaroli et al. [15] show how to build a compressed file format supporting order-preserving matching without the need of full decompression. Experiments show that this compressed file format takes roughly the same space as gzip and that in most cases the search is orders of magnitude faster than the sequential scan of the text. We point out that these approaches, although interesting for the applications, do not have competitive worst case bounds on the search cost as we get from Crochemore et al. and in this paper.

3 Background

In this section we collect a set of algorithmic tools that will be used in our solutions. In the following we report each result together with a brief description of the solved problem. More details can be obtained by consulting the corresponding references. All the results

hold in the unit cost word-RAM model, where each memory word has size $w = \Omega(\log n)$ bits, where n is the input size. In this model arithmetic and boolean operations on memory words require $\mathcal{O}(1)$ time.

Rank queries on binary vector. In the next solutions we will need to support Rank queries on a binary vector $B[1..n]$. Given an index i, Rank(i) on B returns the number of 1s in the prefix $B[1..i]$. We report here a result in [29].

▶ **Theorem 1.** *Given a binary vector $B[1..n]$, we can support* Rank *queries in constant time by using $n + o(n)$ bits of space.*

Elias-Fano representation. In the following we will need to encode an increasing sequence of values in almost optimal space. There are several solutions to this problem, we report here the result obtained with the, so-called, Elias-Fano representation [16, 17].

▶ **Theorem 2.** *An increasing sequence of n values up to u can be represented by using $\log \binom{u}{n} + O(n) = n \log \frac{u}{n} + O(n)$ bits, so that we can access any value of the sequence in constant time.*

Minimal perfect hash functions. In our solution we will make use of *Monotone minimal perfect hash function*s (Mmphf) [2]. Given a set $S = \{x_1, x_2, \ldots, x_n\}$ of size n, a minimal perfect hash function (Mphf) has to injectively map keys in S to the integers in $[n]$.

A monotone minimal perfect hash function (Mmphf) is an Mphf $h()$ that preserves the lexicographic ordering, i.e., for any two strings x and y in the set, $x \leq y$ iff $h(x) \leq h(y)$. Results on Mmphfs have been focused on dictionaries of binary strings [2]. The results can be easily generalized to dictionaries with strings over larger alphabets. The following theorem reports the obvious generalization of Theorem 3.1 in [2] and Theorem 2 in [4].

▶ **Theorem 3.** *Given a dictionary of n strings drawn from the alphabet $[\sigma]$, there is a monotone minimal perfect hash function $h()$ that occupies $\mathcal{O}(n \log(\ell \log \sigma))$ bits of space, where ℓ is the average length of the strings in the dictionary. Given a string $P[1..m]$, $h(P)$ is computed in $\mathcal{O}(1 + m \log \sigma / w)$ time.*

Weak prefix search. The *Prefix Search Problem* is a well-known problem in data-structure design for strings. It asks for the preprocessing of a given set of n strings in such a way that, given a query-pattern P, (the lexicographic range of) all the strings in the dictionary which have P as a prefix can be returned efficiently in time and space.

Belazzougui et al. [4] introduced the weak variant of the problem that allows for a one-sided error in the answer. Indeed, in the *Weak Prefix Search Problem* the answer to a query is required to be correct only in the case that P is a prefix of at least one string in dictionary; otherwise, the algorithm returns an arbitrary answer.

Due to these relaxed requirements, the data structures solving the problem are allowed to use space sublinear in the total length of the indexed strings. Belazzougui et al. [4] focus their attention on dictionaries of binary strings, but their results can be easily generalized to dictionaries with strings over larger alphabets. The following theorem states the obvious generalization of Theorem 5 in [4].

▶ **Theorem 4.** *Given a dictionary of n strings drawn from the alphabet $[\sigma]$, there exists a data structure that weak prefix searches for a pattern $P[1..m]$ in $\mathcal{O}(m \log \sigma / w + \log(m \log \sigma))$ time. The data structure uses $\mathcal{O}(n \log(\ell \log \sigma))$ bits of space, where ℓ is the average length of the strings in the dictionary.*

We remark that the space bound in [4] is better than the one reported above as it is stated in terms of the size the hollow trie, a conceptual tool introduced in [3], associated to the indexed dictionary. This measure is always within $\mathcal{O}(n \log(\ell \log \sigma))$ bits but it may be much better depending on the dictionary. However, the weaker space bound suffices for the aims of this paper.

4 An Encoding for Scan-Based Search

As an introduction to our techniques, we show an $\mathcal{O}(n \log \log n)$ bit encoding supporting scan-based order-preserving matching. Given a sequence $S[1..n]$ we define the rank encoding $E(S)[1..n]$ as

$$E(S)[i] = \begin{cases} 0.5 & \text{if } S[i] \text{ is lexicographically smaller than any character in } \{S[1], \ldots, S[i-1]\}, \\ j & \text{if } S[i] \text{ is equal to the lexicographically } j\text{th character in } \{S[1], \ldots, S[i-1]\}, \\ j + 0.5 & \text{if } S[i] \text{ is larger than the lexicographically } j\text{th character in } \{S[1], \ldots, S[i-1]\} \text{ but smaller than the lexicographically } (j+1)\text{st}, \\ |\{S[1], \ldots, S[i-1]\}| + 0.5 & \text{if } S[i] \text{ is lexicographically larger than any character in } \{S[1], \ldots, S[i-1]\}. \end{cases}$$

This is similar to the representations used in previous papers on order-preserving matching. We can build $E(S)$ in $\mathcal{O}(n \log n)$ time. However, we would ideally need $E(S[i..n])$ for $i = 1, \ldots, n$, since $P[1..m]$ has an order-preserving match in $S[i..i+m-1]$ if and only if $E(P) = E(S[i..i+m-1])$. Assuming P has polylogarithmic size, we can devise a more space efficient encoding.

▶ **Lemma 5.** *Given $S[1..n]$ and a constant $c \geq 1$, let $\ell = \log^c n$. We can store $\mathcal{O}(n \log \log n)$ bits such that later, given i and $m \leq \ell$, we can compute $E(S[i..i+m-1])$ in $\mathcal{O}(m)$ time.*

Proof. For every position i in S which is a multiple of $\ell = \log^c n$, we store the ranks of the characters in the window $S[i..i+2\ell]$. The ranks are values at most $2\ell + 1$, thus they are stored in $\mathcal{O}(\log \ell)$ bits each. We concatenate the ranks of each window in a vector V, which has length $\mathcal{O}(n)$ and takes $\mathcal{O}(n \log \ell)$ bits. Every range $S[i..i+m-1]$ of length $m \leq \ell$ is fully contained in at least one window and in constant time we can convert i into i' such that $V[i'..i'+m-1]$ contains the ranks of $S[i], \ldots, S[i+m-1]$ in that window.

Computing $E(S[i..i+m-1])$ naïvely from these ranks would take $\mathcal{O}(m \log m)$ time. We can speed up this computation by exploiting the fact that $S[i..i+m-1]$ has polylogaritmic length. Indeed, a recent result [37] introduces a data structure to represent a small dynamic set $\mathcal{S}$ of $\mathcal{O}(w^c)$ integers of w bits each supporting, among the others, insertions and rank queries in $\mathcal{O}(1)$ time. Given an integer x, the rank of x is the number of integers in $\mathcal{S}$ that are smaller than or equal to x. All operations are supported in constant time for sets of size $\mathcal{O}(w^c)$. This result allows us to compute $E(S[i..i+m-1])$ in $\mathcal{O}(m)$ time. Indeed, we can use the above data structure to insert $S[i..i+m-1]$'s characters one after the other and compute their ranks in constant time. ◀

It follows from Lemma 5 that given S and c, we can store an $\mathcal{O}(n \log \log n)$-bit encoding of S such that later, given a pattern $P[1..m]$ with $m \leq \log^c n$, we can compute $E(S[i..i+m-1])$

for each position i in turn and compare it to $E(P)$, and thus find all the order-preserving matches of P in $\mathcal{O}(nm)$ time. (It is possible to speed this scan-based algorithm up by avoiding computing each $E(S[i..i+m-1])$ from scratch but, since this is only an intermediate result, we do not pursue it further here.) We note that we can construct the encoding in Lemma 5 knowing only how each character of S compares to $\mathcal{O}(\log^c n)$ neighbouring characters.

▶ **Corollary 6.** *Given $S[1..n]$ and a constant $c \geq 1$, we can store an encoding of S in $\mathcal{O}(n \log\log n)$ bits such that later, given a pattern $P[1..m]$ with $m \leq \log^c n$, we can find all the order-preserving matches of P in S in $\mathcal{O}(nm)$ time.*

We will not use Corollary 6 in the rest of this paper, but we state it as a baseline easily proven from Lemma 5.

5 Adding an Index to the Encoding

Suppose we are given $S[1..n]$ and a constant $c \geq 1$. We build the $\mathcal{O}(n \log\log n)$-bit encoding of Lemma 5 for $\ell = \log^c n + \log n$ and call it S_ℓ. Using S_ℓ we can compute $E(S')$ for any substring S' of S of length $|S'| \leq \ell$ in $\mathcal{O}(|S'|)$ time. We now show how to complement S_ℓ with a kind of "sampled suffix array" using $\mathcal{O}(n \log\log n)$ more bits, such that we can search for a pattern $P[1..m]$ with $m \leq \log^c n$ and return the starting position of an order-preserving match for P in S, if there is one. Our first solution has $\mathcal{O}\big(m \log^3 n\big)$ query time; we will improve the query time to $\mathcal{O}(m)$ in the next section.

We define the rank-encoded suffix array $R[1..n]$ of S such that $R[i] = j$ if $E(S[j..n])$ is the lexicographically ith string in $\{E(S[1..n]), E(S[2..n]), \ldots, E(S[n..n])\}$. Note that $E(S[i..n])$ has length $n-i+1$. Figure 1 shows an example. Our algorithm consists of a *searching phase* followed by a *verification phase*. The goal of the searching phase is to identify a range $[l, r]$ in R which contains all the encodings prefixed by $E(P)$, if any, or an arbitrary interval if P does not occur. The verification phase has to check if there is at least one occurrence of P in this interval, and return a position at which P occurs.

Searching phase. Similarly to how we can use a normal suffix array and S to support normal pattern matching, we could use R and S to find all order-preserving matches for a pattern $P[1..m]$ in $\mathcal{O}(m \log n)$ time via binary search, i.e., at each step we choose an index i, extract $S[R[i]..R[i]+m-1]$, compute its rank encoding and compare it to $E(P)$, all in $\mathcal{O}(m)$ time. If $m \leq \ell$ we can compute $E(S[R[i]..R[i]+m-1])$ using S_ℓ instead of S, still in $\mathcal{O}(m)$ time, but storing R still takes $\Omega(n \log n)$ bits.

Therefore, for our searching phase we sample and store only every d-th element of R, by position, and every element of R equal 1 or n or a multiple of d, where $d = \lfloor \log n / \log\log n \rfloor$. This takes $\mathcal{O}(n \log\log n)$ bits. Notice we can still find in $\mathcal{O}(m \log n)$ time via binary search in the sampled R an order-preserving match for any pattern $P[1..m]$ that has at least d order-preserving matches in S. If P has fewer than d order-preserving matches in S but we happen to have sampled a cell of R pointing to the starting position of one of those matches, then our binary search still finds it. Otherwise, we find an interval of length at most $d-1$ which contains pointers at least to all the order-preserving matches for P in S; on this interval we perform the verification phase.

Verification phase. The verification phase receives a range $R[l, r]$ (although R is not stored completely) and has to check if that range contains the starting position of an order preserving match for P and, if so, return its position. This is done by adding auxiliary data structures to the sampled entries of R.

i	$R[i]$	$L[i]$	$B[i]$	$D[i]$	$E(S[R[i]..n])$
1	**30**				0.5
2	29	**2**	**1.5**	**4**	0.5 0.5
3	22	**2**	**0.5**	**2**	0.5 0.5 0.5 0.5 1.5 5 5.5 6.5 1
4	**13**				0.5 0.5 0.5 1 0.5 1.5 4 4.5 1 4 3.5 3.5 2 3 6 7 7.5 2
5	2	**2**	**0.5**	**1**	0.5 0.5 0.5 1.5 2.5 3.5 5.5 2.5 2 5 4 8 4 1 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 8 2
6	23	**3**	**3.5**	**3**	0.5 0.5 0.5 1.5 4.5 5.5 6.5 1
7	**8**				0.5 0.5 0.5 2.5 2.5 5.5 3 0.5 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2
8	**14**				0.5 0.5 1 0.5 1.5 4 4.5 1 4 3.5 3.5 2 3 6 7 7.5 2
9	**20**				0.5 0.5 1.5 1.5 1.5 1.5 2.5 6 7 7.5 2
10	3	**3**	**3.5**	**1**	0.5 0.5 1.5 2.5 3.5 5.5 2.5 2 5 4 7.5 4 1 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 8 2
11	**16**				0.5 0.5 1.5 3.5 4.5 1 4 3.5 3.5 2 3 6 7 7.5 2
12	**24**				0.5 0.5 1.5 3.5 4.5 5.5 1
13	11	**2**	**0.5**	**3**	0.5 0.5 2.5 1 0.5 1 0.5 1.5 4 5 1 4 3.5 3.5 2 3 6 7 7.5 2
14	9	**3**	**3.5**	**3**	0.5 0.5 2.5 2.5 4.5 3 0.5 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2
15	15	**2**	**1.5**	**1**	0.5 1 0.5 1.5 3.5 4.5 1 4 3.5 3.5 2 3 6 7 7.5 2
16	**28**				0.5 1.5 0.5
17	7	**3**	**1.5**	**4**	0.5 1.5 0.5 0.5 3 2.5 5.5 3 0.5 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2
18	19	**3**	**1.5**	**5**	0.5 1.5 0.5 2 1.5 1.5 1.5 2.5 6 7 7.5 2
19	**12**				0.5 1.5 1 0.5 1 0.5 1.5 4 4.5 1 4 3.5 3.5 2 3 6 7 7.5 2
20	**1**				0.5 1.5 1.5 0.5 2 2.5 3.5 5.5 2.5 2 5 4 8 4 1 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 8 2
21	21	**2**	**2.5**	**1**	0.5 1.5 1.5 1.5 1.5 2.5 6 6.5 7.5 2
22	10	**2**	**1.5**	**2**	0.5 1.5 1.5 3.5 2 0.5 1 0.5 1.5 5 6 1 5 4.5 4 2 3 6 7 7.5 2
23	27	**4**	**1.5**	**2**	0.5 1.5 2.5 0.5
24	**6**				0.5 1.5 2.5 0.5 0.5 4 3 5.5 3 0.5 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2
25	18	**4**	**1**	**3**	0.5 1.5 2.5 0.5 3 2.5 2.5 2 2.5 6 7 7.5 2
26	26	**4**	**0.5**	**1**	0.5 1.5 2.5 3.5 0.5
27	17	**2**	**2.5**	**3**	0.5 1.5 2.5 3.5 1 3 2.5 2.5 2 2.5 6 7 7.5 2
28	**5**				0.5 1.5 2.5 3.5 1.5 1 4 3 5.5 3 0.5 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2
29	25	**2**	**2.5**	**4**	0.5 1.5 2.5 3.5 4.5 1
30	**4**				0.5 1.5 2.5 3.5 4.5 2.5 2 5 4 6.5 4 1 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2

Figure 1 The rank-encoded suffix array $R[1..30]$ for $S[1..30] = 3\,9\,7\,2\,3\,5\,6\,8\,4\,3\,6\,5\,9\,5\,2\,2\,0\,1\,5\,6\,0\,5\,4\,3\,1\,2\,5\,6\,7\,1$, with $L[i]$, $B[i]$ and $D[i]$ computed for $d = 4$. Stored values are shown in boldface.

Suppose that for each unsampled element $R[i] = j$ we store the following data.

- the smallest number $L[i]$ (if one exists) such that $S[j-1..j+L[i]-1]$ has at most d order-preserving matches in S;
- the rank $B[i] = E(S[j-1..j+L[i]-1]^{\text{rev}})[L[i]+1] \leq L[i]+1/2$ of $S[j-1]$ in $S[j..j+L[i]-1]$, where the superscript rev indicates that the string is reversed;
- the distance $D[i]$ to the cell of R containing $j-1$ from the last sampled element x such that $E(S[x..x+L[i]])$ is lexicographically smaller than $E(S[j-1..j+L[i]-1])$.

Figure 1 shows the values in L, B and D for our example.

Assume we are given $P[1..m]$ and i and told that $S[R[i]..R[i]+m-1]$ is an order-preserving match for P, but we are not told the value $R[i] = j$. If $R[i]$ is sampled, of course, then we can return j immediately. If $L[i]$ does not exist or is greater than m then P has at least d order-preserving matches in S, so we can find one in $\mathcal{O}(m)$ time: we consider the sampled values from R that precede and follow $R[i]$ and check with Lemma 5 whether there are order-preserving matches starting at those sampled values. Otherwise, from $L[i]$, $B[i]$ and P, we can compute $E(S[j-1..j+L[i]-1])$ in $\mathcal{O}(m \log m)$ time: we take the length-$L[i]$ prefix of P; if $B[i]$ is an integer, we prepend to $P[1..L[i]]$ a character equal to

the lexicographically $B[i]$th character in that prefix; if $B[i]$ is $r + 0.5$ for some integer r with $1 \leq r < L[i]$, we prepend a character lexicographically between the lexicographically rth and $(r + 1)$st characters in the prefix; if $B[i] = 0.5$ or $B[i] = L[i] + 0.5$, we prepend a character lexicographically smaller or larger than any in the prefix, respectively. We can then find in $\mathcal{O}(m \log n)$ time the position in R of x, the last sampled element such that $E(S[x..x + L[i]])$ is lexicographically smaller than $E(S[j - 1..j + L[i] - 1])$. Adding $D[i]$ to this position gives us the position i' of $j - 1$ in R. Repeating this procedure until we reach a sampled cell of R takes $\mathcal{O}\big(m \log^2 n / \log \log n\big) = \mathcal{O}\big(m \log^2 n\big)$ time, and we can then compute and return j. As the reader may have noticed, the procedure is very similar to how we use backward stepping to locate occurrences of a pattern with an FM-index [19], so we refer to it as a backward step at position i.

Even if we do not really know whether $S[R[i]..R[i] + m - 1]$ is an order-preserving match for P, we can still start at the cell $R[i]$ and repeatedly apply this procedure: if we do not find a sampled cell after $d - 1$ repetitions, then $S[R[i]..R[i] + m - 1]$ is not an order-preserving match for P; if we do, then we add the number of times we have repeated the procedure to the contents of the sampled cell to obtain the contents of $R[i] = j$. Then, using S_ℓ we compute $E(S[j..j + m - 1])$ in $\mathcal{O}(m)$ time, compare it to $E(P)$ and, if they are the same, return j. This still takes $\mathcal{O}\big(m \log^2 n\big)$ time. Therefore, after our searching phase, if we find an interval $[l, r]$ of length at most $d - 1$ which contains pointers to all the order-preserving matches for P in S (instead of an order-preserving match directly), then we can check each cell in that interval with this procedure, in a total of $\mathcal{O}\big(m \log^3 n\big)$ time.

If $R[i] = j$ is the starting position of an order-preserving match for a pattern $P[1..m]$ with $m \leq \log^c n$ that has at most d order-preserving matches in S, then $L[i] \leq \log^c n$. Moreover, if $R[i'] = j - 1$ then $L[i'] \leq \log^c n + 1$ and, more generally, if $R[i''] = j - t$ then $L[i''] \leq \log^c n + t$. Therefore, we can repeat the stepping procedure described above and find j without ever reading a value in L larger than $\log^c n + \log n$ and, since each value in B is bounded in terms of the corresponding value in L, without ever reading a value in B larger than $\log^c n + \log n + 1/2$. It follows that we can replace any values in L and B greater than $\log^c n + \log n + 1/2$ by the flag -1, indicating that we can stop the procedure when we read it. With this modification, each value in L and B takes $\mathcal{O}(\log \log n)$ bits, so L, B and D take a total of $\mathcal{O}(n \log \log n)$ bits. Since also the encoding S_ℓ from Lemma 5 with $\ell = \log^c n + \log n$ takes $\mathcal{O}(n \log \log n)$ bits, the following intermediate theorem summarizes our results so far.

▶ **Theorem 7.** *Given $S[1..n]$ and a constant $c \geq 1$, we can store an encoding of S in $\mathcal{O}(n \log \log n)$ bits such that later, given a pattern $P[1..m]$ with $m \leq \log^c n$, in $\mathcal{O}\big(m \log^3 n\big)$ time we can return the position of an order-preserving match of P in S (if one exists).*

A complete search example. Suppose we are searching for order-preserving matches for $P = 2\,3\,1\,2$ in the string $S[1..30]$ shown in Figure 1. Binary search on R tells us that pointers to all the matches are located in R strictly between $R[16] = 28$ and $R[19] = 12$, because

$$\begin{aligned} E(S[28..30]) = E(6\,7\,1) = 0.5\ 1.5\ 0.5 &\prec E(P) = E(2\,3\,1\,2) = 0.5\ 1.5\ 0.5\ 2 \\ &\prec E(S[12..14]) = E(5\,9\,5) = 0.5\ 1.5\ 1\,; \end{aligned}$$

notice $R[16] = 28$ and $R[19] = 12$ are stored because 16, 28 and 12 are multiples of $d = 4$.

We first check whether $R[17]$ points to an order-preserving match for P. That is, we assume (incorrectly) that it does; we take the first $L[17] = 3$ characters of P; and, because $B[17] = 1.5$, we prepend a character between the lexicographically first and second, say 1.5. This gives us $1.5\,2\,3\,1$, whose encoding is $0.5\,1.5\,2.5\,0.5$. Another binary search on R

shows that $R[20] = 1$ is the last sampled element x such that $E(S[x..x+3])$, in this case $0.5\,1.5\,1.5\,0.5$, is lexicographically smaller than $0.5\,1.5\,2.5\,0.5$. Adding $D[17] = 4$ to 20, we would conclude that $R[24] = R[17] - 1$ (which happens to be true in this case) and that $0.5\,1.5\,2.5\,0.5$ is a prefix of $E(S[R[24]..n])$ (which also happens to be true). Since $R[24] = 6$ is sampled, however, we compute $E(S[7..10]) = 0.5\,1.5\,0.5\,0.5$ and, since it is not the same as P's encoding, we reject our initial assumption that $R[17]$ points to an order-preserving match for P.

We now check whether $R[18]$ points to an order preserving match for P. That is, we assume (correctly this time) that it does; we take the first $L[18] = 3$ characters of P; and, because $B[18] = 1.5$, we prepend a character between the lexicographically first and second, say 1.5. This again gives us $1.5\,3\,2\,1$, whose encoding is $0.5\,1.5\,2.5\,0.5$. As before, a binary search on R shows that $R[20] = 1$ is the last sampled element x such that $E(S[x..x+3])$ is lexicographically smaller than $0.5\,1.5\,2.5\,0.5$. Adding $D[18] = 5$ to 20, we conclude (correctly) that $R[25] = R[18] - 1$ and that $0.5\,1.5\,2.5\,0.5$ is a prefix of $E(S[R[25]..n])$

Repeating this procedure with $L[25] = 4$, $B[25] = 1$ and $D[25] = 3$, we build a string with encoding $0.5\,1.5\,2.5\,0.5$, say $2\,3\,4\,1$, and prepend a character equal to the lexicographically first, 1. This gives us $1\,2\,3\,4\,1$, whose encoding is $0.5\,1.5\,2.5\,3.5\,1$. Another binary search shows that $R[24] = 6$ is the last sampled element x such that $E(S[x..x+4])$ is lexicographically smaller than $0.5\,1.5\,2.5\,3.5\,1$. We conclude (again correctly) that $R[27] = R[18] - 2$ and that $0.5\,1.5\,2.5\,3.5\,1$ is a prefix of $E(S[R[27]..n])$.

Finally, repeating this procedure with $L[27] = 2$, $B[27] = 2.5$ and $D[27] = 3$, we build a string with encoding $0.5\,1.5$, say $1\,2$, and prepend a character lexicographically greater than any currently in the string, say 3. This gives us $3\,1\,2$, whose encoding is $0.5\,0.5\,1.5$. A final binary search show that $R[8] = 14$ is the last sampled element x such that $E(S[x..x+2])$ is lexicographically smaller than $0.5\,0.5\,1.5$. We conclude (again correctly) that $R[11] = R[18] - 3$ and that $0.5\,0.5\,1.5$ is a prefix of $E(S[R[11]..n])$. Since $R[11] = 16$ is sampled, we compute $E(S[19..22]) = 0.5\,1.5\,0.5\,2$ and, since it matches P's encoding, we indeed report $S[19..22]$ as an order-preserving match for P.

6 Achieving $\mathcal{O}(m)$ query time

In this section we prove our main result:

▶ **Theorem 8.** *Given $S[1..n]$ and a constant $c \geq 1$, we can store an encoding of S in $\mathcal{O}(n \log\log n)$ bits such that later, given a pattern $P[1..m]$ with $m \leq \log^c n$, in $\mathcal{O}(m)$ time we can return the position of an order-preserving match of P in S (if one exists). In $\mathcal{O}(m)$ time we can also report the total number of order-preserving occurrences of P in S.*

Compared to Theorem 7, we improve the query time from $\mathcal{O}(m \log^3 n)$ to $\mathcal{O}(m)$. This is achieved by speeding up several steps of the algorithm described in the previous section.

Speeding up pattern's encoding. Given a pattern $P[1..m]$, the algorithm has to compute its encoding $E(P[1..m])$. Doing this naïvely as in the previous section would cost $\mathcal{O}(m \log m)$ time, which is, by itself, larger than our target time complexity. However, since m is polylogarithmic in n, we can speed this up as we sped up the computation of the rank-encoding of $S[i..i+m-1]$ in the proof of Lemma 5, and obtain $E(P)$ in $\mathcal{O}(m)$ time. Indeed, we can insert P's characters one after the other in the data structures of [37] and compute their ranks in constant time.

Dealing with short patterns. The approach used by our solution cannot achieve a $o(d)$ query time. This is because we answer a query by performing $\Theta(d)$ backward steps regardless of the pattern's length. This means that for very short patterns, namely $m = o(d) = o(\log n/\log\log n)$, the solution cannot achieve $\mathcal{O}(m)$ query time. However, we can precompute and store the answers of all these short patterns in $o(n)$ bits. Indeed, we can find a constant c such that the encoding of a pattern of length at most $c\log n/\log\log n$ is a binary string of length at most $\frac{1}{2}\log n$ bits. Thus, there are $\mathcal{O}(\sqrt{n})$ possible encodings. For each of these encodings we explicitly store the number of its occurrences and the position of one of them in $o(n)$ bits. From now on, thus, we can safely assume that $m = \Omega(\log n/\log\log n)$.

Speeding up searching phase. The searching phase of the previous algorithm has two important drawbacks. First, it costs $\mathcal{O}(m\log n)$ time and, thus, it is obviously too expensive for our target time complexity. Second, binary searching on the sampled entries in R gives too imprecise results. Indeed, it finds a range $[l, r]$ of positions in R which may be potential matches for P. However, if the entire range is within two consecutive sampled positions, we are only guaranteed that all the occurrences of P are in the range but there may exist positions in the range which do not match P. This uncertainty forces us to explicitly check *every* single position in the range until a match for P is found, if any. This implies that we have to check $r - l + 1 = \mathcal{O}(d)$ positions in the worst case. Since every check has a cost proportional to m, this gives $\omega(m)$ query time.

We use the data structure for weak prefix search of Theorem 4 to index the encodings of all suffixes of the text truncated at length $\ell = \log^c n + \log n$. This way, we can find the range $[l, r]$ of suffixes prefixed by $E(P[1..m])$ in $\mathcal{O}(m\log\log n/w + \log(m\log\log n)) = \mathcal{O}(m\log\log n/w + \log\log n)$ time with a data structure of size $\mathcal{O}(n\log\log n)$ bits. This is because $E(P[1..m])$ is drawn from an alphabet of size $\mathcal{O}(\log^c n)$, and both m and ℓ are in $\mathcal{O}(\log^c n)$. Apart from its faster query time, this solution has stronger guarantees. Indeed, if the pattern P has at least one occurrence, the range $[l, r]$ contains all and only the occurrences of P. Instead, if the pattern P does not occur, $[l, r]$ is an arbitrary and meaningless range. In both cases, just a single check of any position in the range is enough to answer the order-preserving query. This property gives an $\mathcal{O}(\log n/\log\log n)$ factor improvement over the previous solution.

Speeding up verification phase. It is clear by the discussion above that the verification phase has to check only one position in the range $[l, r]$. If the range contains at least one sampled entry of R, we are done. Otherwise, we have to perform at most d backward steps.

We now improve the computation of every single backward step. Assume we have to perform a backward step at i, where $R[i] = j$. Before performing the backward step, we have to compute the encoding $E(S[j-1..j+L[i]-1])$ given $E(S[j..j+u])$, for some value of $L[i]$ and u with $u \geq L[i]$. Our goal is to do this in $\mathcal{O}(1 + m\log\log n/w)$ time. Notice that removing symbols at the end of $S[j..j+u]$ does not change the encoding of the remaining symbols. However, after the insertion of $S[j-1]$ the encoding of $S[k]$ in $S[j-1..j+L[i]-1]$ either does not change, if $S[j-1] \geq S[k]$, or has to be increased by one, if $S[j-1] > S[k]$. The main issue is that we have to process $O(w/\log\log n)$ symbols in parallel. To this end, apart from $E(S[j-1..j+L[i]-1])$, we also keep a different encoding $R(S[j..j+u])$ for $S[j..j+u]$. The encoding R stores $\mathcal{O}(\log\log n)$-bit ranks which represent the relative order among symbols in $S[j..j+u]$. More precisely, for any two symbols $S[k]$ and $S[k']$, $R(S[j..j+u])[k] < R(S[j..j+u])[k']$ iff $S[k] < S[k']$. Notice that we are not constraining these ranking values to form a consecutive interval, i.e., there may be missing values.

Our goal is to compute $R(S[j-1..j+L[i]-1])$ from $R(S[j..j+u])$ as we perform backward steps. For this reason, we no longer store the value $B[i]$ as in the previous solution. Instead, we store the values $P[i]$ and $O[i]$ which are the positions of the predecessor and an occurrence of $S[j-1]$ in $S[j..j+L[i]-1]$, if any. This way, we can compute $R(S[j-1..j+L[i]-1])$ by prepending an appropriate rank r for symbol $S[j-1]$. It is $r = R(S[j..L[i]-1])[O[i]]$, if there already exists an occurrence of $S[j-1]$ in $S[j..L[i]-1]$, or $r = R(S[j..L[i]-1])[P[i]]+1$, otherwise. In the latter case, we increase any value in $R(S[j..L[i]-1])$ which is larger than or equal to r to guarantee that there is no collision with the assigned rank. This can be done in $\mathcal{O}(1+L[i]\log\log n/w) = \mathcal{O}(1+m\log\log n/w)$ time by exploiting word parallelism. We observe that the positions with a rank larger than r are exactly the positions that we need to increase by one in order to compute $E(S[j-1..j+L[i]-1])$. The backward step at i is $i' = k + D[i]$, where k is the sampled entry in R whose encoding has the prefix of length $L[i]$ which is the largest prefix which is (lexicographically) smaller than or equal to $E(S[j-1..j+L[i]-1])$. Notice that equality may occur only for at most one prefix as otherwise $S[j-1..j+L[i]-1]$ would occur more than d times.

To compute k, given i and $E(S[j-1..j+L[i]-1])$, we observe that $E(S[j-1..j+L[i]-1])$ depends only on S and $L[i]$ and not on the pattern P we are searching for. Thus, there exists just one valid $E(S[j-1..j+L[i]-1])$ that could be used at query time for a backward step at i. Notice that, if the pattern P does not occur, the encoding that will be used at i may be different, but in this case it is not necessary to compute a correct backward step. Consider the set $\mathcal{E}$ of all these, at most n, encodings. The goal is to map each encoding in $\mathcal{E}$ to its corresponding sampled entry in R. This can be done as follows. We build a monotone minimal perfect hash function $h()$ on $\mathcal{E}$ to map each encoding to its lexicographic rank. Obviously, the encodings to be mapped to a certain sampled entry i in R form a consecutive range in the lexicographic ordering. Moreover, none of these ranges overlap. Thus, we can use a binary vector B to mark each of these ranges, so that, given the lexicographic rank of an encoding, we can infer its closest sampled entry. The binary vector is obtained by processing the sampled entries in R in lexicographic order and by writing the size of its range in unary. It is easy to see that the sampled entry prefixed by $x = E(S[j-1..j+L[i]-1])$ can be computed as $\mathsf{Rank}_1(h(x))$ in constant time. The data structure that stores B and supports Rank requires $\mathcal{O}(n)$ bits (see Theorem 1).

Since the evaluation of $h()$ is the dominant cost, a backward step takes $\mathcal{O}(1+m\log\log n/w)$ time. The overall space usage of this solution is $\mathcal{O}(n\log\log n)$ bits, because B has at most $2n$ bits and $h()$ requires $\mathcal{O}(n\log\log n)$ bits by Theorem 3. Since we perform at most d backward steps, the overall query time is $\mathcal{O}(d\times(1+m\log\log n/w) = \mathcal{O}(m)$. The equality follows by observing that $d = \mathcal{O}(\log n/\log\log n)$, $m = \Omega(\log n/\log\log n)$ and $w = \Omega(\log n)$.

Query algorithm. We report here the query algorithm for a pattern $P[1..m]$, with $m = \Omega(\log n/\log\log n)$. Recall that for shorter patterns we store all possible answers.

We first compute $E(P[1..m])$ in $\mathcal{O}(1+m\log\log n/w)$ time. Then, we perform a weak prefix search to identify the range $[l,r]$ of encodings that are prefixed by $E(P[1..m])$ in $\mathcal{O}(m\log\log n/w+\log\log n)$ time. If P has at least one occurrence, the search is guaranteed to find the correct range; otherwise, the range may be arbitrary but the subsequent check will identify the mistake and report zero occurrences.

In the checking phase there are two possible cases. If $[l,r]$ contains a sampled entry, say i, in R we use the encoding from Lemma 5 to compare $E(S[R[i]..R[i]+m-1])$ and $E(P[1..m])$ in $\mathcal{O}(m)$ time. If they are equal, we report $R[i]$; otherwise, we are guaranteed

that there is no occurrence of P in S. If $[l, r]$ contains no sampled entry we arbitrarily select an index $i \in [l, r]$ and we perform a sequence of backward steps starting from i. If P has at least one occurrence, we are guaranteed to find a sampled entry e in at most d backward steps. The overall time of these backward steps is $\mathcal{O}(d \times m \log \log n/w) = \mathcal{O}(m)$. If e is not found, we conclude that P has no occurrence. Otherwise, we explicitly compare $E(S[R[e] + b..R[e] + m + b - 1])$ and $E(P[1..m])$ in $\mathcal{O}(m)$ time, where b is the number of performed backward steps. We report $R[e] + b$ only in case of a successful comparison. Note that if P occurs, then the number of its occurrences is $r - l + 1$.

7 Space Lower Bound

In this section we prove that our solution is space optimal. This is done by showing a lower bound on the space that any data structure must use to solve the easier problem of just establishing if a given pattern P has at least one order-preserving occurrence in S.

▶ **Theorem 9.** *For any n, for any σ such that $\log \sigma = \Omega(\log \log n)$, and for any encoding data structure that, given a pattern $P[1..m]$ with $m = \log n$, establishes if P has any order-preserving occurrence in a given string, there exists a string $S[1..n]$ over the alphabet $[\sigma]$ such that the encoding must use $\Omega(n \log \log n)$ bits of space.*

By contradiction assume there exists a data structure D that uses $o(n \log \log n)$ bits. We prove this implies we can store any string $S[1..n]$ in less than $n \log \sigma$ bits, which is impossible. We start by splitting S into n/m blocks of size $m = \log n$. Let B_i denote the ith block. Observe that if we know the set $L(B_i)$ of characters that occur in B_i, we can recover B_i. This is because $E(B_i)$ implicitly tells us how to permute the characters in $L(B_i)$ to obtain B_i. Obviously, if we are able to reconstruct each B_i, we can reconstruct S. Thus, our goal is to use D together with additional data structures to obtain $E(B_i)$ and $L(B_i)$, for any B_i.

We encode $L(B_i)$ for each i by encoding the sorted sequence of characters with the Elias-Fano representation. By Theorem 2, we know that this requires $\ell \log \frac{\sigma}{\ell} + \mathcal{O}(\ell)$ bits, where $\ell = |L(B_i)| \leq m$. If $\sigma \geq m$, this is at most $m \log \frac{\sigma}{m} + \mathcal{O}(m)$ bits. Summing over all the blocks, the overall space is at most $n \log \frac{\sigma}{m} + \mathcal{O}(n)$ bits. If $\sigma < m$, the representation uses $\mathcal{O}(m)$ bits per block and, thus, $\mathcal{O}(n)$ bits overall.

To represent the encodings of all the blocks, consider the set $\mathcal{E}$ of the encodings of all the substrings of S of length m. We do not store $\mathcal{E}$ because it would require too much space. Instead, for each block B_i, we store the lexicographic rank of B_i in $\mathcal{E}$. This way, we are keeping track of those elements in $\mathcal{E}$ that are blocks and their positions in S. This requires $\mathcal{O}(n)$ bits, because there are $n/\log n$ blocks and storing each rank needs $\mathcal{O}(\log n)$ bits.

We are now ready to retrieve the encoding of all the blocks. This is done by searching in D for every possible encoding of exactly m characters. The data structure will tell us the ones that occur in S, i.e., we are retrieving the entire set $\mathcal{E}$. Thus, we sort $\mathcal{E}$ and replace each the stored rank of each block with its original encoding. Thus, we are able to reconstruct S by using D and additional data structures which uses at most $\max(0, n \log \sigma - n \log \log n) + \mathcal{O}(n)$ bits of space. This implies that D cannot use $o(n \log \log n)$ bits.

Acknowledgements. We thank Djamal Belazzougui, Paweł Gawrychowski, Gonzalo Navarro, Patrick Nicholson and Rajeev Raman for helpful discussions. Parts of this work were done while the first author visited the University of Eastern Piedmont and during Dagstuhl Seminar 16431, "Computation over Compressed Structured Data".

References

1 Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.

2 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, pages 785–794. SIAM, 2009.

3 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Theory and practise of monotone minimal perfect hashing. In *Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 132–144. SIAM, 2009.

4 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 427–438. Springer, 2010.

5 Djamal Belazzougui, Adeline Pierrot, Mathieu Raffinot, and Stéphane Vialette. Single and multiple consecutive permutation motif search. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*, pages 66–77. Springer, 2013.

6 Domenico Cantone, Simone Faro, and M. Oğuzhan Külekci. An efficient skip-search approach to the order-preserving pattern matching problem. In *Proceedings of the Prague Stringology Conference (PSC)*, pages 22–35. Department of Theoretical Computer Science, Czech Technical University in Prague, 2015.

7 Tamanna Chhabra, Simone Faro, M. Oğuzhan Külekci, and Jorma Tarhio. Engineering order-preserving pattern matching with SIMD parallelism. *Software: Practice and Experience*, 2016.

8 Tamanna Chhabra, Emanuele Giaquinta, and Jorma Tarhio. Filtration algorithms for approximate order-preserving matching. In *Proceedings of the Symposium on String Processing and Information Retrieval (SPIRE)*, pages 177–187. Springer, 2015.

9 Tamanna Chhabra, M. Oğuzhan Külekci, and Jorma Tarhio. Alternative algorithms for order-preserving matching. In *Proceedings of the Prague Stringology Conference (PSC)*, pages 36–46. Department of Theoretical Computer Science, Czech Technical University in Prague, 2015.

10 Tamanna Chhabra and Jorma Tarhio. Order-preserving matching with filtration. In *Proceedings of the Symposium on Experimental Algorithms (SEA)*, pages 307–314. Springer, 2014.

11 Tamanna Chhabra and Jorma Tarhio. A filtration method for order-preserving matching. *Information Processing Letters*, 116(2):71–74, 2016.

12 Sukhyeun Cho, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. A fast algorithm for order-preserving pattern matching. *Information Processing Letters*, 115(2):397–402, 2015.

13 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Order-preserving indexing. *Theoretical Computer Science*, 638:122–135, 2016.

14 Pooya Davoodi, Gonzalo Navarro, Rajeev Raman, and S. Srinivasa Rao. Encoding range minima and range top-2 queries. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 372(2016):20130131, 2014.

15 Gianni Decaroli, Travis Gagie, and Giovanni Manzini. A compact index for order-preserving pattern matching. In *Proceedings of the Data Compression Conference (DCC)*, 2017.

16 Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

17 Robert M. Fano. On the number of bits required to implement an associative memory. Technical Report Memorandum 61, Project MAC, Computer Structures Group, Massachusetts Institute of Technology, 1971.

18 Simone Faro and M. Oğuzhan Külekci. Efficient algorithms for the order preserving pattern matching problem. In *Proceedings of the Conference on Algorithmic Applications in Management (AAIM)*, pages 185–196. Springer, 2016.

19 Paolo Ferragina and Giovanni Manzini. An experimental study of a compressed index. *Information Sciences*, 135(1):13–28, 2001.

20 Johannes Fischer. Combined data structure for previous-and next-smaller-values. *Theoretical Computer Science*, 412(22):2451–2456, 2011.

21 Johannes Fischer and Volker Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 459–470. Springer, 2007.

22 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pBWT: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, pages 397–407. SIAM, 2017.

23 Paweł Gawrychowski and Patrick K. Nicholson. Encodings of range maximum-sum segment queries and applications. In *Proceedings of the Symposium on Combinatorial Pattern Matching (CPM)*, pages 196–206. Springer, 2015.

24 Paweł Gawrychowski and Patrick K. Nicholson. Optimal encodings for range top-k, selection, and min-max. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 593–604. Springer, 2015.

25 Paweł Gawrychowski and Przemysław Uznański. Order-preserving pattern matching with k mismatches. *Theoretical Computer Science*, 638:136–144, 2016.

26 Mordecai Golin, John Iacono, Danny Krizanc, Rajeev Raman, Srinivasa Rao Satti, and Sunil Shende. Encoding 2D range maximum queries. *Theoretical Computer Science*, 609:316–327, 2016.

27 Roberto Grossi, John Iacono, Gonzalo Navarro, Rajeev Raman, and Satti Srinivasa Rao. Encodings for range selection and top-k queries. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 553–564. Springer, 2013.

28 Tommi Hirvola and Jorma Tarhio. Approximate online matching of circular strings. In *Proceedings of the Symposium on Experimental Algorithms (SEA)*, pages 315–325. Springer, 2014.

29 Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE, 1989.

30 Varunkumar Jayapaul, Seungbum Jo, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Space efficient data structures for nearest larger neighbor. *Journal of Discrete Algorithms*, 36:63–75, 2016.

31 Seungbum Jo, Rajeev Raman, and Srinivasa Rao Satti. Compact encodings and indexes for the nearest larger neighbor problem. In *Proceedings of the International Workshop on Algorithms and Computation (WALCOM)*, pages 53–64. Springer, 2015.

32 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014.

33 Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.

34 Gonzalo Navarro, Rajeev Raman, and Srinivasa Rao Satti. Asymptotically optimal encodings for range selection. In *Proceedings of the 34th Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 291–301. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

35 Gonzalo Navarro and Sharma V Thankachan. Encodings for range majority queries. In *Proceedings of the Symposium on Combinatorial Pattern Matching (CPM)*, volume 8486 of *Lecture Notes in Computer Science*, pages 262–272. Springer, 2014.

36 Alessio Orlandi and Rossano Venturini. Space-efficient substring occurrence estimation. *Algorithmica*, 74(1):65–90, 2016.

37 Mihai Patrascu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 166–175. IEEE, 2014.

38 Rajeev Raman. Encoding data structures. In *Proceedings of the International Workshop on Algorithms and Computation (WALCOM)*, pages 1–7. Springer, 2015.

39 Rahul Shah. Personal communication, 2017.

Distance-Preserving Subgraphs of Interval Graphs

Kshitij Gajjar[*1] **and Jaikumar Radhakrishnan**[2]

1 **Tata Institute of Fundamental Research, Mumbai, India**
kshitij.gajjar@tifr.res.in
2 **Tata Institute of Fundamental Research, Mumbai, India**
jaikumar@tifr.res.in

Abstract

We consider the problem of finding small distance-preserving subgraphs of undirected, unweighted interval graphs that have k terminal vertices. We show that every interval graph admits a distance-preserving subgraph with $O(k \log k)$ branching vertices. We also prove a matching $\Omega(k \log k)$ lower bound by exhibiting an interval graph based on bit-reversal permutation matrices. In addition, we show that interval graphs admit subgraphs with $O(k)$ branching vertices that approximate distances up to an additive term of $+1$.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases interval graphs, shortest path, distance-preserving subgraphs, bit-reversal permutation matrix

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.39

1 Introduction

We consider the following problem. Given an undirected graph $G = (V, E)$ with k vertices designated as terminals, our goal is to construct a *small* subgraph H of G. Our notion of smallness is non-standard: we compare solutions based on the number of vertices of degree three or more. We have the following definition.

▶ **Definition 1.** Given an undirected, unweighted graph $G = (V, E)$ and a set $R \subseteq V$ (the terminals), we say that a subgraph $H(V, E')$ of G is distance-preserving for (G, R) if for all terminals $u, v \in R$, $d_G(u, v) = d_H(u, v)$, where d_G and d_H denote the distances in G and H respectively. Let $\deg_{\geq 3}(H)$ denote the number of vertices in H with degree at least three (referred to as *branching vertices*). Let

$$\mathsf{B}(G, R) = \min_H \deg_{\geq 3}(H),$$

where H ranges over all subgraphs that are distance-preserving for (G, R). For a family of graphs $\mathcal{F}$ (such as planar graphs, trees, interval graphs), let

$$\mathsf{B}_{\mathcal{F}}(k) = \max_G \mathsf{B}(G, R),$$

where G ranges over all graphs in $\mathcal{F}$, and R ranges over all subsets of $V(G)$ of size k.

In this work, we obtain essentially tight upper and lower bounds on $\mathsf{B}_{\mathcal{I}}(k)$, where $\mathcal{I}$ is the class of interval graphs.

* Supported by a DAE scholarship.

▶ **Theorem 2** (Main result). *Let $\mathcal{I}$ denote the class of interval graphs (see Theorem 6).*

(a) *(Upper bound)* $\mathsf{B}_{\mathcal{I}}(k) = O(k \log k)$.

(b) *(Lower bound) There exists a constant c such that for each k, a positive power of two, there exists an interval graph G_{int} with $|R| = k$ terminals such that $\mathsf{B}(G_{\text{int}}, R) \geq c\, k \log k$. This implies that $\mathsf{B}_{\mathcal{I}}(k) = \Omega(k \log k)$.*

Parts (a) and (b) imply that $\mathsf{B}_{\mathcal{I}}(k) = \Theta(k \log k)$.

Remark (i). Part (a) is constructive. Our proof of the upper bound can be turned into an efficient algorithm that, given an interval graph G with n vertices, produces the required distance-preserving subgraph H in running time polynomial in n.
Remark (ii). Our interval graphs are unweighted. If we consider the family of interval graphs with non-negative weights on their edges ($\mathcal{I}_w$), then using [9, Section 5], it is easy to prove that $\mathsf{B}_{\mathcal{I}_w}(k) = \Theta(k^4)$. Details appear in the full version of the paper.

1.1 Motivation and Related Work

The problem of constructing small distance-preserving subgraphs bears close resemblance to several well-studied problems in graph algorithms: graph compression [5], graph spanners [3, 11], Steiner point removal [7, 8], graph contractions [4], etc.

We emphasize two motivations for studying distance-preserving subgraphs, while basing the measure of efficiency on the number of branching vertices. First, this problem is closely related to the notion of distance-preserving minors introduced by Krauthgamer and Zondiner [10]. Second, although the problem restricted to interval graphs is interesting in its own right, it can be seen to arise naturally in contexts where intervals represent time periods for tasks. Let us now elaborate on our first motivation. Later, we elaborate on the second.

▶ **Definition 3.** Let $G(V, E, w)$ be an undirected graph with weight function $w : E \to \mathbb{R}^{\geq 0}$ and a set of terminals $R \subseteq V$. Then, $H(V', E', w')$ with $R \subseteq V' \subseteq V$ and weight function $w' : E' \to \mathbb{R}^{\geq 0}$ is a distance-preserving minor of G if: (i) H is a minor of G, and (ii) $d_H(u, v) = d_G(u, v)\, \forall u, v \in R$.

Subsequent work by Krauthgamer, Nguyên and Zondiner [9, 10] implies that $\mathsf{B}_{\mathcal{G}}(k) = \Theta(k^4)$, where $\mathcal{G}$ is the family of all undirected graphs. Details appear in the full version of the paper.

Using a reduction from the set cover problem, we prove that it is NP-hard to determine if $\mathsf{B}(G, R) \leq m$, when given a general graph $G \in \mathcal{G}$, a set of terminals $R \subseteq V(G)$, and a positive integer m. Details appear in the full version of the paper.

Following the work of Krauthgamer and Zondiner [10], Cheung *et al.* [1] introduced the notion of distance-approximating minors.

▶ **Definition 4.** Let $G(V, E, w)$ be an undirected graph with weight function $w : E \to \mathbb{R}^{\geq 0}$ and a set of terminals $R \subseteq V$. Then, $H(V', E', w')$ with $R \subseteq V' \subseteq V$ and weight function $w' : E' \to \mathbb{R}^{\geq 0}$ is an α-distance-approximating minor (α-DAM) of G if: (i) H is a minor of G, and (ii) $d_G(u, v) \leq d_H(u, v) \leq \alpha \cdot d_G(u, v)\, \forall u, v \in V$.

In analogy with distance-approximating minors one may ask if interval graphs admit distance-approximating subgraphs with a small number of branching vertices.

▶ **Theorem 5.** *Every interval graph G with k terminals admits a subgraph H with $O(k)$ branching vertices such that for all terminals u and v of G*

$$d_G(u,v) \leq d_H(u,v) \leq d_G(u,v) + 1.$$

A proof of Theorem 5 will appear in the full version of the paper.

We now elaborate on our second motivation. The following example[1] illustrates the relevance of distance-preserving (-approximating) subgraphs for interval graphs.

1.2 The Shipping Problem

The port of Bandarport is a busy sea port. Apart from ships with routes originating or terminating at Bandarport, there are many ships that dock at Bandarport en route to their final destination. Thus, Bandarport can be considered a hub for many ships from all over the world.

Consider the following shipping problem. A cargo ship starts from some port X, and has Bandarport somewhere on its route plan. The ship needs to deliver a freight container to another port Y, which is not on its route plan. The container can be dropped off at Bandarport and transferred through a series of ships arriving there until it is finally picked up by a ship that is destined for port Y. Thus, the container is transferred from X to Y via some "intermediate" ships at Bandarport[2].

However, there is a cost associated with transferring a container from one ship to another. This is because each transfer operation requires considerable manpower and resources. Thus, the number of ship-to-ship transfers that a container undergoes should be as small as possible.

Furthermore, there is an added cost if an intermediate ship receives containers from multiple ships, or sends containers to multiple ships. This is mainly because of the bookkeeping overhead involved in maintaining which container goes to which ship. If a ship is receiving all its containers *from* just one ship and sending all those containers *to* just one other ship, then the cost associated with this transfer is zero (since a container cannot be directed to a wrong ship if there is only one option), and this cost increases as the number of *to* and *from* ships increases.

Thus, given the docking times of ships at Bandarport, and a small subset of these ships that require a transfer of containers between each other, our goal is to devise a transfer strategy that meets the following objectives.

- Minimize the number of transfers for each container.
- Minimize the number of ships that have to deal with multiple transfers.

Representing each ship's visit to the port as an interval on the time line, this problem can be modelled using distance-preserving (-approximating) subgraphs of interval graphs. In this setting, a shortest path from an earlier interval to a later interval corresponds to a valid sequence of transfers across ships that moves forward in time. The first objective corresponds to minimizing pairwise distances between terminals; the second objective corresponds to minimizing the number of branching vertices.

[1] This is not a real-life problem, though we learnt that minimizing the number of branching vertices in shipping schedules is logistically desirable.

[2] The container cannot be left at the warehouse/storage unit of Bandarport itself beyond a certain limited period of time.

Let us now quantify this. Suppose that there are a total of n ships that dock at the port of Bandarport. Out of these, there are k ships that require a transfer of containers between each other (typically $k \ll n$). Our results for interval graphs imply the following.

1. If we must make no more than the minimum number of transfers required for each container, then there is a transfer strategy in which the number of ships that have to deal with multiple transfers is $O(k \log k)$.
2. If we are allowed to make *one* more than the minimum number of transfers required for each container, then there is a transfer strategy in which the number of ships that have to deal with multiple transfers is $O(k)$.
3. Neither bound can be improved, i.e. there exist scheduling configurations in which $\Omega(k)$ and $\Omega(k \log k)$ ships, respectively, have to deal with multiple transfers.

1.3 Our Techniques

The linear upper bound for $\mathsf{B}_{\mathcal{I}}(k)$ mentioned in Theorem 5 is easy to prove, and appears in the full version of the paper. However, if we require that distances be preserved exactly, then the problem becomes non-trivial. We now present a broad overview of the techniques involved in proving our main result.

The Upper Bound: We may restrict attention to interval graphs that have interval representations where the terminals are intervals of length 0 (their left and right end points are the same) and the non-terminals are intervals of length 1 (details appear in the full version of the paper). It is well-known that shortest paths in interval graphs can be constructed using a simple greedy algorithm. We build a subgraph consisting of such shortest paths starting at different terminals and add edges to it so that all inter-terminal shortest paths become available in the subgraph. We use a divide-and-conquer strategy, repeatedly "cutting" the graph down the middle into smaller interval graphs. Then we glue the solutions to the two smaller problems together. For this, we need a key observation (which appears to be applicable specifically to interval graphs) that allows one shortest path to "hop" onto another. In this, our upper bound method is significantly different from methods used previously for other families of graphs.

The Lower Bound: We construct an interval graph and arrange its vertices on a two-dimensional grid instead of the more natural one-dimensional number line. We then show that this grid can be thought of as a matrix, in particular, the bit-reversal permutation matrix (where the ones corresponding to terminals and the zeros to non-terminals). The bit-reversal permutation matrix has seen many applications, most notably in the celebrated Cooley-Tukey algorithm for Fast Fourier Transform [2]. Prior to our work too, it has been used to devise lower bounds (e.g. [6, 12]). Examining the routes available for shortest paths in our interval graph constructing using the bit-reversal permutation matrix requires (i) an analysis of common prefixes of binary sequences, and (ii) building a correspondence between branching vertices and the $k \log k/2$ edges of a $(\log k)$-dimensional Boolean hypercube.

In our formulation, we count the number of branching vertices (vertices with degree ≥ 3). It is also reasonable to consider the number of edges incident on non-terminal branching vertices (we refer to such edges as *branching edges*) as the measure of complexity. Our $\Omega(k \log k)$ lower bound is clearly applicable to the number of branching edges as well. In fact, using a more direct argument, one can show that there are interval graphs with k terminals that admit distance-preserving subgraphs with $O(k)$ branching vertices, but need $\Omega(k \log k)$ branching edges. Details appear in the full version of the paper. However, we do not know if all interval graphs admit distance-preserving subgraphs with $O(k \log k)$ branching edges: the best upper bound we know for this variant is $O(k \log^2 k)$.

2 Interval Graphs

We work with the following definition of interval graphs.

▶ **Definition 6.** An interval graph is an undirected graph $G(V, E, \mathsf{left}, \mathsf{right})$ with vertex set V, edge set E, and real-valued functions $\mathsf{left} : V \to \mathbb{R}$ and $\mathsf{right} : V \to \mathbb{R}$ such that:

- $\mathsf{left}(x) \leq \mathsf{right}(x) \quad \forall x \in V$;
- $(u, v) \in E \Leftrightarrow [\mathsf{left}(u), \mathsf{right}(u)] \cap [\mathsf{left}(v), \mathsf{right}(v)] \neq \emptyset$.

We order the vertices of the interval graph according to the end points of their corresponding intervals. For simplicity, we assume that all the end points of the intervals have distinct values. Define relations "$\preceq$" and "$\prec$" on the set of vertices V as follows.

$$u \preceq v \Leftrightarrow \mathsf{right}(u) \leq \mathsf{right}(v) \qquad \forall u, v \in V.$$
$$u \prec v \Leftrightarrow \mathsf{right}(u) < \mathsf{right}(v) \qquad \forall u, v \in V.$$

Note that if $u \prec v$, then $u \neq v$.

It is well-known that shortest paths in interval graphs can be constructed using a greedy algorithm which proceeds as follows. Suppose we need to construct a shortest path from interval u to interval v (assume $u \prec v$). The greedy algorithm starts at u. In each step it chooses the next interval that intersects the current interval and reaches farthest to the right. It stops as soon as the current interval intersects v. Let $P_G^{\mathrm{gr}}(u, v)$ be the shortest path produced by this greedy algorithm between u and v ($u \prec v$).

Given real numbers $a, b \in \mathbb{R}$ such that $a \leq b$, let $G[a, b]$ be the induced subgraph on those vertices v of G such that $[\mathsf{left}(v), \mathsf{right}(v)] \cap [a, b] \neq \emptyset$. Similarly, let $G[a, b)$ be the induced subgraph on those vertices v of G such that $[\mathsf{left}(v), \mathsf{right}(v)] \cap [a, b) \neq \emptyset$.

3 Proof of the Upper Bound

In this section, we show that any interval graph G with k terminals has a distance-preserving subgraph with $O(k \log k)$ branching vertices, which is simply Theorem 2 (a), restated here for completeness.

▶ **Theorem 7.** *If $\mathcal{I}$ is the family of all interval graphs, then $\mathsf{B}_{\mathcal{I}}(k) = O(k \log k)$.*

Fix an interval graph G on k terminals. Our goal to obtain a distance-preserving subgraph H of G with $O(k \log k)$ branching vertices. Note that the H that we obtain is not necessarily an interval graph. This is because H need not be an induced subgraph of G. We may assume (details in the full version of the paper) that all terminals in G are point intervals and all non-terminals are unit intervals.

Consider the greedy path $P_G^{\mathrm{gr}}(t_i, t_k)$ ($i < k$), where t_k is the rightmost terminal. Our distance-preserving subgraph includes greedy paths from t_i to t_k for all $1 \leq i < k$. Let

$$H_0 = \bigcup_{1 \leq i < k} P_G^{\mathrm{gr}}(i, k). \tag{1}$$

Now, H_0 already provides for shortest paths from each terminal t_i to t_k. In fact, it can be viewed as a shortest path tree with root t_k, but constructed backwards. Thus, the total number of branching vertices in H_0 is $O(k)$. We still need to arrange for shortest paths between other pairs of terminals (t_i, t_j). The path $P_G^{\mathrm{gr}}(t_i, t_j)$ (for $i < j < k$) is either entirely contained in $P_G^{\mathrm{gr}}(t_i, t_k)$, or it follows $P_G^{\mathrm{gr}}(t_i, t_k)$ until it reaches a neighbour of t_j and then *branches off* to connect to t_j. We can consider including all paths of the form $P_G^{\mathrm{gr}}(t_i, t_j)$ in

H_0. That is, we need to link each such t_j to vertices from H_0 so that each path $P_G^{\text{gr}}(t_i, t_j)$ becomes available. If this is done without additional care, we might end up introducing $\Omega(k)$ additional branching vertices per terminal, and $\Omega(k^2)$ branching vertices in all, far more than we claimed.

The crucial idea for overcoming this difficulty is contained in the following lemma.

▶ **Lemma 8.** *Suppose $v \prec w$ and $d(v, w) = 1$. Let $(v, v_1, v_2, \ldots, v_\ell)$ and $(w, w_1, w_2, \ldots, w_{\ell'})$ be greedy shortest paths starting from v and w respectively. Suppose $\mathsf{right}(v_\ell) < \mathsf{right}(w_{\ell'})$. Then, $\ell \leq \ell'$.*

Proof. Since $d(v, w) = 1$, the greedy strategy reaches at least as far in $j + 1$ steps from v as it does in j steps from w. Suppose for contradiction that $\ell > \ell'$ (that is $\ell \geq \ell' + 1$). Then, we have $\mathsf{right}(w_{\ell'}) \leq \mathsf{right}(v_{\ell'+1}) \leq \mathsf{right}(v_\ell)$, contradicting our assumption that $\mathsf{right}(v_\ell) < \mathsf{right}(w_{\ell'})$. ◀

The above lemma is crucial for the construction of our subgraph H. For example, suppose t_i and t_j both need to reach t_r via a shortest path. Suppose (w_i, t_r) is the last edge of $P_G^{\text{gr}}(t_i, t_r)$ and (w_j, t_r) is the last edge of $P_G^{\text{gr}}(t_j, t_r)$. We claim that it is sufficient to include **only** one of these edges in H. If $\mathsf{right}(w_j) < \mathsf{right}(w_i)$, then it is enough to include the edge (w_j, t_r) in H; as long as t_i has a shortest path to w_j, this edge serves for shortest paths *to* t_r *from* both t_i and t_j. In the construction below, we add links to the greedy paths of H_0 so that we need to provide only one such edge per terminal. This idea forms the basis of the divide-and-conquer strategy which we present below.

Suppose G has 2ℓ terminals. We find a point x so that both $G_{\text{left}} = G[-\infty, x]$ and $G_{\text{right}} = G[x, \infty]$ have ℓ terminals. By induction, we find distance-preserving subgraphs H_{left} and H_{right} of G_{left} and G_{right} with at most $f(\ell)$ branching vertices each. The union of H_{left} and H_{right} has just $2f(\ell)$ branching vertices, but it does not yet guarantee shortest paths from terminals in H_{left} to terminals in H_{right}. Using Theorem 8 and the discussion above, we connect each terminal t_j in H_{right} to **only** one of the greedy shortest paths of terminals from H_{left}, and ensure that shortest paths to t_j are preserved from all terminals t_i in H_{left}. This creates $O(\ell)$ additional branching vertices and give us a recurrence of the form

$$f(2\ell) \leq 2f(\ell) + O(\ell),$$

and the desired upper bound of $O(k \log k)$. Unfortunately, there are technical difficulties in implementing the above strategy as stated. It is therefore helpful to augment H_0 by adding all greedy paths $P_G^{\text{gr}}(t_i, t_j)$, where $d(i, j) \leq 4$. As a result, for each terminal t_i, the first three vertices on $P_G^{\text{gr}}(t_i, t_k)$ might become branching vertices. In all, this adds a *one-time cost* of $O(k)$ branching vertices to our subgraph. We now present the argument formally.

For each (a, b), let $f(a, b)$ be the minimum number of non-terminals in a subgraph H of $G[a, b]$ such that $H_0 \cup H$ preserves all inter-terminal distances in $G[a, b]$; let

$$f(\ell) = \max_{(a,b)} f(a, b),$$

where (a, b) ranges over all pairs such that $G[a, b]$ has at most ℓ terminals. The following lemma is the basis of our induction.

▶ **Lemma 9.**

(i) $f(1) = 0$;

(ii) $f(2\ell) \leq 2f(\ell) + O(\ell)$.

Proof. Part (i) is trivial. For part (ii), fix a pair (a, b) such that $G[a, b]$ has at most 2ℓ terminals. If $b - a \leq 1$, H_0 already preserves distances between every two terminals in $G[a, b]$. So, we may take H to be empty. Now assume that $b - a > 1$. Pick $x \in [a, b]$ as large as possible such that (i) $b - x \geq 1$, and (ii) $G[x, b]$ has at least ℓ terminals.

Let $G_{\text{left}} = G[a, x)$ and $G_{\text{right}} = G[x, b]$. Since G_{right} has at least ℓ terminals, G_{left} has at most ℓ terminals. So, we obtain (by induction) a subgraph H_{left} of $G[a, b]$ with at most $f(\ell)$ non-terminals, such that $H_0 \cup H_{\text{left}}$ preserves inter-terminal distances in G_{left}. If $b - x > 1$, then G_{right} has exactly ℓ terminals, and we obtain by induction a subgraph H_{right} with at most $f(\ell)$ non-terminals such that $H_0 \cup H_{\text{right}}$ preserves all inter-terminal distances in $G[x, b]$. If $b - x = 1$, then we may take H_{right} to be empty (for H_0 already preserves inter-terminal distances in $G[x, b]$).

Our final subgraph H shall be of the form $H_{\text{left}} \cup H_{\text{right}} \cup H_A \cup H_B$, where H_A and H_B are defined as follows. First, consider H_A. Let P_{left} be the set of greedy paths from the terminals in H_{left} to the terminal t_k. Let V_A be the set of all non-terminal intervals of P_{left} that intersect with the interval $[x, x + 1]$. It is easy to see that any path in P_{left} contributes at most 4 non-terminals to V_A. So, $|V_A| \leq 4\ell$. Let H_A be the subgraph of $G[a, b]$ induced by V_A and the terminals in $G[x, x + 1]$.

Note that $H_0 \cup H_{\text{left}} \cup H_{\text{right}} \cup H_A$ preserves all inter-terminal distances in $G[a, x + 1]$ as well as all inter-terminal distances in $G[x + 1, b]$. It, in fact, does more. For each terminal t_i in $G[a, x)$, let v_i be the last vertex on the greedy path $P^{\text{gr}}_G(t_i, t_k)$ that is in V_A. Then, the above graph contains the greedy shortest path from every terminal t_j in $G[a, x]$ to v_i.

Now, it only remains to ensure that distances between terminals in $G[a, x)$ and terminals in $G[x + 1, b]$ are preserved. Let us now define H_B. For each terminal t_j in $G[x + 1, b]$, let v be the earliest interval (with respect to $\prec$) of P_{left} that contains t_j. Then, we include the edge (v, t_j) in H_B. Thus, H_B contains at most one non-terminal per vertex in $G[x + 1, b]$, that is, $O(\ell)$ non-terminals in all. This completes the description of H_A and H_B. The final subgraph is $H = H_{\text{left}} \cup H_{\text{right}} \cup H_A \cup H_B$.

▶ **Claim 10.** *Let t_i be a terminal in $G[a, x)$ and t_r be a terminal in $G[x, b]$. Then, $H_0 \cup H$ preserves the distance between terminal t_i and t_r.*

Proof of Claim 10. Let v be the vertex that we attached to t_r in H_B. If v is on $P^{\text{gr}}_G(t_i, t_k)$, then it follows that $P^{\text{gr}}_G(t_i, t_r)$ is in H, and we are done. So we assume that v is not on $P^{\text{gr}}_G(t_i, t_k)$. Then, let $j \neq i$ be such that $v \in P^{\text{gr}}_G(t_j, t_k)$. Then, we have paths

$$P_G(t_i, t_r) = (t_i, w_1, w_2, \ldots, w_p, w_{p+1}, \ldots, w_{\ell'}, t_r);$$
$$P_H(t_i, t_r) = (t_i, w_1, w_2, \ldots, w_p, v_{q+1}, \ldots, v_\ell = v, t_r),$$

where v_{q+1} is the last vertex on $P^{\text{gr}}_G(t_j, t_k)$ in $G[x, x + 1]$, and w_p is the first vertex on $P^{\text{gr}}_G(t_i, t_r)$ such that $(w_p, v_{q+1}) \in E(G)$. From the construction of H_A, $(w_p, v_{q+1}) \in E(H)$. Following v_q, $(v_{q+1}, \ldots, v_\ell = v, t_r)$ are the subsequent vertices on $P^{\text{gr}}_G(t_j, t_r)$. Note that: (i) $v_{q+1} \prec w_{p+1}$ (otherwise v is on $P^{\text{gr}}_G(t_i, t_k)$), (ii) $d(v_{q+1}, w_{p+1}) = 1$ (both intervals contain $\mathsf{right}(w_p)$), and (iii) $\mathsf{right}(v_\ell) < \mathsf{right}(w_{\ell'})$ (since v is the earliest interval of P_{left} that contains t_j). By Theorem 8, $\ell - q - 1 \leq \ell' - p - 1$. Thus, $P_H(t_i, t_r)$ is no longer than $P^{\text{gr}}_G(t_i, t_r)$. ◀
◀

We can now complete the proof of Theorem 7. By Theorem 9, there is a subgraph H' of G such that $H = H_0 \cup H'$ preserves all inter-terminal distances in G, H_0 has $O(k)$ branching vertices and H' has $O(k \log k)$ non-terminals. It follows that H has $O(k \log k)$ branching vertices.

4 Proof of the Lower Bound

In this section, we show that there exists an interval graph G_{int} such that any distance-preserving subgraph of G_{int} has $\Omega(k \log k)$ branching vertices, which is simply Theorem 2 (b), restated here for completeness.

▶ **Theorem 11.** *If $\mathcal{I}$ is the family of all interval graphs, then $\mathsf{B}_{\mathcal{I}}(k) = \Omega(k \log k)$.*

4.1 Preliminaries

We first set up some terminology that we use in this section. Let $k = 2^\gamma$, where γ is a positive integer. We identify the numbers in the set $\{0, 1, \ldots, k-1\}$ with elements of $\{0,1\}^\gamma$ using the γ-bit binary representation. We index the bits of the binary strings from left to right using integers $i = 1, 2, \ldots, \gamma$. Thus, $x[i]$ denotes the i-th bit of x (from the left); we use $x[i,j]$ to denote the string $x[i]\,x[i+1]\ldots x[j]$ of length $j - i + 1$ (here i, j satisfy $1 \le i \le j \le \gamma$).

For a string of bits a, we use $\mathbf{rev}_\gamma(a)$ to represent the reverse of a, that is, the binary string obtained by writing the bits of a in the reverse order (e.g., $\mathbf{rev}_\gamma(00010) = 01000$). We may arrange binary strings in a binary tree. Refer to Figure 1 for an example. The root is the empty string; the left child of a vertex x is the vertex $x\,0$, and its right child is the vertex $x\,1$. In particular, the string y is a *descendant* of the string x if y is obtained by concatenating x with some (possibly empty) string z, that is, $y = x\,z$. Consider the binary tree of depth γ, whose leaves correspond to elements of $\{0,1\}^\gamma$. For distinct elements $x, y \in \{0,1\}^\gamma$, let $\mathbf{lca}(x,y)$ be the *lowest common ancestor* of x and y defined as follows:

$$\mathbf{lca}(x,y) = x[1, \ell-1] = y[1, \ell-1], \text{ where } \ell = \min\{i \in [\gamma] : x[i] \neq y[i]\}.$$

For example, $\mathbf{lca}(0100111, 0101010) = 010$. Let $\lfloor \mathbf{lca}(x,y) \rfloor$ be the *floor of* $\mathbf{lca}(x,y)$, and $\lceil \mathbf{lca}(x,y) \rceil$ be the *ceiling of* $\mathbf{lca}(x,y)$ defined as follows:

$$\lfloor \mathbf{lca}(x,y) \rfloor = \mathbf{lca}(x,y)\,0\,1^{\gamma-\ell}$$
$$\lceil \mathbf{lca}(x,y) \rceil = \mathbf{lca}(x,y)\,1\,0^{\gamma-\ell}$$

Since $\lfloor \mathbf{lca}(x,y) \rfloor, \lceil \mathbf{lca}(x,y) \rceil \in \{0,1\}^\gamma$, we may regard $\lfloor \mathbf{lca}(x,y) \rfloor$ and $\lceil \mathbf{lca}(x,y) \rceil$ as numbers in the set $\{0, 1, \ldots, k-1\}$. Note that $\lfloor \mathbf{lca}(x,y) \rfloor = \lceil \mathbf{lca}(x,y) \rceil - 1$, and if $x < y$, then $\lfloor \mathbf{lca}(x,y) \rfloor \in [x,y)$ and $\lceil \mathbf{lca}(x,y) \rceil \in (x,y]$[3].

Strings in $\{0,1\}^\gamma$ can also be viewed as vertices of an γ-dimensional hypercube, with edge set

$$\mathcal{H}_\gamma = \{(x,x') : x, x' \in \{0,1\}^\gamma \text{ and } x < x' \text{ and } \mathtt{Ham}(x,x') = 1\},$$

where $\mathtt{Ham}(x,x')$ is the Hamming distance between x and x'. Thus, if $(x,x') \in \mathcal{H}_\gamma$, then x and x' differ at a unique location where x has a zero and x' a one.

▶ **Claim 12.** *Suppose (x,x') and (y,y') are distinct edges of $\mathcal{H}_\gamma$.*
(a) *If $\mathbf{lca}(x,x') = \mathbf{lca}(y,y')$, then $[\mathbf{rev}_\gamma(x), \mathbf{rev}_\gamma(x')] \cap [\mathbf{rev}_\gamma(y), \mathbf{rev}_\gamma(y')] = \emptyset$.*
(b) *If $\{\lfloor \mathbf{lca}(x,x') \rfloor, \lfloor \mathbf{lca}(y,y') \rfloor\} \subseteq [x,x') \cap [y,y')$, then*

$$[\mathbf{rev}_\gamma(x), \mathbf{rev}_\gamma(x')] \cap [\mathbf{rev}_\gamma(y), \mathbf{rev}_\gamma(y')] = \emptyset.$$

[3] $[x,y] \triangleq \{x, x+1, x+2, \ldots, y\}$ and $[x,y) \triangleq \{x, x+1, x+2, \ldots, y-1\}$.

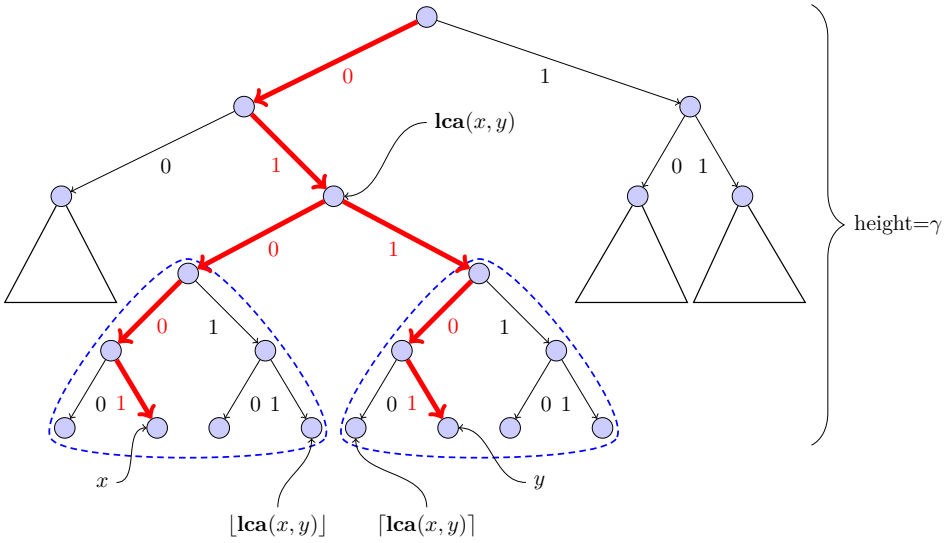


Figure 1 A complete binary tree of height γ having $k = 2^\gamma$ leaves. In this example, $\gamma = 5$, $x = 01001$ and $y = 01101$. Thus, $\texttt{Ham}(x, y) = 1$ and $|\mathbf{lca}(x, y)| = 2$.

Proof. Although part (b) implies part (a), it is easier to show part (a) first, and then derive part (b) from it. For part (a), let $|\mathbf{lca}(x, x')| = |\mathbf{lca}(y, y')| = \ell - 1$. Let $a, b \in \{0, 1\}^{\gamma-\ell}$ be such that

$$a = x[\ell + 1, \gamma] = x'[\ell + 1, \gamma] \neq y[\ell + 1, \gamma] = y'[\ell + 1, \gamma] = b.$$

In particular, we have $a \neq b$ (implying $\mathbf{rev}_{\gamma-\ell}(a) \neq \mathbf{rev}_{\gamma-\ell}(b)$). Note that $\mathbf{rev}_\gamma(a)$ represents the $\gamma - \ell$ most significant bits of $\mathbf{rev}_\gamma(x)$ and $\mathbf{rev}_\gamma(x')$; similarly, $\mathbf{rev}_\gamma(b)$ represents the $\gamma - \ell$ most significant bits of $\mathbf{rev}_\gamma(y)$ and $\mathbf{rev}_\gamma(y')$.

If $\mathbf{rev}_{\gamma-\ell}(a) < \mathbf{rev}_{\gamma-\ell}(b)$ then $\mathbf{rev}_\gamma(x') < \mathbf{rev}_\gamma(y)$; and if $\mathbf{rev}_{\gamma-\ell}(b) < \mathbf{rev}_{\gamma-\ell}(a)$ then $\mathbf{rev}_\gamma(y') < \mathbf{rev}_\gamma(x)$. In either case, $[\mathbf{rev}_\gamma(x), \mathbf{rev}_\gamma(x')]$ and $[\mathbf{rev}_\gamma(y), \mathbf{rev}_\gamma(y')]$ are disjoint, proving part (a).

Next, consider part (b). Suppose $\lfloor \mathbf{lca}(x, x) \rfloor, \lfloor \mathbf{lca}(y, y') \rfloor \in [x, x') \cap [y, y')$. Since every $p \in [x, x')$ is a descendant of $\mathbf{lca}(x, x')$, we conclude that $\mathbf{lca}(y, y')$ is a descendant of $\mathbf{lca}(x, x')$. Similarly, $\mathbf{lca}(x, x')$ is a descendant of $\mathbf{lca}(y, y')$. But then $\mathbf{lca}(x, x') = \mathbf{lca}(y, y')$, and part (b) follows from part (a). ◀

4.2 Manhattan Graphs

In this section, we describe a directed grid graph G_k^{bit} (which we refer to as the Manhattan graph) with $3k$ terminals. We show that any distance-preserving subgraph of G_k^{bit} has $\Omega(k \log k)$ branching vertices. The graph has $k^2 + 2k$ vertices arranged in a square grid. The vertices and edges of G_k^{bit} are defined as follows. (Figure 2 makes this definition easier to understand.)

1. $V(G_k^{\text{bit}}) = \{0, 1, 2, \ldots, k-1\} \times \{-1, 0, 1, \ldots, k\}$.
2. There are three kinds of edges: horizontal, upward and downward; the edge set is given by $E(G_k^{\text{bit}}) = E_{\text{hor}} \cup E_{\text{up}} \cup E_{\text{down}}$, where

$$E_{\text{hor}} = \{((i, j), (i, j + 1)) : i = 0, 1, \ldots, k - 1 \text{ and } j = -1, 0, \ldots, k - 1\};$$
$$E_{\text{up}} = \{((i_1, j), (i_2, j)) : 0 \leq i_2 < i_1 \leq k - 1 \text{ and } j = -1, 0, \ldots, k\};$$
$$E_{\text{down}} = \{((i_1, j), (i_2, j)) : 0 \leq i_1 < i_2 \leq k - 1 \text{ and } j = -1, 0, \ldots, k\}.$$

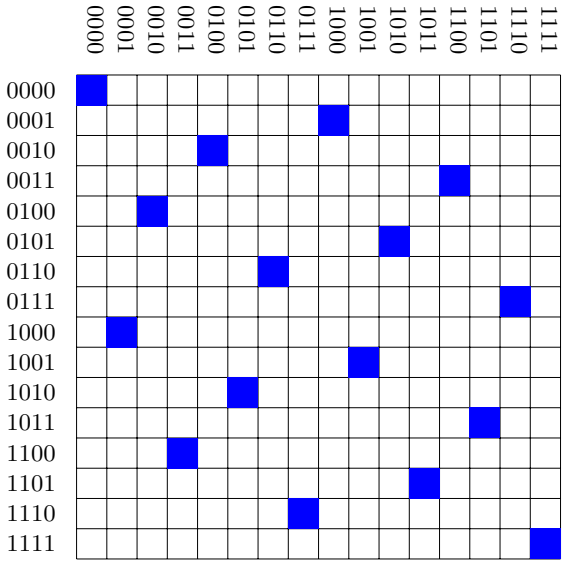


Figure 2 The bit-reversal permutation matrix for $k = 16$. Each cell represents a vertex: the blue cells represent the terminal vertices of T_{mid}; all the other vertices are non-terminals. Edges are named horizontal, upward and downward in the natural way.

3. The edge weights are given by the function $w : E(G_k^{\text{bit}}) \to \{0, 1\}$, defined as follows: $w(e) = 1$ if $e \in E_{\text{hor}} \cup E_{\text{up}}$, and $w(e) = 0$ if $e \in E_{\text{down}}$.

The set of terminals are of the form $T = T_{\text{left}} \cup T_{\text{mid}} \cup T_{\text{right}}$, where

$$T_{\text{left}} = \{0, 1, \ldots, k-1\} \times \{-1\},$$
$$T_{\text{right}} = \{0, 1, \ldots, k-1\} \times \{k\};$$
$$T_{\text{mid}} = \{(\mathbf{rev}_\gamma(i), i) : i = 0, 1, \ldots, k-1\}.$$

This completes the definition of G_k^{bit}.

Fix an optimal distance-preserving subgraph H_k^{bit} of G_k^{bit}. We shall show that H_k^{bit} has $\Omega(k \log k)$ vertices of degree at least 3.

▶ **Lemma 13.** *$V(H_k^{\text{bit}}) = V(G_k^{\text{bit}})$ and $E_{\text{hor}} \subseteq E(H_k^{\text{bit}})$.*

Proof. Note that the *unique* shortest path between the terminals $(i, -1)$ and (i, k) is precisely $((i, -1), (i, 0), \ldots, (i, k))$. Thus, all vertices and all horizontal edges in the i-th row of G_k^{bit} must be part of H_k^{bit}. ◀

It follows from Theorem 13 that every non-terminal vertex in H_k^{bit} has degree at least two, namely the two horizontal edges incident on it.

Special edges: From now on, we rely solely on the fact that H_k^{bit} is distance-preserving for every pair of terminals in T_{mid}, i.e. we prove the stronger statement that just preserving terminal distances in T_{mid} requires $\Omega(k \log k)$ branching vertices.

Order the vertices in T_{mid} as $t_0, t_1, \ldots, t_{k-1}$, where $t_i = (\mathbf{rev}_\gamma(i), i)$. Note that these terminals appear in different rows and columns. Consider the following pairs of terminals.

$$T_{\text{twins}} = \{(t_i, t_j) : (i, j) \in \mathcal{H}_\gamma\}.$$

For each twin (t_i, t_j), fix $P(i, j)$, a path of minimum distance between t_i and t_j in H_k^{bit}. We are now set to formally define special edges.

▶ **Definition 14.** Let $\mathbf{spcl}(i,j) = ((r_{ij}, \lfloor \mathbf{lca}(i,j) \rfloor), (r_{ij}, \lceil \mathbf{lca}(i,j) \rceil))$ be an edge of $P(i,j)$, where $\mathbf{rev}_\gamma(i) \leq r_{ij} \leq \mathbf{rev}_\gamma(j)$. (By Theorem 15, such an edge exists.) Let $\mathbf{spcl} = \{\mathbf{spcl}(i,j) : (t_i, t_j) \in T_{\text{twins}}\}$.

▶ **Lemma 15.** *Let $(t_i, t_j) \in T_{\text{twins}}$; let $\ell = \lfloor \mathbf{lca}(i,j) \rfloor$. Then, there is an $r_{ij} \in [\mathbf{rev}_\gamma(i), \mathbf{rev}_\gamma(j)]$ such that $P(i,j)$ contains the edge $((r_{ij}, \ell), (r_{ij}, \ell+1))$.*

Proof. We have $i < j$, $t_i = (\mathbf{rev}_\gamma(i), i)$ and $t_j = (\mathbf{rev}_\gamma(j), j)$. Also note that since $(i,j) \in \mathcal{H}_\gamma$, $\mathbf{rev}_\gamma(i) < \mathbf{rev}_\gamma(j)$. Thus, $d(i,j) = j - i$, and the shortest path $P(t_i, t_j)$ goes from column i to column j and never skips a column. Since $\ell \in [i,j)$, there must be an edge in $P(i,j)$ of the form $((r_{ij}, \ell), (r_{ij}, \ell+1))$ (say, the edge of $P(i,j)$ that leaves column ℓ for the last time). We claim that $r_{ij} \in [\mathbf{rev}_\gamma(i), \mathbf{rev}_\gamma(j)]$. For otherwise, $P(i,j)$ would contain an edge in E_{up}. Then, apart from the $j - i$ edges from E_{hor}, $P(i,j)$ would contain an additional edge from E_{up} of weight 1; that is, the length of $P(i,j)$ would be at least $j - i + 1$—contradicting the fact that $d(i,j) = j - i$. ◀

▶ **Lemma 16** (Key lemma). *Suppose $(t_x, t_{x'})$ and $(t_y, t_{y'})$ are distinct pairs in T_{twins} such that their special edges are in the same row r, that is,*

$$\mathbf{spcl}(x,x') = ((r,\alpha),(r,\alpha+1))$$
$$\mathbf{spcl}(y,y') = ((r,\beta),(r,\beta+1)),$$

where $\alpha = \lfloor \mathbf{lca}(x,x') \rfloor$ and $\beta = \lfloor \mathbf{lca}(y,y') \rfloor$.

(a) *Then, $\alpha \neq \beta$. In particular, $\mathbf{spcl}(x,x') \neq \mathbf{spcl}(y,y')$.*

(b) *Suppose $\alpha < \beta$. Then, there exists an $\ell \in [\alpha+1, \beta]$ such that (r,ℓ) is either a branching vertex or a terminal in H_k^{bit}.*

Proof. Part (a) follows from Claim 12 (a). Consider part (b). By our definition of special edge, $r \in [\mathbf{rev}_\gamma(x), \mathbf{rev}_\gamma(x')]$ and $r \in [\mathbf{rev}_\gamma(y), \mathbf{rev}_\gamma(y')]$. So, $[\mathbf{rev}_\gamma(x), \mathbf{rev}_\gamma(x')] \cap [\mathbf{rev}_\gamma(y), \mathbf{rev}_\gamma(y')] \neq \emptyset$, and by Claim 12 (b) (in the contrapositive) either $\alpha \notin [y, y')$ or $\beta \notin [x, x')$. If $\alpha \notin [y, y')$, $\mathbf{spcl}(x,x')$ is not on $P(y,y')$. The first vertex in row r that is part of $P(y,y')$ is in a column $\ell \in [\alpha+1, \beta]$. Then, (r,ℓ) is either a branching vertex or the terminal t_y. On the other hand, if $\beta \notin [x, x')$, then the last vertex of $P(t_x, t_{x'})$ in row r lies in a column $\ell \in [\alpha+1, \beta]$, so (r,ℓ) is either a branching vertex or the terminal $t_{x'}$. ◀

▶ **Corollary 17.**

(a) $|\mathbf{spcl}| = |T_{\text{twins}}| = k \log k/2$ *(since $|T_{\text{twins}}| = |\mathcal{H}_\gamma| = k \log k/2$).*

(b) *If two edges in* $\mathbf{spcl}$ *fall in the same row, then there is a branching vertex or a terminal separating them.*

▶ **Theorem 18.** *H_k^{bit} has $\Omega(k \log k)$ branching vertices.*

Proof. For each $i \in \{0, 1, \ldots, k-1\}$, let δ_i be the number of distinct edges in $\mathbf{spcl}$ in row i. Then, by Theorem 17 (a), we have

$$\sum_{i=0}^{k-1} \delta_i = |\mathbf{spcl}| = \left(\frac{k \log k}{2}\right).$$

Furthermore, Theorem 17 (b) implies that there are at least $\delta_i - 2$ many branching vertices of the form (i,x) in H_k^{bit}, where $0 \leq x \leq k-1$. Thus, the total number of branching vertices in H_k^{bit} is at least

$$(\delta_0 - 2) + (\delta_1 - 2) + \cdots + (\delta_{k-1} - 2) = \left(\sum_{i=0}^{k-1} \delta_i\right) - 2k = \left(\frac{k \log k}{2}\right) - 2k.$$

Since this quantity is $\Omega(k \log k)$, this completes the proof. ◀

4.3 Translating the Lower Bound to Interval Graphs

In this section, we present an interval graph G_{int} with $O(k)$ terminals, for which every distance-preserving subgraph has $\Omega(k \log k)$ branching vertices. Our lower bound relies on the lower bound for the Manhattan graph shown in the previous section. Let us describe the interval graph. Let $\mathcal{J}$ be the set of intervals.

$$\mathcal{J} = \{[x, x+1] : x = -1, -1+1/k, \ldots, -1/k, 0, \ldots, k, k+1/k, \ldots, k+1-1/k\}.$$

Thus, we have unit intervals starting at all integral multiples of $1/k$ in the range $[-1, k+1-1/k]$; in all we have $k(k+2)$ intervals in $\mathcal{J}$. These intervals naturally define an interval graph. Furthermore, the edges of G_{int} are *directed* as follows. Orient the edges of G_{int} *from* an earlier interval *to* a later interval, i.e. $([x, x+1], [y, y+1])$ is a directed edge *from* $[x, x+1]$ *to* $[y, y+1]$ if and only if $x < y \leq x+1$. Note that this orientation does not affect shortest paths. Any shortest path from $[i, i+1]$ to $[j, j+1]$ (where $i < j$) in the undirected interval graph is also a valid directed shortest path in G_{int}. Also, G_{int} has $k^2 + 2k$ vertices, which (surprisingly?) is the number of vertices in the Manhattan graph of the previous section. In fact, the connection is deeper. Let us arrange the intervals in a two-dimensional array

$$\mathbf{A} = \langle a_{i,j} : i = 0, \ldots, k-1 \text{ and } j = -1, 0, \ldots, k \rangle,$$

where a_{ij} corresponds to the interval $[j + (k-1-i)/k, j+1+(k-1-i)/k]$. Thus, the first k intervals of $\mathcal{J}$ occupy the left most column of the array $\mathbf{A}$ (from bottom to top); the next k intervals occupy the next column (again from bottom to top), and so on. It is easy to check that, after this arrangement, the directed edges of G_{int} are of three types: horizontal, upward and slanting.

$$\begin{aligned}
E_{\text{hor}}(G_{\text{int}}) &= \{(a_{i,j}, a_{i,j+1}) : 0 \leq i \leq k-1 \text{ and } -1 \leq j \leq k-1\};\\
E_{\text{up}}(G_{\text{int}}) &= \{(a_{i,j}, a_{i',j}) : 1 \leq i \leq k-1 \text{ and } 0 \leq i' < i \text{ and } -1 \leq j \leq k\};\\
E_{\text{slant}}(G_{\text{int}}) &= \{(a_{i,j}, a_{i',j+1}) : 0 \leq i \leq k-2 \text{ and } i < i' \leq k-1 \text{ and } -1 \leq j \leq k-1\}.
\end{aligned}$$

Thus, $E(G_{\text{int}}) = E_{\text{hor}}(G_{\text{int}}) \cup E_{\text{up}}(G_{\text{int}}) \cup E_{\text{slant}}(G_{\text{int}})$. All edges in $E(G_{\text{int}})$ have weight 1. This 2d array can be viewed as a $k \times (k+2)$ grid, and we place terminals in this graph at the same $3k$ locations as in the Manhattan graph. This completes the description of G_{int}. Using the lower bound shown for Manhattan graphs in the previous section (Theorem 18), we complete the proof of Theorem 11. (Details appear in the full version of the paper.)

Acknowledgments. We are grateful to Nithin Varma and Rakesh Venkat for introducing us to the problem and helping with the initial analysis of shortest paths in interval graphs, and for their comments at various stages of this work. We would also like to thank the anonymous reviewers of this paper for their helpful suggestions and comments.

References

1 Yun Kuen Cheung, Gramoz Goranci, and Monika Henzinger. Graph Minors for Preserving Terminal Distances Approximately - Lower and Upper Bounds. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 131:1–131:14, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2016.131.

2 James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

3 Don Coppersmith and Michael Elkin. Sparse sourcewise and pairwise distance preservers. *SIAM Journal on Discrete Mathematics*, 20(2):463–501, 2006.

4 Karl Däubel, Yann Disser, Max Klimm, Torsten Mütze, and Frieder Smolny. Distance-preserving graph contractions. *CoRR*, abs/1705.04544, 2017. URL: `http://arxiv.org/abs/1705.04544`.

5 Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. *J. Comput. System Sci.*, 51(2):261–272, 1995. `doi:10.1006/jcss.1995.1065`.

6 Greg N Frederickson and Nancy A Lynch. Electing a leader in a synchronous ring. *Journal of the ACM (JACM)*, 34(1):98–115, 1987.

7 Anupam Gupta. Steiner points in tree metrics don't (really) help. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 220–227. Society for Industrial and Applied Mathematics, 2001.

8 Lior Kamma, Robert Krauthgamer, and Huy L Nguyên. Cutting corners cheaply, or how to remove steiner points. *SIAM Journal on Computing*, 44(4):975–995, 2015.

9 Robert Krauthgamer, Huy Nguyên, and Tamar Zondiner. Preserving terminal distances using minors. *SIAM Journal on Discrete Mathematics*, 28(1):127–141, 2014. `doi:10.1137/120888843`.

10 Robert Krauthgamer and Tamar Zondiner. Preserving terminal distances using minors. In *Automata, Languages, and Programming*, volume 7391 of *Lecture Notes in Computer Science*, pages 594–605. Springer Berlin Heidelberg, 2012. `doi:10.1007/978-3-642-31594-7_50`.

11 David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989. `doi:10.1002/jgt.3190130114`.

12 Mihai Pătraşcu and Erik D Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.

Dispersion on Trees*†

Paweł Gawrychowski[1], Nadav Krasnopolsky[2], Shay Mozes[3], and Oren Weimann[4]

1 **University of Haifa, Israel**
2 **University of Haifa, Israel**
3 **IDC Herzliya, Israel**
4 **University of Haifa, Israel**

Abstract

In the k-dispersion problem, we need to select k nodes of a given graph so as to maximize the minimum distance between any two chosen nodes. This can be seen as a generalization of the independent set problem, where the goal is to select nodes so that the minimum distance is larger than 1. We design an optimal $O(n)$ time algorithm for the dispersion problem on trees consisting of n nodes, thus improving the previous $O(n \log n)$ time solution from 1997.

We also consider the weighted case, where the goal is to choose a set of nodes of total weight at least W. We present an $O(n \log^2 n)$ algorithm improving the previous $O(n \log^4 n)$ solution. Our solution builds on the search version (where we know the minimum distance λ between the chosen nodes) for which we present tight $\Theta(n \log n)$ upper and lower bounds.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases parametric search, dispersion, k-center, dynamic programming

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.40

1 Introduction

Facility location is a family of problems dealing with the placement of facilities on a network in order to optimize certain distances between the facilities, or between facilities and other nodes of the network. Such problems are usually if not always NP-hard on general graphs. There is a rich literature on approximation algorithms (see e.g. [14, 16] and references therein) as well as exact algorithms for restricted inputs. In particular, many linear and near-linear time algorithms were developed for facility location problems on edge-weighted trees.

In the most basic problem, called *k-center*, we are given an edge-weighted tree with n nodes and wish to designate up to k nodes to be facilities, so as to minimize the maximum distance of a node to its closest facility. This problem was studied in the early 80's by Megiddo et al. [12] who gave an $O(n \log^2 n)$ time algorithm that was subsequently improved to $O(n \log n)$ by Frederickson and Johnson [9]. In the early 90's, an optimal $O(n)$ time solution was given by Frederickson [8, 6] using a seminal approach based on parametric search, also for two other versions where points on edges can be designated as facilities or where we minimize over points on edges. In yet another variant, called weighted k-center, every node has a positive weight and we wish to minimize the maximum weighted distance of a node to its closest facility. Megiddo et al. [12] solved this in $O(n \log^2 n)$ time, and Megiddo and

* The research was supported in part by Israel Science Foundation grant 794/13.
† The full version of this paper, containing missing proofs and supplementary figures, is available at `http://arxiv.org/abs/1706.09185`.

Tamir [11] designed an $O(n \log^2 n \log\log n)$ time algorithm when allowing points on edges to be designated as facilities. The latter complexity can be further improved to $O(n \log^2 n)$ using a technique of Cole [5]. A related problem, also suggested in the early 80's [1, 13], is *k-partitioning*. In this problem the nodes have weight and we wish to delete k edges in the tree so as to maximize the weight of the lightest resulting subtree. This problem was also solved by Frederickson in $O(n)$ time [7] using his parametric search framework.

The focus of this paper is the *k-dispersion* problem, where we wish to designate k nodes as facilities so as to maximize the distances among the facilities. In other words, we wish to select k nodes that are as spread-apart as possible. More formally, let $d(u, v)$ denote the distance between nodes u and v, and for a subset of nodes P let $f(P) = \min_{u,v \in P}\{d(u, v)\}$.

- *The Dispersion Optimization Problem.* Given a tree with non-negative edge lengths, and a number k, find a subset P of nodes of size k such that $f(P)$ is maximized.

The dispersion problem can be seen as a generalization of the classical maximum independent set problem (that can be solved by binary searching for the largest value of k for which the minimum distance is at least 2). It can also be seen as a generalization of the diameter problem (i.e., when $k = 2$).

It turns out that the dispersion and the k-partitioning problems are actually equivalent in the one-dimensional case (i.e., when the tree is a path). The reduction simply creates a new path whose edges correspond to nodes in the original path and whose nodes correspond to edges in the original path. However, such equivalence does not apply to general trees, on which k-dispersion seems more difficult than k-partitioning. In particular, until the present work, no linear time solution for k-dispersion was known. The dispersion optimization problem can be solved by repeatedly querying a *feasibility test* that solves the dispersion search problem.

- *The Dispersion Search Problem (feasibility test).* Given a tree with non-negative edge lengths, a number k, and a number λ, find a subset P of nodes of size k such that $f(P) \geq \lambda$, or declare that no such subset exists.

Bhattacharya and Houle [2] presented a linear-time feasibility test, and used a result by Frederickson [9] that enables binary searching over all possible values of λ (i.e., all pairwise distances in the tree). That is, a feasibility test with a running time τ implies an $O(n \log n + \tau \cdot \log n)$ time algorithm for the dispersion optimization problem. Thus, the algorithm of Bhattacharya and Houle for the dispersion optimization problem runs in $O(n \log n)$ time. We present a linear time algorithm for the optimization problem. Our solution is based on a simplified linear-time feasibility test, which we turn into a sublinear-time feasibility test in a technically involved way closely inspired by Frederickson's approach.

In the *weighted* dispersion problem, nodes have non-negative weights. Instead of k we are given W, and the goal is then to find a subset P of nodes of total weight at least W s.t. $f(P)$ is maximized. Bhattacharya and Houle considered this generalization in [3]. They presented an $O(n \log^3 n)$ feasibility test for this generalization, that by the same reasoning above solves the weighted optimization problem in $O(n \log^4 n)$ time. We give an $O(n \log n)$-time feasibility test, and a matching lower bound. Thus, our algorithm for the weighted optimization problem runs in $O(n \log^2 n)$ time. Our solution uses novel ideas, and differs substantially from Frederickson's approach.

Our technique for the unweighted dispersion problem. Our solution to the k-dispersion problem can be seen as a modern adaptation of Frederickson's approach based on a hierarchy of micro-macro decompositions. While achieving this adaptation is technically involved, we

believe this modern view might be of independent interest. As in Frederickson's approach for k-partitioning and k-center, we develop a feasibility test that requires *linear* time preprocessing and can then be queried in *sublinear* time. Equipped with this sublinear feasibility test, it is still not clear how to solve the whole problem in $O(n)$ time, as in such complexity it is not trivial to represent all the pairwise distances in the tree in a structure that enables binary searching. To cope with this, we maintain only a subset of candidate distances and represent them using matrices where both rows and columns are sorted. Running feasibility tests on only a few candidate entries from such matrices allows us to eliminate many other candidates, and prune the tree accordingly. We then repeat the process with the new smaller tree. This is similar to Frederickson's approach, but our algorithm (highlighted below) differs in how we construct these matrices, in how we partition the input tree, and in how we prune it.

Our algorithm begins by partitioning the input tree T into $O(n/b)$ *fragments*, each with $O(b)$ nodes and at most two *boundary nodes* incident to nodes in other fragments: the root of the fragment and, possibly, another boundary node called the *hole*. We use this to simulate a bottom-up feasibility test by jumping over entire fragments, i.e., knowing λ, we wish to extend in $O(\log b)$ time a solution for a subtree of T rooted at the fragment's hole to a subtree of T rooted at the fragment's root. This is achieved by efficient preprocessing: The first step of the preprocessing computes values λ_1 and λ_2 such that (1) there is no solution to the search problem on T for any $\lambda \geq \lambda_2$, (2) there is a solution to the search problem on T for any $\lambda \leq \lambda_1$, and (3) for *most* of the fragments, the distance between any two nodes is either smaller or equal to λ_1 or larger or equal to λ_2. This is achieved by applying Frederickson's parametric search on sorted matrices capturing the pairwise distances between nodes in the same fragment. The (few) fragments that do not satisfy property (3) are handled naively in $O(b)$ time during query time. The fragments that do satisfy property (3) are further preprocessed. We look at the path from the hole to the root of the fragment and run the linear-time feasibility test for all subtrees hanging off from it. Because of property (3), this can be done in advance without knowing the actual exact value of $\lambda \in (\lambda_1, \lambda_2)$, which will only be determined at query time. Let P be a solution produced by the feasibility test to a subtree rooted at a node u. It turns out that the interaction between P and the solution to the entire tree depends only on two nodes of P, which we call the *certain* node and the *candidate* node. We can therefore conceptually replace each hanging subtree by two leafs, and think of the fragment as a caterpillar connecting the root and the hole. After some additional pruning, we can precompute information that will be used to accelerate queries to the feasibility test. During a query we will be able to jump over each fragment of size $O(b)$ in just $O(\log b)$ time, so the test takes $O(\frac{n}{b} \log b)$ time.

The above sublinear-time feasibility test is presented in Section 3, with an overall preprocessing time of $O(n \log\log n)$. The test is then used to solve the optimization problem within the same time. This is done, again, by maintaining an interval $[\lambda_1, \lambda_2)$ and applying Frederickson's parametric search, but now we apply a heavy path decomposition to construct the sorted matrices. To accelerate the $O(n \log\log n)$ time algorithm, we construct a hierarchy of feasibility tests by partitioning the input tree into larger and larger fragments. In each iteration we construct a feasibility test with better running time, until finally, after $\log^* n$ iterations we obtain a feasibility test with $O(\frac{n}{\log^4 n} \cdot \log\log n)$ query-time, which we use to solve the dispersion optimization problem in linear time. It is relatively straightforward to implement the precomputation done in a single iteration in $O(n)$ time. However, achieving total $O(n)$ time over all the iterations, requires reusing the results of the precomputation across iterations as well as an intricate global analysis of the overall complexity. Thus, the details of the linear-time algorithm are technically involved and appear in the full version.

Our technique for the weighted dispersion problem. Our solution for the weighted case differs substantially from Frederickson's approach. In contrast to the unweighted case, where it suffices to consider a single candidate node, in the weighted case each subtree might have a large number of candidate nodes. To overcome this, we represent the candidates of a subtree with a *monotonically decreasing polyline*: for every possible distance d, we store the maximum weight $W(P)$ of a subset of nodes P such that the distance of every node of P to the root of the subtree is at least d. This can be conveniently represented by a sorted list of breakpoints, and the number of breakpoints is at most the size of the subtree. We then show that the polyline of a node can be efficiently computed by merging the polylines of its children. If the polylines are stored in augmented balanced search trees, then two polylines of size x and y can be merged in time $O(\min(x,y)\log\max(x,y))$, and by standard calculation we obtain an $O(n\log^2 n)$ time feasibility test. To improve on that and obtain an optimal $O(n\log n)$ feasibility test, we need to be able to merge polylines in $O(\min(x,y)\log\frac{\max(x,y)}{\min(x,y)})$ time. An old result of Brown and Tarjan [4] is that, in exactly such time we can merge two *2-3 trees* representing two sorted lists of length x and y (and also delete x nodes in a tree of size y). This was later generalized by Huddleston and Mehlhorn [10] to any sequence of operations that exhibits a certain locality of reference. However, in our specific application we need various non-standard batch operations on the lists. any balanced search tree with split and join capabilities. Our data structure both simplifies and extends that of Brown and Tarjan [4], and might be of independent interest.

2 A Linear Time Feasibility Test

Given a tree T with non-negative lengths and a number λ, the feasibility test finds a subset of nodes P such that $f(P)\geq\lambda$ and $|P|$ is maximized, and then checks if $|P|\geq k$. To this end, the tree is processed bottom-up while computing, for every subtree T_r rooted at a node r, a subset of nodes P such that $f(P)\geq\lambda$, $|P|$ is maximized, and in case of a tie $\min_{u\in P} d(r,u)$ is additionally maximized. We call the node $u\in P$, s.t. $d(r,u)<\frac{\lambda}{2}$, the *candidate* node of the subtree (or a candidate with respect to r). There is at most one such candidate node. The remaining nodes in P are called *certain* (with respect to r) and the one that is nearest to the root is called the certain node. When clear from the context, we will not explicitly say which subtree we are referring to.

In each step we are given a node r, its children nodes $r_1, r_2, \ldots, r_\ell$ and, for each child r_i, a maximal valid solution P_i for the feasibility test on T_{r_i} together with the candidate and the certain node. We obtain a maximal valid solution P for the feasibility test on T_r as follows:

1. Take all nodes in $P_1,\ldots,P_\ell$, except for the candidate nodes.
2. Take all candidate nodes u s.t. $d(u,r)\geq\frac{\lambda}{2}$ (i.e., they are certain w.r.t. r).
3. If it exists, take u', the candidate node farthest from r s.t. $d(u',r)<\frac{\lambda}{2}$ and $d(u',x)\geq\lambda$, where x is the closest node to u' we have taken so far.
4. If the distance from r to the closest vertex in P is at least λ, add r to P.

Iterating over the input tree bottom-up as described results in a valid solution P for the whole tree. Finally, we check if $|P|\geq k$.

▶ **Lemma 1.** *The above feasibility test works in linear time and finds P such that $f(P)\geq\lambda$ and $|P|$ is maximized.*

3 An $O(n \log\log n)$ Time Algorithm for the Dispersion Problem

To accelerate the linear-time feasibility test described in Section 2, we will partition the tree into $O(n/b)$ *fragments*, each of size at most b. We will preprocess each fragment to implement the bottom-up feasibility test in sublinear time by "jumping" over fragments in $O(\log b)$ time instead of $O(b)$. The preprocessing takes $O(n \log b)$ time (Section 3.1), and each feasibility test can then be implemented in sublinear $O(\frac{n}{b} \cdot \log b)$ time (Section 3.2). Using heavy-path decomposition, we design an algorithm for the unweighted dispersion optimization problem whose running time is dominated by $O(\log^2 n)$ calls it makes to the sublinear feasibility test (Section 3.3). By setting $b = \log^2 n$ we obtain an $O(n \log\log n)$ time algorithm.

Each fragment is defined by one or two *boundary* nodes: a node u, and possibly a descendant v of u. The fragment whose boundary nodes are u and v consists of the subtree of u without the subtree of v (v does not belong to the fragment). Thus, each fragment is connected to the rest of the tree only through its boundary nodes. We call the path from u to v the fragment's *spine*, and v's subtree its *hole*. If the fragment has only one boundary node, i.e., the fragment consists of a node and all its descendants, we say that there is no hole. A partition of a tree into $O(n/b)$ such fragments, each of size at most b, is called a *good partition*. Note that we can assume that the input tree is binary: given a non-binary tree, we can replace every degree $d \geq 3$ node with a binary tree on d leaves. The edges of the binary tree are all of length zero, so at most one node in the tree can be taken.

▶ **Lemma 2.** *For any binary tree on n nodes and a parameter b, a good partition of the tree can be found in $O(n)$ time.*

3.1 The preprocessing

Recall that the goal in the optimization problem is to find the largest feasible λ^*. Such λ^* is a distance between an unknown pair of vertices in the tree. The first goal of the preprocessing step is to eliminate many possible pairwise distances, so that we can identify a small interval $[\lambda_1, \lambda_2)$ that contains λ^*. We want this interval to be sufficiently small so that for (almost) every fragment F, handling F during the bottom up feasibility test for any value λ in $[\lambda_1, \lambda_2)$ is the same. Observe that the feasibility test in Section 2 for value λ only compares distances to λ and to $\lambda/2$. We therefore call a fragment F *inactive* if for any two nodes $u_1, u_2 \in F$ the following two conditions hold: (1) $d(u_1, u_2) \leq \lambda_1$ or $d(u_1, u_2) \geq \lambda_2$, and (2) $d(u_1, u_2) \leq \frac{\lambda_1}{2}$ or $d(u_1, u_2) \geq \frac{\lambda_2}{2}$. For an inactive fragment F, all the comparisons performed by the feasibility test for any $\lambda \in [\lambda_1, \lambda_2)$ only depend on the interval $[\lambda_1, \lambda_2)$, but not on the particular value of λ. Therefore, once we find an interval $[\lambda_1, \lambda_2)$ for which (almost) all fragments are inactive, we can precompute, for each inactive fragment F, information that will enable us to process F in $O(\log b)$ time during any subsequent feasibility test with $\lambda \in (\lambda_1, \lambda_2)$.

The first goal of the preprocessing step is therefore to find a small enough interval $[\lambda_1, \lambda_2)$. For each fragment F, we construct an implicit representation of $O(b)$ sorted matrices of total side length $O(b \log b)$, s.t. for every two nodes u_1, u_2 in F, $d(u_1, u_2)$ (and also $2d(u_1, u_2)$) is an entry in some matrix. This is done using the standard centroid decomposition, in $O(\frac{n}{b} \cdot b \log b) = O(n \log b)$ total time using the following lemma.

▶ **Lemma 3.** *Given a tree T on b nodes, we can construct in $O(b \log b)$ time an implicit representation of $O(b)$ sorted matrices of total side length $O(b \log b)$ such that, for any $u, v \in T$, $d(u, v)$ is an entry in some matrix.*

Then, we repeatedly choose an entry of a matrix and run a feasibility test with its value. Depending on the outcome, we then appropriately shrink the current interval $[\lambda_1, \lambda_2)$ and

discard this entry. Because the matrices are sorted, running a single feasibility test can actually allow us to discard multiple entries in the same matrix (and, possibly, also entries in some other matrices). The following theorem by Frederickson shows how to exploit this to discard most of the entries with very few feasibility tests.

▶ **Theorem 4** ([7]). *Let $M_1, M_2, ..., M_N$ be a collection of sorted matrices in which matrix M_j is of dimension $m_j \times n_j$, $m_j \leq n_j$, and $\sum_{j=1}^{N} m_j = m$. Let p be nonnegative. The number of feasibility tests needed to discard all but at most p of the elements is $O(\max\{\log(\max_j\{n_j\}), \log(\frac{m}{p+1})\})$, and the total running time exclusive of the feasibility tests is $O(\sum_{j=1}^{N} m_j \cdot \log(2n_j/m_j))$.*

Setting $m = b \log b \cdot \frac{n}{b} = n \log b$ and $p = n/b^2$, the theorem implies that we can use $O(\log b)$ calls to the linear time feasibility test and discard all but n/b^2 elements of the matrices. Therefore, all but at most n/b^2 fragments are inactive.

The second goal of the preprocessing step is to compute information for each inactive fragment that will allow us to later "jump" over it in $O(\log b)$ time when running the feasibility test. We next describe this computation. We choose λ arbitrarily in (λ_1, λ_2). This is done just so that we have a concrete value of λ to work with.

1. **Reduce the fragment to a caterpillar:** a fragment consists of the spine and the subtrees hanging off the spine. We run our linear-time feasibility test on the subtrees hanging off the spine, and obtain the candidate and the certain node for each of them. The fragment can now be reduced to a caterpillar with at most two leaves attached to each spine node: a candidate node and a certain node.
2. **Find candidate nodes that cannot be taken into the solution:** for each candidate node we find its nearest certain node. Then, we compare their distance to λ and remove the candidate node if it cannot be taken. To find the nearest certain node, we first scan all nodes bottom-up (according to the natural order on the spine nodes they are attached to) and compute for each of them the nearest certain node below it. Then, we repeat the scan in the other direction to compute the nearest certain node above. This gives us, for every candidate node, the nearest certain node above and below. We delete all candidate nodes for which one of these distances is smaller than λ. We store the certain node nearest to the root, the certain node nearest to the hole and the total number of certain nodes, and from now on ignore certain nodes and consider only the remaining candidate nodes.
3. **Prune leaves to make their distances to the root non-decreasing:** let the i-th leaf, u_i, be connected with an edge of length y_i to a spine node at distance x_i from the root, and order the leaves so that $x_1 < x_2 < \ldots < x_s$. Note that $y_i < \frac{\lambda}{2}$, as otherwise u_i would be a certain node. Suppose that u_{i-1} is farther from the root than u_i (i.e., $x_{i-1}+y_{i-1} > x_i+y_i$), then: $d(u_i, u_{i-1}) = x_i - x_{i-1} + y_i + y_{i-1} = x_i + y_i - x_{i-1} + y_{i-1} < 2y_{i-1} < \lambda$. Therefore an optimal solution cannot contain both u_i and u_{i-1}. We claim that if the solution contains u_i then it can be replaced with u_{i-1}. To prove this, it is enough to argue that u_{i-1} is farther away from any node above it than u_i, and u_i is closer to any node below it than u_{i-1}. Consider a node u_j that is above u_{i-1} (so $j < i-1$), then: $d(u_j, u_{i-1}) - d(u_j, u_i) = y_{i-1} - (x_i - x_{i-1}) - y_i = x_{i-1} + y_{i-1} - (x_i + y_i) > 0$. Now consider a node u_j that is below u_i (so $j > i$), then: $d(u_j, u_{i-1}) - d(u_j, u_i) = y_{i-1} + (x_i - x_{i-1}) - y_i > 2(x_i - x_{i-1}) > 0$. So in fact, we can remove the i-th leaf from the caterpillar if $x_{i-1} + y_{i-1} > x_i + y_i$. To check this condition efficiently, we scan the caterpillar from top to bottom while maintaining the most recently processed non-removed leaf. This takes linear time in the number of candidate nodes and ensures that the distances of the remaining leaves from the root are non-decreasing.

4. **Prune leaves to make their distances to the hole non-increasing:** this is done as in the previous step, except we scan in the other direction.
5. **Preprocess for any candidate and certain node with respect to the hole:** we call $u_1, u_2, \ldots, u_i$ a *prefix* of the caterpillar and, similarly, $u_{i+1}, u_{i+2}, \ldots, u_s$ a *suffix*. For every possible prefix, we would like to precompute the result of running the linear-time feasibility test on that prefix. In Section 3.2 we will show that, in fact, this is enough to efficiently simulate running the feasibility test on the whole subtree rooted at r if we know the candidate and the certain node w.r.t. the hole. Consider running the feasibility test on $u_1, u_2, \ldots, u_i$. Recall that its goal is to choose as many nodes as possible, and in case of a tie to maximize the distance of the nearest chosen node to r. Due to distances of the leaves to r being non-decreasing, it is clear that u_i should be chosen. Then, consider the largest $i' < i$ such that $d(u_{i'}, u_i) \geq \lambda$. Due to distances of the leaves to the hole being non-decreasing, nodes $u_{i'+1}, u_{i'+2}, \ldots, u_{i-1}$ cannot be chosen and furthermore $d(u_j, u_i) \geq \lambda$ for any $j = 1, 2, \ldots, i'$. Therefore, to continue the simulation we should repeat the reasoning for $u_1, u_2, \ldots, u_{i'}$. This suggests the following implementation: scan the caterpillar from top to bottom and store, for every prefix $u_1, u_2, \ldots, u_i$, the number of chosen nodes, the certain node and the candidate node. While scanning we maintain i' in amortized constant time. After increasing i, we only have to keep increasing i' as long as $d(u_i, u_{i'}) \geq \lambda$. To store the information for the current prefix, copy the computed information for $u_1, u_2, \ldots, u_{i'}$ and increase the number of chosen nodes by one. Then, if the certain node is set to NULL, we set it to be u_i. If there is no $u_{i'}$, and u_i is the top-most chosen candidate, we need to set it to be the candidate (if $d(r, u_i) < \frac{\lambda}{2}$) or the certain node otherwise.

3.2 The feasibility test

The sublinear feasibility test for a value $\lambda \in (\lambda_1, \lambda_2)$ processes the tree bottom-up. For every fragment with root r, we would like to simulate running the linear-time feasibility test on the subtree rooted at r to compute: the number of chosen nodes, the candidate node, and the certain node. We assume that we already have such information for the fragment rooted at the hole of the current fragment. If the current fragment is active, we process it naively in $O(b)$ time using the linear-time feasibility test. If it is inactive, we process it (jump over it) in $O(\log b)$ time. This can be seen as, roughly speaking, attaching the hole as another spine node to the corresponding caterpillar and executing steps (2)-(5).

We start by considering the case where there is no candidate node w.r.t. the hole. Let v be the certain node w.r.t. the hole. Because distances of the leaves from the hole are non-increasing, we can compute the prefix of the caterpillar consisting of leaves that can be chosen, by binary searching for the largest i such that $d(v, u_i) \geq \lambda$. Then, we retrieve and return the result stored for $u_1, u_2, \ldots, u_i$ (after increasing the number of chosen nodes and, if the certain node is set to NULL, updating it to v).

Now consider the case where there is a candidate node u w.r.t. the hole. We start with binary searching for i as explained above. Then, we check if the distance between u and the certain node nearest to the hole is smaller than λ or $d(u_i, r) > d(u, r)$, and if so return the result stored for $u_1, u_2, \ldots, u_i$. Then, again because distances of the leaves to the hole are non-increasing, we can binary search for the largest $i' \leq i$ such that $d(u_{i'}, u) \geq \lambda$ (note that this also takes care of pruning leaves u_k that are closer to the hole than u). Finally, we retrieve and return the result stored for $u_1, u_2, \ldots, u_{i'}$ (after increasing the number of chosen nodes and possibly updating the candidate and the certain node).

We process every inactive fragment in $O(\log b)$ time and every active fragment in $O(b)$ time, so the total time is $O(\frac{n}{b} \cdot \log b) + O(\frac{n}{b^2} \cdot b) = O(\frac{n}{b} \cdot \log b)$.

3.3 The algorithm for the optimization problem

The general idea is to use a heavy path decomposition to solve the optimization problem with $O(\log^2 n)$ feasibility tests. The *heavy edge* of a non-leaf node of the tree is the edge leading to the child with the largest number of descendants. The heavy edges define a decomposition of the nodes into heavy paths. A heavy path p starts with a head $\mathsf{head}(p)$ and ends with a tail $\mathsf{tail}(p)$ such that $\mathsf{tail}(p)$ is a descendant of $\mathsf{head}(p)$, and its depth is the number of heavy paths p' s.t. $\mathsf{head}(p')$ is an ancestor of $\mathsf{head}(p)$. The depth is always $O(\log n)$ [15].

We process all heavy paths at the same depth together while maintaining an interval $[\lambda_1, \lambda_2)$ such that λ_1 is feasible while λ_2 is not, that is, the sought λ^* belongs to the interval. The goal of processing the heavy paths at depth d is to further shrink the interval so that, for any heavy path p at depth d, the result of running the feasibility test on any subtree rooted at $\mathsf{head}(p)$ is the same for any $\lambda \in [\lambda_1, \lambda_2)$ and therefore can be already determined. We start with the heavy paths of maximal depth and terminate with $\lambda^* = \lambda_1$ after having determined the result of running the feasibility test on the whole tree.

Let n_d denote the total size of all heavy paths at depth d. For every such heavy path we construct a caterpillar by replacing any subtree that hangs off by the certain and the candidate node (this is possible, because we have already determined the result of running the feasibility test on that subtree). To account for the possibility of including a node of the heavy path in the solution, we attach an artificial leaf connected with a zero-length edge to every such node. The caterpillar is then pruned similarly to steps (2)-(4) from Section 3.1, except that after having found the nearest certain node for every candidate node we cannot simply compare their distance to λ. Instead, we create an 1×1 matrix storing the relevant distance for every candidate node. Then, we apply Theorem 4 with $p = 0$ to the obtained set of $O(n_d)$ matrices of dimension 1×1. This allows us to determine, using only $O(\log n)$ feasibility tests and $O(n_d)$ time exclusive of the feasibility tests, which distances are larger than λ^*, so that we can prune the caterpillars and work only with the remaining candidate nodes. Then, for every caterpillar we create a row- and column-sorted matrix storing pairwise distance between its leaves. By applying Theorem 4 with $p = 0$ on the obtained set of square matrices of total side length $O(n_d)$ we can determine, with $O(\log n)$ feasibility tests and $O(n_d)$ time exclusive of the feasibility tests, which distances are larger than λ^*. This allows us to run the bottom-up procedure described in Section 2 to produce the candidate and the certain node for every subtree rooted at $\mathsf{head}(p)$, where p is a heavy path at depth d.

All in all, for every d we spend $O(n_d)$ time and execute $O(\log n)$ feasibility tests. Summing over all depths d, this is $O(n)$ plus $O(\log^2 n)$ calls to the feasibility test. Setting $b = \log^2 n$, the total time is thus $O(n + n \log\log n + \frac{n}{\log^2 n} \cdot \log\log n \cdot \log^2 n) = O(n \log\log n)$.

4 The Weighted Dispersion Problem

In this section we present an $O(n \log n)$ time algorithm for the weighted search problem (a matching lower bound is shown in the full version). As explained in the introduction, this then implies an $O(n \log^2 n)$ time solution for the optimization problem. Similarly to the unweighted case, we compute for each node of the tree, the subset of nodes P in its subtree s.t. $f(P) \geq \lambda$ and the total weight of P is maximized. We compute this by going over the nodes of the tree bottom-up. Previously, the situation was simpler, as for any subtree we had just one candidate node (i.e., a node that may or may not be in the optimal solution

for the entire input tree). This was true because nodes had uniform weights. Now however, there could be many candidates in a subtree, as the certain nodes are only the ones that are at distance at least λ from the root (and not $\frac{\lambda}{2}$ as in the unweighted case).

Let P be a subset of the nodes in the subtree rooted at v, and h be the node in P minimizing $d(h, v)$. We call h the *closest chosen node* in v's subtree. In our feasibility test, v stores an optimal solution P for each possible value of $d(h, v)$ (up to λ, otherwise the closest chosen node does not affect nodes outside the subtree). That is, a subset of nodes P in v's subtree, of maximal weight, s.t. the closest chosen node is at distance *at least* $d(h, v)$ from v, $f(P) \geq \lambda$. This can be viewed as a monotone polyline, since the weight of P (denoted $W(P)$) only decreases as the distance of the closest chosen node increases (from 0 to λ). $W(P)$ changes only at certain points called *breakpoints* of the polyline. Each point of the polyline is a key-value pair, where the key is $d(h, v)$ and the value is $W(P)$. We store with each breakpoint the value of the polyline between it and the next breakpoint, i.e., for a pair of consecutive breakpoints with keys a and $a + b$, the polyline value of the interval $(a, a + b]$ is associated with the former. The representation of a polyline consists of its breakpoints, and the value of the polyline at key 0.

The algorithm computes such a polyline for the subtrees rooted at every node v of the tree by merging the polylines computed for the subtrees rooted at v's children. We assume w.l.o.g. that the input tree is binary (for the same reasoning as in the unweighted case), and show how to implement this step in time $O(x \log(\frac{2y}{x}))$, where x is the number of breakpoints in the polyline with fewer breakpoints, and y is the number of breakpoints in the other.

Constructing a polyline. We now present a single step of the algorithm. We postpone the discussion of the data structure used to store the polylines for now, and first describe how to obtain the polyline of v from the polylines of its children. Then, we state the exact interface of the data structure that allows executing such a procedure efficiently, show how to implement such an interface, and finally analyze the complexity of the resulting algorithm.

If v has only one child, u, we build v's polyline by querying u's polyline for the case that v is in the solution (i.e., query u's polyline with distance of the closest chosen node being $\lambda - d(v, u)$), and add to this value the weight of v itself. We then construct the polyline by taking the obtained value for $d(h, v) = 0$ and merging it with the polyline computed for u, shifted to the right by $d(v, u)$ (since we now measure the weight of the solution as a function of the distance of the closest chosen node to v, not to u). The value between zero and $d(v, u)$ will be the same as the value of the first interval in the polyline constructed for u, so the shift is actually done by increasing the keys of all but the first breakpoint by $d(v, u)$.

If v has a left child u_1 and a right child u_2, we have two polylines p_1 and p_2 (that represent the solutions inside the subtrees rooted at u_1 and u_2), and we want to create the polyline p for the subtree rooted at v. Denote the number of breakpoints in p_1 by x and the number of breakpoints in p_2 by y. Assume w.l.o.g. that $x \leq y$. We begin with computing the value of p for key zero (i.e. v is in the solution). In this case we query p_1 and p_2 for their values with keys $\lambda - d(v, u_1)$ and $\lambda - d(v, u_2)$ respectively (if one of these is negative, we take zero instead), and add them together with the weight of v. Note that it is possible for the optimal solution in v's subtree not to include v. Therefore we need to check, after constructing the rest of the polyline, whether the value stored at the first breakpoint (which is the weight of the optimal solution where v is not included) is greater than the value we computed for the case v is chosen. If so, we store the value of the first breakpoint also as the value for key zero.

It remains to construct the rest of the polyline p. Notice that we need to maintain that $d(h_1, h_2) \geq \lambda$ (where h_1 is the closest chosen node in u_1's subtree and h_2 is the closest chosen

node in u_2's subtree). We start by shifting p_1 and p_2 to the right by $d(v, u_1)$ and $d(v, u_2)$ respectively, because now we measure the distance of h from v, not from u_1 or u_2. We then proceed in two steps, each computing half of the polyline p.

4.1 Constructing the second half of the polyline.

We start by constructing the second half of the polyline, where $d(h, v) \geq \frac{\lambda}{2}$. In this case we query both polylines with the same key, since $d(h_1, v) \geq \frac{\lambda}{2}$ and $d(h_2, v) \geq \frac{\lambda}{2}$ implies that $d(h_1, h_2) \geq \lambda$. The naive way to proceed would be to iterate over the second half of both polylines in parallel, and at every point sum the values of the two polylines. This would not be efficient enough, and so we only iterate over the breakpoints in the second half of p_1 (the smaller polyline). These breakpoints induce intervals of p_2. For each of these intervals we increase the value of p_2 by the value in the interval in p_1. This might require inserting some of the breakpoints from p_1, where there is no such breakpoint already in p_2. Thus, we obtain the second half of p by modifying the second half of p_2.

4.2 Constructing the first half of the polyline.

We need to consider two possible cases: either $d(h_1, v) < d(h_2, v)$ (i.e. the closest chosen node in v's subtree is inside u_1's subtree), or $d(h_1, v) > d(h_2, v)$ (h is in u_2's subtree). Note that in this half of the polyline $d(h, v) < \frac{\lambda}{2}$, and therefore $d(h_1, v) \neq d(h_2, v)$. For each of the two cases we will construct the first half of the polyline, and then we take the maximum of the two resulting polylines at every point, in order to have the optimal solution for each key.

Case I: $d(h_1, v) < d(h_2, v)$. Since we are only interested in the first half of the polyline, we know that $d(h_1, v) < \frac{\lambda}{2}$. Since $d(h_2, v) + d(h_1, v) \geq \lambda$ we have that $d(h_2, v) > \frac{\lambda}{2}$. Again, we cannot afford to iterate over the breakpoints of p_2, so we need to be more subtle.

We start by splitting p_1 at $\frac{\lambda}{2}$ and taking the first half (denoted by p_1'). We then split p_2 at $\frac{\lambda}{2}$ and take the second half (denoted by p_2'). Consider two consecutive breakpoints of p_1' with keys x and $x + y$. We would like to increase the value of p_1' in the interval $(x, x + y]$ s.t. the new value is the maximal weight of a valid subset of nodes from *both* subtrees rooted at u_1 and u_2, s.t. $x < d(h_1, v) \leq x + y$. Therefore $d(h_2, v) \geq \lambda - x - y$. p_2' is monotonically decreasing, and so we query it at $\lambda - x - y$, and increase by the resulting value.

This process might result in a polyline which is not monotonically decreasing, because as we go over the intervals of p_1' from left to right we increase the values there more and more. To complete the construction, we make the polyline monotonically decreasing by scanning it from $\frac{\lambda}{2}$ to zero and deleting unnecessary breakpoints. We can afford to do this, since the number of breakpoints in this polyline is no larger than the number of breakpoints in p_1. Note that we have assumed we have access to the original data structure representing p_2, but this structure has been modified to obtain the second half of p. However, we started with computing the second half of p only to make the description simpler. We can simply start with the first half.

Case II: $d(h_1, v) > d(h_2, v)$. Symmetrically to the previous case, we increase the values in the intervals of p_2 induced by the breakpoints of p_1 by the appropriate values of p_1 (similarly to what we do in Subsection 4.1). Again, the resulting polyline may be non-monotone, but this time we cannot solve the problem by scanning the new polyline and deleting breakpoints, since there are too many of them. Instead, we go over the breakpoints of the second half of p_1. For each such breakpoint with key k, we check if the new polyline has a breakpoint

with key $\lambda - k$. If so, denote its value by w, otherwise continue to the next breakpoint of p_1. These are the points where we might have increased the value of p_2. We then query the new polyline with a *value predecessor* query: this returns the breakpoint with the largest key s.t. its key is smaller than $\lambda - k$ and its value is at least w. If this breakpoint exists, and it is not the predecessor of the breakpoint at $\lambda - k$, then the values of the new polyline between its successor breakpoint and $\lambda - k$ should all be w (i.e. we delete all breakpoints in this interval and set the successor's value to w). If it does not exist, then the values between zero and $\lambda - k$ should be w (i.e. we delete all the previous breakpoints). This ensures that the resulting polyline is monotonically decreasing.

Merging cases I and II. We now need to build one polyline for the first half of the polyline, taking into account both cases. Let p_a and p_b denote the polylines we have constructed in cases I and II respectively (so the number of breakpoint in p_a is at most x, the number of breakpoints in p_b is at most y, and $x \leq y$).

We now need to take the maximum of the values of p_a and p_b, for each key. We do this by finding the intersection points of the two polylines. Notice that since both polylines are monotonically decreasing, these intersections can only occur at (i) the breakpoints of p_a, and (ii) at most one point between two consecutive breakpoints of p_a.

We iterate over p_a and for each breakpoint, we check if the value of p_b for the same key is between the values of this breakpoint and the predecessor breakpoint in p_a. If so, this is an intersection point. Then, we find the intersection points which are between breakpoints of p_a, by running a value predecessor query on p_b for every breakpoint in p_a except for the first. After such computation, we know which polyline gives us the best solution for every point between zero and $\frac{\lambda}{2}$, and where are the intersection points where this changes. We can now build the new polyline by doing insertions and deletions in p_b according to the intersection points: For every interval of p_b defined by a pair of consecutive intersection points, we check if the value of p_a is larger than the value of p_b in the interval, and if so, delete all the breakpoints of p_b in the interval, and insert the relevant breakpoints from p_a. The number of intersection points is linear in the number of breakpoints of p_a, and so the total number of interval deletions and insertions is $O(x)$.

To conclude, the final polyline p is obtained by concatenating the value computed for key zero, the polyline computed for the first half, and the polyline computed for the second half.

4.3 The polyline data structure

We now specify the data structure for storing the polylines. The required interface is:

1. Split the polyline at some key.
2. Merge two polylines (s.t. all keys in one polyline are smaller than all keys in the other).
3. Retrieve the value of the polyline for a certain key $d(h, v)$.
4. Return a sorted list of the breakpoints of the polyline.
5. Batched interval increase – Given a list of disjoint intervals of the polyline, and a number for each interval, increase the values of the polyline in each interval by the appropriate number. Each interval is given by the keys of its endpoints.
6. Batched value predecessor – Given a list of key-value pairs, (k_i, v_i), find for each k_i, the maximal key k'_i, s.t. $k'_i < k_i$ and the value of the polyline at k'_i is at least v_i, assuming that the intervals (k'_i, k_i) are disjoint.
7. Batched interval insertions – Given a list of pairs of consecutive breakpoints in the polyline, insert between each pair a list of breakpoints.

8. Batched interval deletions – Given a list of disjoint intervals of the polyline, delete all the breakpoints inside the intervals.

We now describe the data structure implementing the above interface. We represent a polyline by storing its breakpoints in an augmented 2-3 tree, where the data is stored in the leaves. Each node stores a key-value pair, and we maintain the following property: the key of each breakpoint is the sum of the keys of the corresponding leaf and of all its ancestors, and similarly for the values. In addition, we store in each node the maximal sum of keys and values on a path from that node to a leaf in its subtree. We also store in each node the number of leaves in its subtree. Operations 1 and 2 use standard split and join procedures for 2-3 trees in logarithmic time. Operation 3 runs a predecessor query and returns the value at the returned breakpoint in logarithmic time. Operation 4 is done by an inorder traversal of the tree (of p_1 in $O(x)$ time). Operations 1-4 are performed only a constant number of times per step, and so their total cost is $O(\log x + \log y + x)$. The next four operations are more costly, since they consists a batch of $O(x)$ operations given in sorted order (by keys).

Operation 5 – batched interval increase. Consider the following implementation for Operation 5. We iterate over the intervals, and for each of them, we find its left endpoint, and traverse the path from the left endpoint, through the LCA, to the right endpoint. The traversal is guided by the maximal key stored in the current node (that are used to find the maximal key of a breakpoint stored in its subtree by adding the sum of all keys from the root to the current node, which is maintained in constant time after moving to a child or the parent). While traversing the path from the left endpoint to the LCA (from the LCA to the right endpoint), we increase the value of every node hanging to the right (left) of this path. We also update the maximal value field in each node we reach (including the nodes on the path from the LCA to the root). Notice that if one of the endpoints of the interval is not in the structure, we need to insert it. We might also need to delete a breakpoint if it is a starting point of some interval and its new value is now equal to the value of its predecessor. This implementation would take time which is linear in the number of traversed nodes, plus the cost of insertions and deletions (whose number is linear in the number of intervals). Because the depth of a 2-3 tree of size $O(y)$ is $O(\log y)$, this comes up to $O(x \log y)$. Such time complexity for each step would imply $O(n \log^2 n)$ total time for the feasibility test.

We improve the running time by performing the operations on smaller trees. The operation therefore begins by splitting the tree into $O(x)$ smaller trees, each with $O(\frac{y}{x})$ leaves. This is done by recursively splitting the tree, first into two trees with $O(\frac{y}{2})$ leaves, then we split each of these trees into two trees with $O(\frac{y}{4})$ leaves, and so on, until we have trees of size $O(\frac{y}{x})$. We then increase the values in the relevant intervals using the small trees. For this, we scan the roots of the small trees, searching for the left endpoint of the first interval (by using the maximal key stored in the root of each tree). Once we have found the left endpoint of the interval, we check if the right endpoint of the interval is in the same tree or not (again, using the maximal key). In the first case, the interval is contained in a single tree, and can be increased in this tree in time $O(\log(\frac{2y}{x}))$ using the procedure we have previously described. In the second case, the interval spans several trees, and so we need to do an interval increase in the two trees containing the endpoints of the interval, and additionally increase the value stored in the root of every tree that is entirely contained in the interval. We then continue to the next interval, and proceed in the same manner. Since the intervals are disjoint and we do at most two interval increases on small trees per interval, the total time for the increases in the small trees is $O(x \cdot \log(\frac{2y}{x}))$. Scanning the roots of the small trees adds $O(x)$ to the complexity, leading to $O(x \cdot \log(\frac{2y}{x}) + x) = O(x \log(\frac{2y}{x}))$ overall for processing the small trees.

Before the operation terminates, we need to join the small trees to form one large tree. This is symmetric to splitting and analyzed with the same calculation.

▶ **Lemma 5.** *The time to obtain the small trees is* $O(x \log(\frac{2y}{x}))$.

The cost of all joins required to patch the small trees together can be bounded by the same calculation as the cost of the splits made to obtain them, and so the operation takes $O(x \log(\frac{2y}{x}))$ time in total. The rest of the batched operations are also done by splitting the tree into small trees. There is an additional technical difficulty in Operation 6, as in our case the intervals $(k_{i'}, k_i)$ might not be disjoint. We make them disjoint with some extra work. In Operation 7, some of the small trees might become much larger due to the insertions. This also requires some extra work, see the full version for a complete description.

▶ **Theorem 6.** *The above implementation implies an* $O(n \log n)$ *weighted feasibility test.*

References

1 R.I. Becker, S.R. Schach, and Y. Perl. A shifting algorithm for min-max tree partitioning. *J. ACM*, 29(1):56–67, 1982.

2 B.K. Bhattacharya and M.E. Houle. Generalized maximum independent sets for trees. In *CATS*, pages 17–25, 1997.

3 B.K. Bhattacharya and M.E. Houle. Generalized maximum independent sets for trees in subquadratic time. In *ISAAC*, pages 435–445, 1999.

4 M.R. Brown and R.E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

5 Richard Cole. Slowing down sorting networks to obtain faster sorting algorithms. *J. ACM*, 34(1):200–208, 1987.

6 G.N. Frederickson. Optimal algorithms for partitioning trees and locating p-centers in trees. Technical Report CSD-TR-1029, Purdue University, 1990.

7 G.N. Frederickson. Optimal algorithms for tree partitioning. In *SODA*, pages 168–177, 1991.

8 G.N. Frederickson. Parametric search and locating supply centers in trees. In *WADS*, pages 299–319, 1991.

9 G.N. Frederickson and D.B. Johnson. Finding k-th paths and p-centers by generating and searching good data structures. *J. Algorithms*, 4(1):61–80, 1983.

10 Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Inf.*, 17:157–184, 1982.

11 N. Megiddo and A. Tamir. New results on the complexity of p-center problems. *SIAM J. Computing*, 12(3):751–758, 1983.

12 N. Megiddo, A. Tamir, E. Zemel, and R. Chandrasekaran. An $O(n \log^2 n)$ algorithm for the k-th longest path in a tree with applications to location problems. *SIAM J. Computing*, 10(2):328–337, 1981.

13 Y. Perl and S.R. Schach. Max-min tree partitioning. *J. ACM*, 28(1):5–15, 1981.

14 D.B. Shmoys, É. Tardos, and K. Aardal. Approximation algorithms for facility location problems. In *STOC*, pages 265–274, 1997.

15 D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

16 V.V. Vazirani. *Approximation Algorithms*. Springer, 2003.

Real-Time Streaming Multi-Pattern Search for Constant Alphabet*

Shay Golan[1] and Ely Porat[2]

1 **Bar Ilan University, Ramat Gan, Israel**
golansh1@cs.biu.ac.il
2 **Bar Ilan University, Ramat Gan, Israel**
porately@cs.biu.ac.il

Abstract

In the streaming multi-pattern search problem, which is also known as the streaming dictionary matching problem, a set $D = \{P_1, P_2, \ldots, P_d\}$ of d patterns (strings over an alphabet Σ), called the *dictionary*, is given to be preprocessed. Then, a text T arrives one character at a time and the goal is to report, before the next character arrives, the longest pattern in the dictionary that is a current suffix of T. We prove that for a constant size alphabet, there exists a randomized Monte-Carlo algorithm for the streaming dictionary matching problem that takes constant time per character and uses $\mathcal{O}(d \log m)$ words of space, where m is the length of the longest pattern in the dictionary. In the case where the alphabet size is not constant, we introduce two new randomized Monte-Carlo algorithms with the following complexities:

- $\mathcal{O}(\log\log|\Sigma|)$ time per character in the worst case and $\mathcal{O}(d \log m)$ words of space.
- $\mathcal{O}(\frac{1}{\varepsilon})$ time per character in the worst case and $\mathcal{O}(d|\Sigma|^{\varepsilon} \log \frac{m}{\varepsilon})$ words of space for any $0 < \varepsilon \leq 1$.

These results improve upon the algorithm of Clifford et al. [12] which uses $\mathcal{O}(d \log m)$ words of space and takes $\mathcal{O}(\log\log(m + d))$ time per character.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases multi-pattern, dictionary, streaming pattern matching, fingerprints

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.41

1 Introduction

We consider one of the most fundamental pattern matching problems, the *dictionary matching problem* [12, 16, 5, 6, 30, 20, 7, 21, 17, 18, 4], where a set of patterns $D = \{P_1, P_2, \ldots, P_d\}$, called the *dictionary*, is given along with a string T, called the *text*, such that each pattern P_i is a string of length m_i, and all the strings are over an alphabet Σ. The goal is to find all the occurrences of patterns from D in T. The dictionary matching problem is a natural generalization of the simple pattern matching problem of one pattern, and it has many applications in different areas. For example, in the area of Intrusion Detection and Anti-Viruses systems [36], the goal is to detect viruses in a stream of data by looking for known digital signatures of these viruses. Due to the importance of the problem, significant efforts have been made to speed up algorithms for this problem, for example, by using GPUs [38, 39, 37, 40, 43, 26] or even using a designated hardware [15, 2, 41, 29, 42, 9].

* This work is supported in part by ISF grant 1278/16, and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 683064).

The streaming model. In the *streaming* model [3, 25, 32, 28], we have a stream of data to process in near real-time while using only sublinear space. For pattern matching problems, the pattern is given in advance and the text arrives one character at a time, and the goal is to decide after the arrival of each character, whether the current suffix of the text matches the preprocessed pattern. Since the seminal paper of Porat and Porat [33] which introduced the first algorithm for this problem in the streaming model, there has been a rising interest in solving pattern matching problems in the streaming model [10, 14, 31, 11, 27, 12, 13, 22].

For the dictionary matching problem in the streaming model, D is given in advance, and the text T arrives one character at a time. After the arrival of $T[q]$ the algorithm must report the *longest*[1] suffix of $T[1..q]$, which is a pattern in D. The space usage of the algorithm is limited to sublinear space, hence, one cannot even store the dictionary D explicitly. The efficiency of algorithms in this model is measured by the amount of time required to process a text character and the total space usage of the algorithm. Another closely related model is the *online* model, which is the same as the streaming model without the constraint of using sublinear space.

Previous results and related work. The current most efficient algorithm for the streaming dictionary matching is due to Clifford et al. [12] which uses $\mathcal{O}(d \log m)$ words[2] of space and takes $\mathcal{O}(\log\log(d+m))$ time per character, where $m = \max\{m_i\}$ is the length of the longest pattern. This algorithm assumes that there are no two patterns $P_i, P_j \in D$ such that P_i is a suffix of P_j. Otherwise, the algorithm reports any time *some* pattern that is a current suffix of the text, but not necessarily the longest one. The algorithm is a randomized Monte-Carlo algorithm and is correct with high probability.

In the online model, most of the algorithms are variations of the Aho and Corasick [1] algorithm. This algorithm has two versions, the DFA (deterministic finite automaton) and the state-machine. The DFA version takes $\mathcal{O}(1)$ time per character and uses $\mathcal{O}(M|\Sigma|)$ words of space, where $M = \sum_{i=1}^{d} m_i$ is the sum of the patterns' length. The state-machine version uses only $\mathcal{O}(M)$ words of space, and its amortized running time is also constant. However, for the online model, which measures the running time per character, in the worst case the state-machine version takes $\Omega(m)$ time per character, which is unreasonable. Hence, the algorithm of Kopelowitz et al. [30], improves the state-machine algorithm, to $\mathcal{O}(\log\log|\Sigma|)$ time per character, and it still uses $\mathcal{O}(M)$ words of space and $\mathcal{O}(1)$ amortized time per character. Both Aho and Corasick [1] and Kopelowitz et al. [30] algorithms are deterministic, and we use some of their concepts in our results.

Our results. Our first result is for the case of a constant alphabet and is stated in the following theorem:

▶ **Theorem 1.** *For a constant size alphabet, there exists a randomized Monte-Carlo algorithm for the streaming dictionary matching problem that succeeds with probability* $1 - 1/poly(n)$*, spends constant time per arriving text character and uses* $\mathcal{O}(d \log m)$ *words of space.*

---

[1] This is a common simplification in which one must only report the longest pattern that has arrived (if several patterns end at the same text location), since converting such a solution to one that reports all the patterns is straightforward with additional time which is linear in the number of reported patterns, and this way the focus is on the time cost that is independent from the output size.

[2] We assume the RAM model where each word has size of $\Omega(\log n)$ bits

We mention the open problem of Breslauer and Galil [10] who solve the problem for the case of one pattern. They ask whether $\Omega(d \log m)$ words of space is required for any streaming dictionary matching algorithm. If the answer to this problem is positive, then the algorithm of Theorem 1 has optimal time and space.

For the general case where the alphabet size is arbitrary, we introduce two new algorithms, and each of them suffices as a proof for Theorem 1. The first algorithm is an improvement over the algorithm of Clifford et al. [12] with the same space usage, but with running time of $\mathcal{O}(\log\log|\Sigma|)$ time per character, compared to $\mathcal{O}(\log\log(m+d))$ time per character of [12]. Moreover, our algorithm solves the stronger version of the problem where the algorithm has to report the *longest*[1] pattern in D which is a current suffix of the text.

▶ **Theorem 2.** *There exists a randomized Monte-Carlo algorithm for the streaming dictionary matching problem that succeeds with probability* $1 - 1/poly(n)$*, spends* $\mathcal{O}(\log\log|\Sigma|)$ *time per arriving text character and uses* $\mathcal{O}(d \log m)$ *words of space.*

We point out that even though someone who is familiar with the area would expect an *amortized* $\mathcal{O}(1)$ time per character for the algorithm of Theorem 2, unfortunately this is not the case. In the algorithm for the online model by Kopelowitz et al. [30], the algorithm has a tree of states, and after the arrival of a character the algorithm moves to a state which its depth is larger than the former state's depth by at most one. Thus, the amortized analysis of this algorithm was based on the depth of algorithm's state. In our algorithm, we also have states with depths but we could sometimes jump into a state that is much deeper than the former state, therefore, such an amortized analysis will not hold. An interesting question is whether one can design an algorithm with the same space usage and worst case time per character, but with amortized $\mathcal{O}(1)$ time per character.

Our second algorithm is a real-time algorithm, with a small amount of extra space.

▶ **Theorem 3.** *For any constant* $0 < \varepsilon \leq 1$ *there exists a randomized Monte-Carlo algorithm for the streaming dictionary matching problem that succeeds with probability* $1 - 1/poly(n)$*, spends* $\mathcal{O}(\frac{1}{\varepsilon})$ *time per arriving text character and uses* $\mathcal{O}(d|\Sigma|^{\varepsilon} \log \frac{m}{\varepsilon})$ *words of space.*

1.1 Algorithmic Overview

We prove simultaneously Theorem 2 and a degenerate version of Theorem 3, where $\varepsilon = 1$, as stated in the following lemma:

▶ **Lemma 4.** *There exists a randomized Monte-Carlo algorithm for the streaming dictionary matching problem that succeeds with probability* $1 - 1/poly(n)$*, spends* $\mathcal{O}(1)$ *time per arriving text character and uses* $\mathcal{O}(d|\Sigma| \log m)$ *words of space.*

Then, Theorem 3 is deduced from Lemma 4 by implying the following theorem of Rozen [34].

▶ **Theorem 5.** *Let* $\mathcal{A}$ *be an algorithm for the online dictionary pattern matching problem which uses* $\mathcal{O}(s_{\mathcal{A}}(d, m, |\Sigma|))$ *words of space and takes* $\mathcal{O}(t_{\mathcal{A}}(d, m, |\Sigma|))$ *time per character. Then, for any* $0 < \varepsilon \leq 1$*, there exists an algorithm* $\mathcal{A}_{\varepsilon}$ *for this problem which uses* $\mathcal{O}(s_{\mathcal{A}}(d, \frac{m}{\varepsilon}, |\Sigma|^{\varepsilon}))$ *words of space and takes* $\mathcal{O}(\frac{1}{\varepsilon} t_{\mathcal{A}}(d, \frac{m}{\varepsilon}, |\Sigma|^{\varepsilon}))$ *time per character.*

The algorithms for Theorem 2 and Lemma 4 are very similar and have only small number of differences, therefore we describe them as one algorithm, and demonstrate only the differences. We follow the basic partition of D, presented by Clifford et al. [12], into three types of patterns. The types are short patterns, long patterns with a small period length, and long patterns with a large period length. We introduce an algorithm for each type, $\mathcal{A}_1$, $\mathcal{A}_{2a}$, and $\mathcal{A}_{2b}$, respectively. Theorem 2 and Lemma 4 are obtained by running all three algorithms in parallel.

Each one of the algorithms $\mathcal{A}_1$, $\mathcal{A}_{2a}$, and $\mathcal{A}_{2b}$ is composed of two phases. At the high level, the algorithm considers for each pattern $\log m$ prefixes, called *heads* of the pattern. For a pattern P_i of length m_i, the algorithm considers all the prefixes of length $\ell \in (m_i - 2\log m, m_i - \log m]$. Thus, the total number of heads is at most $d \log m$. The algorithm utilizes the fact that each occurrence of P_i in the text must begin with an occurrence of some head, such that this occurrence ends at a position which is a multiple of $\log m$.[3]

In the first phase, at each text position that is a multiple of $\log m$, the algorithm finds the current longest suffix that is a head of some pattern. The running time of the first phase is as stated in Theorem 2 or Lemma 4, with additional $\mathcal{O}(\log m)$ running time for each text position that is a multiple of $\log m$. This runtime is de-amortized during the arrival of $\log m$ characters between each two such positions. We introduce the first phase of $\mathcal{A}_1$ in Section 4 and the first phase of $\mathcal{A}_{2a}$ and $\mathcal{A}_{2b}$ in Section 5.

In the second phase, after finding the longest suffix that is a head of some pattern, the algorithm reads the text one character at a time using a state machine, which is inspired by the Aho and Corasick [1] algorithm. The initial state is obtained by the longest head that is found in the first phase, and each state transition is done according to the character that arrived. Whenever a pattern in the dictionary is a current suffix of the text, the state of the machine represents this pattern or a longer string which this pattern is its suffix. Hence, the algorithm has the correct pattern to report at any time. The details of the second phase appear in Section 3.

2 Preliminaries

A string S of length $|S| = \ell$ is a sequence of characters $S[1]S[2]\dots S[\ell]$ over an alphabet Σ. A *substring* of S is denoted by $S[x..y] = S[x]S[x+1]\dots S[y]$ for $1 \le x \le y \le \ell$. If $x = 1$, the substring is called a *prefix* of S, and if $y = \ell$, the substring is called a *suffix* of S.

A prefix of S of length $y \ge 1$ is called a *period* of S if and only if $S[i] = S[i+y]$ for all $1 \le i \le \ell - y$. The shortest period of S is called *the principal period* of S, and its length is denoted by ρ_S. If $\rho_S \le \frac{|S|}{2}$ we say that S is *periodic*.

The proof of this lemma and other lemmas will appear in the final version of this paper.

▶ **Lemma 6.** *Let u be a periodic string with principal period length ρ_u. If v is a substring of u of length at least $2\rho_u$ then $\rho_u = \rho_v$.*

The *cyclic shift of* S is $\sigma(S) = S[2..\ell]S[1]$. For any $0 \le i < \ell$ the i^{th} cyclic shift of S is $\sigma^i(S) = S[i+1..\ell]S[1..i]$.

Fingerprints. For a natural number n we denote $[n] = \{1, 2, \dots, n\}$. For the following let $u, v \in \bigcup_{i=0}^{n} \Sigma^i$ be two strings of length at most n. Porat and Porat [33] and Breslauer and Galil [10] proved that for every constant $c > 1$ there exists a *fingerprint function* $\phi : \bigcup_{i=0}^{n} \Sigma^i \to [n^c]$, such that:

1. If $|u| = |v|$ and $u \ne v$ then $\phi(u) \ne \phi(v)$ with high probability (at least $1 - \frac{1}{n^{c-1}}$).
2. *The sliding property:* Let w=uv be the concatenation of u and v. If $|w| \le n$, then given the length and the fingerprints of any two strings from u,v and w, one can compute the fingerprint of the third string in constant time.

[3] For the sake of simplicity we assume that $\log m$ is an integer, if this is not the case we use $\lceil \log m \rceil$ instead.

Our algorithms often uss fingerprints in order to quickly validate if two strings are equal or not. To ease presentation, in the rest of the paper we assume that fingerprints never give false positives. This assumption is covered by the algorithms only failing with small probability.

2.1 Multi-labeled Trees with Lowest Labeled Ancestor queries

Let $\mathcal{L}$ be a set of labels of size $|\mathcal{L}| = \lambda$. A multi-labeled tree T is a rooted tree such that each node $v \in T$ is associated with some $\mathcal{L}_v \subseteq \mathcal{L}$. For each $v \in T$ and $\ell \in \mathcal{L}$, the lowest label ancestor $\mathsf{LLA}(v, \ell)$ is the lowest node u on the path from the root of T to v such that $\ell \in \mathcal{L}_u$, or $\perp$ if such u does not exist. We denote the total size of all the labels among the entire tree by $M = \sum_{v \in T} |\mathcal{L}_v|$. The following theorem is due to Kopelowitz et al. [30].

▶ **Theorem 7** (Deduced from [30, Theorem 3]). *For any multi-labeled tree T with a label set $\mathcal{L} = \{1, 2, \ldots, \lambda\}$, there exists a data structure that supports* LLA *queries in $\mathcal{O}(\log\log\lambda)$ time and uses $\mathcal{O}(n + M)$ words of space where n is the size of the tree and $M = \sum_{v \in T} |\mathcal{L}_v|$.*

3 The Second Phase Algorithm

In this section, we introduce the second phase of algorithms $\mathcal{A}_1$, $\mathcal{A}_{2a}$, and $\mathcal{A}_{2b}$. For each $P_i \in D$ we define the j^{th} head of P_i to be $P_i[1..m_i - j]$. The j^{th} heads set is $\mathsf{Heads}_j(D) = \{P_i[1..m_i - j] \,|\, P_i \in D\,,\, |P_i| \geq j\}$ and for a set of lengths L we define:

$$\mathsf{Heads}_L(D) = \bigcup_{j \in L} \mathsf{Heads}_j(D) = \{P_i[1..m_i - j] \,|\, P_i \in D\,,\, j \in L\,,\, |P_i| \geq j\}$$

We assume that an algorithm $\mathcal{A}$ for the first phase is given such that at each text position q that is a multiple of $\log m$, $\mathcal{A}$ finds the longest string in $\mathsf{Heads}_{[\log m, 2\log m)}(D)$ that is a suffix of $T[1..q]$. The time per character and space usage of $\mathcal{A}$ matches Theorem 2 or Lemma 4, with additional $\mathcal{O}(\log m)$ time per text position that is a multiple of $\log m$.

Our algorithm uses concepts from the Aho and Corasick algorithm [1] and especially from its online version of Kopelowitz et al. [30]. The main idea in our implementation is that instead of creating the complete Aho and Corasick state machine, we create only states that correspond to strings in $\mathsf{Heads}_{[0,2\log m)}(D)$, which are all the $2\log m$ longest prefixes of each pattern in the dictionary. Thus, the number of states is $\mathcal{O}(d \log m)$. To overcome the missing states, the algorithm uses the pattern prefixes reported by $\mathcal{A}$ to jump into the correct state soon enough, before it has to report on a pattern occurrence.

The algorithm creates a state v_S for each $S \in \mathsf{Heads}_{[0,2\log m)}(D)$ and one additional state v_ε for the empty string. In addition, the algorithm creates a perfect hash table [19, 23, 24, 35] that stores a pointer from the fingerprint of any $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D)$ to the state v_S. Another perfect hash table stores for the fingerprint of each $S \in \mathsf{Heads}_{[0,2\log m)}(D)$ the index of the longest pattern in D which is a suffix of S, if such a pattern exists.

Intuitively, the goal is that whenever the machine's state is v_S and the character that arrived is ω, the machine transits into the state $v_{S'}$ where S' is the longest suffix of $S\omega$ among all the states' strings. Since v_ε exists, such a transition is always well defined. For the algorithm of Lemma 4 each state stores explicitly all the $|\Sigma|$ transitions that correspond to any possible character, as in DFA. However, for Theorem 2 this goal is apparently impossible without a factor of $|\Sigma|$ for the space usage. Therefore, we are satisfied with a slightly weaker property, which is sufficient for the goal of reporting all patterns' occurrences. In the following paragraphs we introduce the details of the algorithm for Theorem 2.

State machine for Theorem 2. For each state v_S that represents the string S and for each $\omega \in \Sigma$, if there exists a state corresponding to the string $S' = S\omega$, the algorithm has a *goto link* from S to S' with the label ω. All the goto links of v_S are maintained in a perfect hash table due to their labels. In addition, v_S has a *failure link* to v_{S^*} where S^* is the longest proper suffix of S that has a state in the machine. Since the empty string has a state, v_ε, the failure link is defined for every state, except for v_ε itself.

The failure tree. We define the *failure tree* of the state machine, T_{fail}, as the tree induced by the states of the machine and the failure links. Since each state has exactly one failure link, except for v_ε, T_{fail} is well defined. We consider T_{fail} as a multi-labeled tree, with $\mathcal{L} = \Sigma$ as the set of labels, and for each state $v_S \in T_{\mathsf{fail}}$ and $\omega \in \Sigma$ we have that $\omega \in \mathcal{L}_{v_S}$ if and only if v_S has a goto link with the character ω. The algorithm creates the data structure of Theorem 7, which supports LLA queries in $\mathcal{O}(\log\log|\Sigma|)$ time on T_{fail}.

Performing a transition. When the machine's state is v_S and the character ω arrives, the algorithm performs the following transition. Firstly, if v_S has a goto link with label ω, the algorithm uses this link and moves into $v_{S\omega}$. If such a link does not exist, the algorithm performs the $\mathsf{LLA}(v_S, \omega)$ query. Let $v_{S'} \in T$ be the result of the query, then $v_{S'}$ has a goto link with the label ω, and the algorithm uses this link to move into $v_{S'\omega}$. If $\mathsf{LLA}(v_S, \omega) = \perp$, the algorithm moves to state v_ε. This ends the special part of the algorithm for Theorem 2.

Text processing. The second phase algorithm runs $\mathcal{A}$ to process each text character that arrives. On each position q which is a multiple of $\log m$ the algorithm creates a new process, which is alive until the time $T[q + 2\log m]$ arrives, and then the process is terminated by the algorithm. Hence, at any time the algorithm runs two processes.

Focus on the process that starts when $T[q]$ arrives for q which is a multiple of $\log m$. While the first $\frac{\log m}{2}$ characters $(T[q], \ldots, T[q + \frac{\log m}{2} - 1])$ arrive, the algorithm executes $\mathcal{A}$ for $\mathcal{O}(\log m)$ time to retrieve $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D)$ which is the longest suffix of $T[1..q]$, and keeps a buffer of the arriving characters. This execution takes $\mathcal{O}(1)$ time per character by standard de-amortization. Then, when the subsequent $\frac{\log m}{2}$ characters arrive the algorithm uses the buffer, and performs all the $\log m$ transitions beginning at v_S. Thus, by performing two transitions per arriving character, using the buffer, when $T[q+\log m]$ arrives, the machine already performed the transitions corresponding to the first $\log m$ characters.

At the following $\log m$ characters, the algorithm continues to perform transitions according to the text characters. Whenever the machine is in a state $v_{S'}$, the algorithm reports the longest suffix of S' that is a pattern in D, as the current longest suffix of the text that is a pattern in D, using the preprocessed hash table.

Due to the following lemma, the machine's state corresponds to a sufficiently long suffix of the text at any time. In particular, while processing the last $\log m$ characters of each process, if some P_i is a suffix of the text, then P_i is also a suffix of the machine's state string.

▶ **Lemma 8.** *Consider the process that starts when $T[k\log m]$ arrives. Let v_S be the state of the machine after processing $T[k\log m + i]$ for $0 \le i < 2\log m$. Then, the longest suffix of $T[1..k\log m + i]$ in $\mathsf{Heads}_{[\max\{0,\log m - i\}, 2\log m - i)}(D)$ is a suffix of S.*

Hence, since at any time there is a process which reads the arriving character as part of its last $\log m$ characters and reports matches, we deduce the following corollary.

▶ **Corollary 9.** *When $T[q]$ arrives, the algorithm reports the longest pattern in D that is a suffix of $T[1..q]$.*

Complexities. The number of states in the machine is $\mathcal{O}(|\mathsf{Heads}_{[0,2\log m)}(D)|) = \mathcal{O}(d\log m)$. For the algorithm of Lemma 4, each state has $|\Sigma|$ links, one for each $\omega \in \Sigma$. Therefore, the space usage of the second phase is $\mathcal{O}(d|\Sigma|\log m)$. The second phase for Lemma 4 takes $\mathcal{O}(1)$ time per character, since each transition is performed in constant time.

For the algorithm of Theorem 2, each state v_S has at most one goto link into v_S and one failure link from v_S. Therefore, there are $\mathcal{O}(d\log m)$ links. Since T_{fail} has $\mathcal{O}(d\log m)$ nodes, and the total size of all the nodes' labels sets is exactly the number of goto links, then the LLA data structure uses $\mathcal{O}(d\log m)$ words of space. Therefore, the total space usage of the algorithm is $\mathcal{O}(d\log m)$ words. The processing of each character requires a LLA query, thus, the second phase of Theorem 2 takes $\mathcal{O}(\log\log|\Sigma|)$ time per character.

4 Short Patterns

For very short strings $P_i \in D$ of length at most $2\log m$, we use the algorithm of Aho and Corasick [1]. More specifically, for the algorithm of Theorem 2, we use the version of Kopelowitz et al. [30], which takes $\mathcal{O}(\log\log|\Sigma|)$ time per character in the worst-case and uses $\mathcal{O}(d\log m)$ words of space. For the algorithm of Lemma 4, we use the DFA version of [1, Section 6], which takes $\mathcal{O}(1)$ time per character and uses $\mathcal{O}(d|\Sigma|\log m)$ words of space.

In this section, we introduce the first phase of $\mathcal{A}_1$, which deals with patterns of $D_1 = \{P_i \in D \mid 2\log m < m_i \leq 8d\log m\}$. Intuitively, at each position that is a multiple of $\log m$ the algorithm performs kind of a binary search for the longest text suffix which is a string in $\mathsf{Heads}_{[\log m,2\log m)}(D_1)$, similarly to the *fat binary search* of Belazzougui et al. [8]. We define the *text fingerprint* of position q as $\phi(T[1..q])$. The algorithm maintains a sliding window of the last $8d\log m$ text fingerprints. Maintaining this window takes $\mathcal{O}(1)$ time per character using the sliding property of ϕ. Using this sliding window, for any $0 \leq \ell < 8d\log m$, one can compute the fingerprint of $T[q-\ell+1..q]$ in constant time. Hence, if the algorithm had all the fingerprints of all the suffixes of strings from $\mathsf{Heads}_{[\log m,2\log m)}(D_1)$, the algorithm can easily perform the binary search where for each length it would make one query on a perfect hash table that maintains the suffixes' fingerprints. However, there exist too many such suffixes to maintain in a perfect hash table.

Let $i_{max} = \lceil \log_2 \min\{m, 8d\log m\}\rceil$ be the number of bits required to represent the lengths of patterns in D_1. The algorithm performs the binary search on the interval $[0, 2^{i_{max}}]$, such that at each iteration the length of range considered by the algorithm is a power of 2. For each $P \in \mathsf{Heads}_{[\log m,2\log m)}(D_1)$ the algorithm maintains the fingerprints of all the suffixes whose lengths may be queried by the binary search. These lengths are exactly the lengths whose binary representation is the same as the binary representation of $|P|$, except for some suffix of the representation that is replaced by zeros. Let $\Delta_{|P|} = \{|P| - (|P| \bmod 2^i) \mid 0 \leq i \leq i_{max}\}$ be the set of suffixes lengths for the string P. We define $\mathsf{Suffixes}_1 = \bigcup_{P\in\mathsf{Heads}_{[\log m,2\log m)}(D_1)}\{P[|P|-\ell+1..|P|] \mid \ell \in \Delta_{|P|}\}$ to be the set of all suffixes that may be queried by the binary search. Notice that $|\Delta_{|P|}| \leq i_{max} \leq \lceil\log m\rceil$ and therefore the total size of $\mathsf{Suffixes}_1$ is $\mathcal{O}(d\log^2 m)$. Given a perfect hash table that maintains the fingerprints of all the strings in $\mathsf{Suffixes}_1$, the algorithm is able to find the longest string in $\mathsf{Suffixes}_1$ that is a current suffix of the text by a binary search. At each iteration, the algorithm computes the fingerprint of some suffix of $T[1..q]$ and queries the hash table with this fingerprint to validate that this suffix is in $\mathsf{Suffixes}_1$. By maintaining with each $S \in \mathsf{Suffixes}_1$, the longest suffix of S from $\mathsf{Heads}_{[\log m,2\log m)}(D_1)$, the algorithm is able to report in $\mathcal{O}(\log m)$ time the longest string from $\mathsf{Heads}_{[\log m,2\log m)}(D_1)$ that is suffix of the text. However, storing such a perfect hash table takes $\mathcal{O}(d\log^2 m)$ words of space, which is too much. Thus, we have to reduce the space usage to $\mathcal{O}(d\log m)$.

Suffixes tree. In order to reduce the number of strings, we consider for each $S \in \mathsf{Suffixes}_1$ the string $p(S)$ which must precede S in the binary search as the *parent string* of S. Formally, if S is a suffix of some $P \in \mathsf{Heads}_{[\log m, 2\log m)}(D_1)$ of length $|S| = \ell \in \Delta_{|P|}$ and $\ell' = \max\{\ell' \in \Delta_{|P|} \mid \ell' < \ell\}$ is the length preceding ℓ in $\Delta_{|P|}$ (if $\ell = 0$, which is the minimum value in $\Delta_{|P|}$ let $\ell' = 0$). Then, we define $p(S) = P[|P| - \ell' + 1..|P|]$ to be the suffix of S of length ℓ'. It is straightforward that $p(S) \in \mathsf{Suffixes}_1$. We define the *suffixes tree*, T_{suf}, as the tree induced by the strings and the parent string relation. Since each string has exactly one parent, except for the empty string, T_{suf} is well defined. Notice that a binary search that finds $S \in \mathsf{Suffixes}_1$ as an intermediate result, must consider all the strings on the path from the root to the node of S during its execution. Moreover, for each string S of length ℓ that is an intermediate result of the binary search, in the iteration after finding S, the binary search focuses on the range $[\ell, \ell + 2^{i(S)}]$ where $i(S) = \max\{i \mid \ell \bmod 2^{i-1} = 0\}$.

Compress T_{suf}. The algorithm traverses the tree from the root to the leaves. For any string S which has only one child, the algorithm shrinks the path from S to its first descendant S' that has at least two children, or S' is in $\mathsf{Heads}_{[\log m, 2\log m)}(D_1)$. The shrinking is done by setting $\mathsf{child}(S) = S'$ and removing all the strings between. Let $\mathsf{Suffixes}'_1$ be the set of remaining strings in the tree after the compression. The algorithm maintains a perfect hash table that maps any $S \in \mathsf{Suffixes}'_1$ into $i(S)$, and if S has only one child, S is associated also with the length $|\mathsf{child}(S)|$. In addition, S is associated with the index of the longest string from $\mathsf{Heads}_{[\log m, 2\log m)}(D_1)$ that is a suffix of S as well.

Query processing. For each text position which is a multiple of $\log m$, the algorithm finds the current longest suffix from $\mathsf{Heads}_{[\log m, 2\log m)}(D_1)$ as follows. The algorithm initializes a length $\ell = 0$ and an exponent $i = i_{max} + 1$. At each iteration, it must be that the suffix of the text of length ℓ is in $\mathsf{Suffixes}'_1$, let denote this string as T_ℓ. On iterations where T_ℓ has multiple children (which can be retrieved from the hash table), the algorithm decrements i by one, computes the fingerprint of the text suffix of length $\ell + 2^i$ and queries the hash table with this fingerprint. If this fingerprint is maintained in the table then the length ℓ is updated to $\ell + 2^i$. On iterations where T_ℓ has only one child, the algorithm computes the fingerprint of the text of length $|\mathsf{child}(T_\ell)|$ and queries the hash table with this fingerprint. If this fingerprint is maintained in the table then the length ℓ is updated. Otherwise, the search is terminated. When $i < 0$, the search is also terminated.

The following lemma states that the longest suffix of $T[1..q]$ from $\mathsf{Suffixes}'_1$ is found by the binary search. Since $\mathsf{Heads}_{[\log m, 2\log m)}(D_1) \subseteq \mathsf{Suffixes}'_1$, it is guaranteed that if S is the current longest suffix of the text from $\mathsf{Heads}_{[\log m, 2\log m)}(D_1)$ then the algorithm finds a string S' such that S is a suffix of S'. Hence, since the algorithm reports the longest suffix of S' that is a string from $\mathsf{Heads}_{[\log m, 2\log m)}(D_1)$, this string must be S.

▶ **Lemma 10.** *When $T[q]$ arrives, for q which is a multiple of $\log m$, the first phase of $\mathcal{A}_1$ finds the longest suffix of $T[1..q]$ that is a string in $\mathsf{Suffixes}'_1$.*

Complexities. For every text character, the first phase of the algorithm just updates the sliding window of fingerprints, in constant time. For a text position that is a multiple of $\log m$, the binary search is performed in $\mathcal{O}(i_{max}) = \mathcal{O}(\log m)$ time. The algorithm maintains a sliding window of $\mathcal{O}(d \log m)$ text fingerprints. In addition, it stores a hash table that maintains a constant number of words per each string in $\mathsf{Suffixes}'_1$. By simple analysis, we have that $|\mathsf{Suffixes}'_1| = \mathcal{O}(d \log m)$, so, the total space usage of the first phase of $\mathcal{A}_1$ is $\mathcal{O}(d \log m)$ words of space.

5 Long Patterns

In this section, we treat the patterns whose length is at least $8d \log m$. The algorithm distinguishes between patterns with a small period length and those with a large period length. For each pattern P_i with length $m_i > 8d \log m$, we define Q_i to be the prefix of P_i of length $|Q_i| = m_i - (2d+2)\log m$. In Section 5.1, we introduce the first phase of algorithm $\mathcal{A}_{2a}$ for $D_{2a} = \{P_i \in D \,|\, m_i > 8d \log m \text{ and } \rho_{Q_i} \leq d \log m\}$. In Section 5.2, we introduce the first phase of algorithm $\mathcal{A}_{2b}$ for $D_{2b} = \{P_i \in D \,|\, m_i > 8d \log m \text{ and } \rho_{Q_i} > d \log m\}$.

5.1 Long Patterns with Short Periods

In this section, we introduce the first phase of $\mathcal{A}_{2a}$ which considers patterns P_i of length at least $8d \log m$, with $\rho_{Q_i} < d \log m$. Intuitively, we utilize the periodicity of the patterns prefixes and search for the same periodicity in the text in a sufficiently long substring. At each position, if a string from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ ended in this position, there exists an occurrence of Q_i which ends at the $(2d+2)\log m$ preceding positions. In particular, since $\rho_{Q_i} < d \log m$, it must be that the text contains a long substring that has a period length ρ_{Q_i} that continues (at least) until the last $(2d+2)\log m$ positions. At any text position that is a multiple of $\log m$, by computing the fingerprint of the last $6d \log m$ characters, the algorithm determines a set of optional strings from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ whose suffix of length $6d \log m$ is a current suffix of the text. Notice that for each pair of such strings, one string must be the suffix of the other. This is because for each $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ there exists a Q_i, which is a prefix of S, and the suffix of S of length $6d \log m$ must contain at least two periods of Q_i, thus, this periodicity must continue until the suffix of S of length $6d \log m$. Therefore, to identify the longest text suffix that is a string in $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$, the algorithm finds the longest optional string that appears in the text due to the length of the periodic subtext. The main challenge is in maintaining the periodicity of the text in a manner that is easy to update (in constant time per character) and query. To tackle this challenge we utilize the combinatorial relationships between different Q_is.

For each $P_i \in D_{2a}$, we denote $\mathsf{prefix}(P_i) = P_i[1..2d \log m]$. Due to Lemma 6 we have that $\rho_{\mathsf{prefix}(P_i)} = \rho_{Q_i}$. So, if P_i occurs at position c of the text, then in particular, $\mathsf{prefix}(P_i)$ occurs at c, and by the periodicity of Q_i, $\mathsf{prefix}(P_i)$ occurs also at any position $c_k = c + k \cdot \rho_{\mathsf{prefix}(P_i)}$ for any positive integer k such that $c_k + |\mathsf{prefix}(P_i)| \leq c + |Q_i|$. To identify occurrences of prefixes of P_i from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ the algorithm searches for a sufficiently long arithmetic progression of $\mathsf{prefix}(P_i)$ occurrences in the text. Since all the $\mathsf{prefix}(\cdot)$ strings that the algorithm searches for are of the same length $2d \log m$, the algorithm is able to search them with a constant time per character by using the sliding window of the last $8d \log m$ text fingerprints and a perfect hash table of all the prefixes of patterns in D_{2a}.

Let $\mathsf{Prefixes}_{2a} = \{\mathsf{prefix}(P_i) \,|\, P_i \in D_{2a}\}$ be the set of all prefixes of length $2d \log m$, and let $\mathsf{p}_i, \mathsf{p}_j \in \mathsf{Prefixes}_{2a}$ be two strings from this set. We distinguish between two cases: in the first case, the prefixes p_i and p_j *agree* with each other, which means that between two close occurrences of one of them, there must exist an occurrence of the other. In the second case, p_i and p_j *disagree*, and whenever there exist two close occurrences of one of them, there is no occurrence of the other.

Formally, for each $\mathsf{p} \in \mathsf{Prefixes}_{2a}$, let $\alpha(\mathsf{p}) = \min\{\sigma^s(P[1..\rho_{\mathsf{p}}]) \,|\, s \in \{0, 1, \ldots, \rho_{\mathsf{p}} - 1\}\}$ be the *identify period* (also known as the Lyndon representation) of p, where $\sigma(\cdot)$ is the cyclic shift function (see Section 2), and the minimum is taken according to lexicographic order. The following lemma formalizes the possible relations between prefixes of patterns from D_{2a}.

▶ **Lemma 11.** *Let* $\mathsf{p}_i, \mathsf{p}_j \in \mathsf{Prefixes}_{2a}$*, and let* S *be the string of length* $2d\log m + \rho_{\mathsf{p}_i}$ *such that the prefix and suffix of* S *are both equal* p_i*. Then,* S *contains an occurrence of* p_j *if and only if* $\alpha(\mathsf{p}_i) = \alpha(\mathsf{p}_j)$*.*

Detect periodic substrings. In the preprocessing phase, we cluster the strings of $\mathsf{Prefixes}_{2a}$ according to their identify period. Let $\mathsf{Prefixes}^u_{2a} = \{\mathsf{p}_i \mid \mathsf{p}_i \in \mathsf{Prefixes}_{2a} \text{ and } \alpha(\mathsf{p}_i) = u\}$ be the cluster of prefixes with identify period u. We associate with each prefix $\mathsf{p}_i \in \mathsf{Prefixes}^u_{2a}$ a shift value $0 \leq s(\mathsf{p}_i) < \rho_{\mathsf{p}_i}$, such that the prefix of p_i of length $|u|$ is $\sigma^{s(\mathsf{p}_i)}(u)$. Let $\Psi = \{s(\mathsf{p}_i) \mid \mathsf{p}_i \in \mathsf{Prefixes}^u_{2a}\}$ be the set of all shift values of strings in $\mathsf{Prefixes}^u_{2a}$ and let $s_1 < s_2 < \cdots < s_h$ be the elements of Ψ ordered by increasing value. With each string $\mathsf{p}_i \in \mathsf{Prefixes}^u_{2a}$ such that $s(\mathsf{p}_i) = s_j$, the algorithm maintains the length $\delta(\mathsf{p}_i) = s_j - s_{j-1}$ (for $j = 0$ we have $\delta(\mathsf{p}_i) = |u| - s_h + s_1$), and the string $\mathsf{p}' \in \mathsf{Prefixes}^u_{2a}$ with shift value $s(\mathsf{p}') = s_{i-1}$ (for $i = 0$, $s(\mathsf{p}') = s_h$).

For each cluster $\mathsf{Prefixes}^u_{2a}$, we say that a string X is *u-periodic* if and only if $|X| \geq 2d\log m$, the prefix and suffix of length $2d\log m$ of X are strings in $\mathsf{Prefixes}^u_{2a}$, and the principal period length of X is $|u|$ (i.e., $\rho_X = |u|$).

In order to detect periodic substrings of the text, the algorithm maintains a sliding window of the last $8d\log m$ positions, where at each text position q, which is the end of some string $\mathsf{p}_i \in \mathsf{Prefixes}_{2a}$, the algorithm maintains (1) the id number of p_i and (2) the length of the *maximal* suffix of $T[1..q]$ that is $\alpha(\mathsf{p}_i)$-periodic string. Whenever a new character arrives, the algorithm computes the fingerprint of the current suffix of length $2d\log m$ and checks if it is a fingerprint of some $\mathsf{p}_i \in \mathsf{Prefixes}_{2a}$. If there exists such p_i, let $u = \alpha(\mathsf{p}_i)$ and let p' be the string preceded p_i in the cluster $\mathsf{Prefixes}^u_{2a}$ according to the cyclic shift. To compute the total length of the maximal u-periodic suffix of $T[1..q]$, the algorithm checks whether an occurrence of p' ended at $T[q - \delta(\mathsf{p}_i)]$. If such an occurrence exists, the length of the current maximal suffix is the sum of the length of the u-periodic suffix of $T[1..q - \delta(\mathsf{p}_i)]$ and $\delta(\mathsf{p}_i)$. If p' does not end at $T[q - \delta(\mathsf{p}_i)]$, then the length of the u-periodic sequence up to position q is exactly $2d\log m$.

Heads detection. For each $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$, we denote by $\mathsf{suffix}(S)$ the suffix of S of length $6d\log m$. The algorithm stores the fingerprints of all the strings in $\mathsf{Suffixes}_{2a} = \{\mathsf{suffix}(S) \mid S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})\}$ in a perfect hash table (notice that $\mathsf{Suffixes}_1$ and $\mathsf{Suffixes}_{2a}$ are two sets of heads suffixes, but their definitions are quite different). For each $\mathsf{s} \in \mathsf{Suffixes}_{2a}$, there exists a string $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ such that $\mathsf{s} = \mathsf{suffix}(S)$. By definition, we have that S is a prefix of some $P_i \in D_{2a}$ and Q_i is a prefix of S. Hence, due to the periodicity of Q_i, it must be that the prefix of s of length $3d\log m$ has a principal period length ρ_{Q_i}. Let v be the prefix of s of length $2d\log m$, and let $u = \alpha(v)$. By the periodicity, it must be that all the strings in $\mathsf{Prefixes}^u_{2a}$ appears in s. We denote by $\delta'(\mathsf{s})$ the distance between the last occurrence of some string from $\mathsf{Prefixes}^u_{2a}$ and the end of s. Formally, $\delta'(\mathsf{s}) = |\mathsf{s}| - \max\{j \mid \mathsf{s}[j - 2d\log m + 1..j] \in \mathsf{Prefixes}^u_{2a}\}$. Since there exists a string from $\mathsf{Prefixes}^u_{2a}$ that appears in s, it is obvious that $\delta'(\mathsf{s})$ is less than $6d\log m$. The following lemma proves that in order to detect occurrences of strings from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$, it suffices to detect $\mathsf{suffixes}$ and periodic substrings of the text.

▶ **Lemma 12.** *Let* $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ *we have that* $S = T[q - |S| + 1..q]$ *if and only if* $T[q - 6d\log m + 1..q] = \mathsf{suffix}(S)$ *and the length of maximal periodic suffix of* $T[1..q - \delta'(\mathsf{suffix}(S))]$ *is at least* $|S| - \delta'(\mathsf{suffix}(S))$*.*

Whenever the algorithm finds an occurrence of $\mathsf{s} \in \mathsf{Suffixes}_{2a}$ ended at position q, it might be the end of all the strings in $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2a})$ which their suffix is s. Let $\mathsf{Optional}_{\mathsf{s}} = \{S \mid S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2a}) \text{ and } \mathsf{suffix}(S) = \mathsf{s}\}$ be the set of heads which are

optional occurrences when s occurs. It is straightforward that $|\mathsf{Optional}_{\mathsf{s}}| \leq m$ due to the periodicity of sufficiently long prefixes of D_{2a}. The algorithm stores all the indices of these strings indexed according to their length in a balanced binary tree. Due to Lemma 12, in order to detect the longest string from $\mathsf{Optional}_{\mathsf{s}}$ that is a suffix of the text, the algorithm has to find the longest string whose length is at most the sum of the length of the longest periodic suffix of $T[1..q - \delta'(\mathsf{s})]$ and $\delta'(\mathsf{s})$.

At each text position q that is a multiple of $\log m$, the algorithm computes the fingerprint of the last $6d \log m$ characters using the sliding window of text fingerprints. If this suffix is some $\mathsf{s} \in \mathsf{Suffixes}_{2a}$, the algorithm uses the sliding window of periodic suffixes to retrieve the length ℓ of the longest periodic suffix of $T[1..q - \delta'(\mathsf{s})]$. Then, the algorithm finds the predecessor of $\ell + \delta'(\mathsf{s})$ in the binary tree and reports the string associated with this length.

Complexities. The algorithm maintains the fingerprints of strings from $\mathsf{Prefixes}_{2a}$ and $\mathsf{Suffixes}_{2a}$, and since each of them is of size $\mathcal{O}(d \log m)$, we have $\mathcal{O}(d \log m)$ fingerprints and each of them is maintained within constant space. Moreover, the algorithm maintains sliding windows of fingerprints and periodic sequences information, each of them of length $\mathcal{O}(d \log m)$ and at each position the sliding windows use $\mathcal{O}(1)$ words of space. Therefore the first phase of $\mathcal{A}_{2a}$ uses $\mathcal{O}(d \log m)$ words of space. The first phase takes $\mathcal{O}(1)$ time per character for computing the fingerprint of the current $2d \log m$ suffix and an additional $\mathcal{O}(\log m)$ time per each text position that is a multiple of $\log m$, for the predecessor query.

5.2 Long Patterns with Long Periods

In this section, we introduce the first phase of $\mathcal{A}_{2b}$, which considers patterns P_i of length at least $8d \log m$, with $\rho_{Q_i} > d \log m$. An overview of the algorithm is as follows. First, the algorithm finds all the occurrences of strings Q_i, with a delay of at most $d \log m$ characters. Each occurrence of Q_i is a possible occurrence of P_i, so the algorithm computes the position q that is a multiple of $\log m$ and is in the range of $[\log m, 2 \log m)$ positions preceding the end of the possible occurrence of P_i. Then, the algorithm computes the expected text fingerprint in this position if this occurrence of Q_i is indeed an occurrence of P_i. The expected text fingerprint is maintained in a designated data structure. When $T[q]$ arrives, the algorithm checks if the text fingerprint matches the expected fingerprint, and if so, it reports the appropriate prefix of P_i from $\mathsf{Heads}_{[\log m, 2 \log m)}(D_{2b})$.

Finding Q_i with a delay. The algorithm finds all the occurrences of any Q_i in the text. The algorithm uses a similar technique to algorithm $\mathcal{A}_{2b}$ of Clifford et al. [12], which creates $\mathcal{O}(\log m)$ *levels* for each pattern in the dictionary. Each level maintains occurrences of a prefix of the pattern of length which is a power of two. The algorithm finds for each pattern occurrences of the shortest prefix of the pattern whose principal period length is greater than $d \log m$, by applying algorithms $\mathcal{A}_1$ and $\mathcal{A}_{2a}$. Thus, all the longer prefixes have at least $d \log m$ characters between any two occurrences, and therefore by a round-robin fashion their levels are treated in $\mathcal{O}(1)$ time per character. The algorithm has the guarantee that each occurrence of any Q_i in the text is found by the algorithm with a delay of at most $d \log m$ text characters. The complexities of this part are according to Theorem 2 or Lemma 4 and the complete details will appear in the final version of this paper.

Extend Q_i to prefix from $\mathsf{Heads}_{[\log m, 2 \log m)}(D_{2b})$. After finding each Q_i with a delay of at most $d \log m$ text characters, the goal is to find at each position q that is a multiple of $\log m$, the longest suffix of the text that is a string from $\mathsf{Heads}_{[\log m, 2 \log m)}(D_{2b})$. Let c

be a text index where an occurrence of Q_i begins. Thus, c is a possible occurrence of P_i,[4] and the algorithm must validate it. The algorithm computes q_c that is the text index in the range $q_c \in (c + m_i - 1 - 2\log m, c + m_i - 1 - \log m]$ which is a multiple of $\log m$. Since the length of the range is $\log m$, there is exactly one such position. Let S be the prefix of P_i of length $\ell = q_c - c + 1$, by definition $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2b})$. The algorithm computes the expected fingerprint of the text in position q_c from $\phi(T[1..c-1])$ and $\phi(S)$. This fingerprint, together with the index of S is maintained in a data structure associated with position q_c.

The algorithm maintains a sliding window of the subsequent $(2d+1)$ positions that are multiples of $\log m$. Since $m_i - 2\log m < |S| \leq m_i - \log m$ and $|Q_i| = m_i - (2d+2)\log m$, the distance (in text characters) between the end of the Q_i's occurrence and q_c is in the range $(2d\log m, (2d+1)\log m]$. Hence, q_c is in the sliding window. For each position q in the sliding window, the algorithm maintains an AVL tree, which maintains expected text fingerprints. Each expected fingerprint is maintained with an id number of the corresponding string from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2b})$. When a new occurrence of some Q_i is found, the algorithm inserts into the AVL tree the expected fingerprint it computes, associated with the id number of the corresponding $S \in \mathsf{Heads}_{[\log m, 2\log m)}(D_{2b})$. If this fingerprint already exists in the tree, the algorithm updates the corresponding string to be the longest between the existing string and the new string.

Since $\rho_{Q_i} > d\log m$, at any sequence of $2d\log m$ characters, there exist at most 2 occurrences of Q_i. Thus, the total number of values inserted into the trees in a sequence of $2d\log m$ characters is at most $\mathcal{O}(d)$. In addition, for each position q in the sliding window the number of elements in the AVL tree is at most d, hence, the insertion takes $\mathcal{O}(\log d)$ time. So, the total time of insertions in a sequence of $2d\log m$ characters is $\mathcal{O}(d\log d)$. Since $D_{2b} \neq \emptyset$ we have some strings of length at least $8d\log m$, and therefore $m > d$ and thus $\mathcal{O}(d\log d) = \mathcal{O}(d\log m)$. Using the round-robin fashion, all the insertions to the trees are de-amortized to $\mathcal{O}(1)$ time per character. The round-robin fashion may create a delay of at most $d\log m$ text characters. Recall that any occurrence of Q_i is found with a delay of at most $d\log m$ characters and that the distance between the end of the Q_i's occurrence and q_c is at least $2d\log m$. Thus, there exists at least $d\log m$ characters between the recognizing of Q_i's occurrence and q_c. Hence, when $T[q]$ arrives, its expected fingerprints tree contains all the expected fingerprints corresponding to occurrences of all Q_is which their q_c is q.

When $T[q]$ arrives, for q that is a multiple of $\log m$, the algorithm searches for the current text fingerprint $\phi(T[1..q])$ in the AVL tree of position q. The time required for this search is $\mathcal{O}(\log d) = \mathcal{O}(\log m)$. In the following lemma, we prove that the algorithm indeed finds the longest suffix of $T[1..q]$ from $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2b})$ for every q that is a multiple of $\log m$.

▶ **Lemma 13.** *When $T[q]$ arrives, for q which is a multiple of $\log m$, the first phase of $\mathcal{A}_{2b}$ finds the longest suffix of $T[1..q]$ that is a string in $\mathsf{Heads}_{[\log m, 2\log m)}(D_{2b})$.*

Complexities. Since all the usage of round-robin fashion described above is on ranges of size $\mathcal{O}(d\log m)$, the de-amortization uses $\mathcal{O}(d\log m)$ words of space. Hence, summing all the parts, the time and space of the first phase of $\mathcal{A}_{2b}$ are as stated in Theorem 2 or Lemma 4 with $\mathcal{O}(\log m)$ additional time for any position which is a multiple of $\log m$.

[4] For the sake of simplicity, we assume that for any two different patterns $P_i, P_j \in D_{2b}$, we have $Q_i \neq Q_j$. Otherwise, we treat each occurrence of Q_i multiple times, each time as the prefix of another $P_i \in D_{2b}$.

References

1. Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.
2. Mansoor Alicherry, Muthusrinivasan Muthuprasanna, and Vijay Kumar. High speed pattern matching for network IDS/IPS. In *Proceedings of the 14th IEEE International Conference on Network Protocols, ICNP*, pages 187–196, 2006. doi:10.1109/ICNP.2006.320212.
3. Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999. doi:10.1006/jcss.1997.1545.
4. Amihood Amir, Tsvi Kopelowitz, Avivit Levy, Seth Pettie, Ely Porat, and B. Riva Shalom. Mind the gap: Essentially optimal algorithms for online dictionary matching with one gap. In *27th International Symposium on Algorithms and Computation, ISAAC 2016*, pages 12:1–12:12, 2016. doi:10.4230/LIPIcs.ISAAC.2016.12.
5. Amihood Amir, Avivit Levy, Ely Porat, and B. Riva Shalom. Dictionary matching with one gap. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM*, pages 11–20, 2014.
6. Amihood Amir, Avivit Levy, Ely Porat, and B. Riva Shalom. Dictionary matching with a few gaps. *Theor. Comput. Sci.*, 589:34–46, 2015.
7. Tanver Athar, carl Barton, Widmer bland, Jia Gao, Costas S. Illopoulos, Chang Liu, and Solon P. Pissis. Fast circular dictionary-matching algorithm. *Mathematical Structures in Computer Science*, pages 1–14, 2015.
8. Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009*, pages 785–794, 2009. URL: http://dl.acm.org/citation.cfm?id=1496770.1496856.
9. Anat Bremler-Barr, David Hay, and Yaron Koral. Compactdfa: Scalable pattern matching using longest prefix match solutions. *IEEE/ACM Trans. Netw.*, 22(2):415–428, 2014. doi:10.1109/TNET.2013.2253119.
10. Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Transactions on Algorithms*, 10(4):22:1–22:12, 2014. doi:10.1145/2635814.
11. Dany Breslauer, Roberto Grossi, and Filippo Mignosi. Simple real-time constant-space string matching. *Theor. Comput. Sci.*, 483:2–9, 2013. doi:10.1016/j.tcs.2012.11.040.
12. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. Dictionary matching in a stream. In *Proceedings of Annual European Symposium on Algorithms, ESA*, pages 361–372, 2015.
13. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. The k-mismatch problem revisited. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 2039–2052, 2016. doi:10.1137/1.9781611974331.ch142.
14. Funda Ergün, Hossein Jowhari, and Mert Saglam. Periodicity in streams. In *Proceedings of Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 13th International Workshop, APPROX 2010, and 14th International Workshop, RANDOM 2010*, pages 545–559, 2010. doi:10.1007/978-3-642-15369-3_41.
15. Yu Fang, Randy H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *Proceedings of 12th IEEE International Conference on Network Protocols (ICNP 2004)*, pages 174–183, 2004. doi:10.1109/ICNP.2004.1348108.
16. Guy Feigenblat, Ely Porat, and Ariel Shiftan. An improved query time for succinct dynamic dictionary matching. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM*, pages 120–129, 2014.

17 Guy Feigenblat, Ely Porat, and Ariel Shiftan. Linear time succinct indexable dictionary construction with applications. In *Data Compression Conference , DCC*, pages 13–23, 2016.

18 Johannes Fischer, Travis Gagie, Pawel Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Proceedings of Algorithms - ESA 2015 - 23rd Annual European Symposium*, pages 533–544, 2015. doi:10.1007/978-3-662-48350-3_45.

19 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

20 Arnab Ganguly, Wing-Kai Hon, Kunihiko Sadakane, Rahul Shah, Sharma V. Thankachan, and Yilin Yang. Space-efficient dictionaries for parameterized and order-preserving pattern matching. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM*, pages 2:1–2:12, 2016.

21 Arnab Ganguly, Wing-Kai Hon, and Rahul Shah. A framework for dynamic parameterized dictionary matching. In *15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT*, pages 10:1–10:14, 2016.

22 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Streaming Pattern Matching with d Wildcards. In *24th Annual European Symposium on Algorithms (ESA)*, pages 44:1–44:16, 2016.

23 Torben Hagerup. Sorting and searching on the word RAM. In *STACS 98, 15th Annual Symposium on Theoretical Aspects of Computer Science, Paris, France, February 25-27, 1998, Proceedings*, pages 366–398, 1998. doi:10.1007/BFb0028575.

24 Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001. doi:10.1006/jagm.2001.1171.

25 Monika Rauch Henzinger, Prabhakar Raghavan, and Sridar Rajagopalan. *External Memory Algorithms*, chapter Computing on data streams, pages 107–118. American Mathematical Society, 1999.

26 Cheng-Liang Hsieh, Lucas Vespa, and Ning Weng. A high-throughput DPI engine on GPU via algorithm/implementation co-optimization. *J. Parallel Distrib. Comput.*, 88:46–56, 2016. doi:10.1016/j.jpdc.2015.11.001.

27 Markus Jalsenius, Benny Porat, and Benjamin Sach. Parameterized matching in the streaming model. In *proceedings of 30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013*, pages 400–411, 2013. doi:10.4230/LIPIcs.STACS.2013.400.

28 Daniel M. Kane, Jelani Nelson, Ely Porat, and David P. Woodruff. Fast moment estimation in data streams in optimal space. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011*, pages 745–754, 2011. doi:10.1145/1993636.1993735.

29 Junghak Kim and Song-in Choi. High speed pattern matching for deep packet inspection. In *Communications and Information Technology, 2009. ISCIT 2009. 9th International Symposium on*, pages 1310–1315. IEEE, 2009.

30 Tsvi Kopelowitz, Ely Porat, and Yaron Rozen. Succinct Online Dictionary Matching with Improved Worst-Case Guarantees. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CPM.2016.6.

31 Lap-Kei Lee, Moshe Lewenstein, and Qin Zhang. Parikh matching in the streaming model. In *Proceedings of String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012*, pages 336–341, 2012. doi:10.1007/978-3-642-34109-0_35.

32 S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005. doi:10.1561/0400000002.

33 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *Proceedings of 50th Annual IEEE Symposium on Foundations of Computer*

Science, FOCS 2009, October 25-27, 2009, pages 315–323, 2009. doi:10.1109/FOCS.2009.11.

34 Yaron Rozen. On the online dictionary matching problem. Master's thesis, Bar-Ilan University, Ramat Gan, Israel, September 2016. Under the supervision of Prof. Ely Porat.

35 Milan Ruzic. Constructing efficient dictionaries in close to sorting time. In *Proceedings of Automata, Languages and Programming, 35th International Colloquium, ICALP 2008*, pages 84–95, 2008. doi:10.1007/978-3-540-70575-8_8.

36 Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings IEEE INFOCOM 2004, The 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, 2004. URL: http://www.ieee-infocom.org/2004/Papers/54_5.PDF.

37 Antonino Tumeo, Oreste Villa, and Daniel G. Chavarría-Miranda. Aho-corasick string matching on shared and distributed-memory parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 23(3):436–443, 2012. doi:10.1109/TPDS.2011.181.

38 Antonino Tumeo, Oreste Villa, and Donatella Sciuto. Efficient pattern matching on gpus for intrusion detection systems. In *Proceedings of the 7th Conference on Computing Frontiers, 2010*, pages 87–88, 2010. doi:10.1145/1787275.1787296.

39 Lucas Vespa and Ning Weng. GPEP: graphics processing enhanced pattern-matching for high-performance deep packet inspection. In *2011 IEEE International Conference on Internet of Things (iThings) & 4th IEEE International Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 74–81, 2011. doi:10.1109/iThings/CPSCom.2011.36.

40 Lucas Vespa and Ning Weng. Swm: Simplified wu-manber for gpu-based deep packet inspection. In *Proceedings of the International Conference on Security and Management (SAM)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

41 Yaron Weinsberg, Shimrit Tzur-David, Danny Dolev, and Tal Anker. High performance string matching algorithm for a network intrusion prevention system (nips). In *High Performance Switching and Routing, 2006 Workshop on*, pages 7–pp. IEEE, 2006.

42 SangKyun Yun. An efficient tcam-based implementation of multipattern matching using covered state encoding. *IEEE Transactions on Computers*, 61(2):213–221, 2012.

43 Xinyan Zha and Sartaj Sahni. Gpu-to-gpu and host-to-host multipattern string matching on a GPU. *IEEE Trans. Computers*, 62(6):1156–1169, 2013. doi:10.1109/TC.2012.61.

Improved Bounds for 3SUM, k-SUM, and Linear Degeneracy*†

Omer Gold[1] **and Micha Sharir**[2]

1 **School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
omergold@post.tau.ac.il
2 **School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
michas@post.tau.ac.il

Abstract

Given a set of n real numbers, the 3SUM problem is to decide whether there are three of them that sum to zero. Until a recent breakthrough by Grønlund and Pettie [FOCS'14], a simple $\Theta(n^2)$-time deterministic algorithm for this problem was conjectured to be optimal. Over the years many algorithmic problems have been shown to be reducible from the 3SUM problem or its variants, including the more generalized forms of the problem, such as k-SUM and k-variate linear degeneracy testing (k-LDT). The conjectured hardness of these problems have become extremely popular for basing conditional lower bounds for numerous algorithmic problems in P.

In this paper, we show that the randomized 4-linear decision tree complexity[1] of 3SUM is $O(n^{3/2})$, and that the randomized $(2k-2)$-linear decision tree complexity of k-SUM and k-LDT is $O(n^{k/2})$, for any odd $k \geq 3$. These bounds improve (albeit being randomized) the corresponding $O(n^{3/2}\sqrt{\log n})$ and $O(n^{k/2}\sqrt{\log n})$ bounds obtained by Grønlund and Pettie. Our technique includes a specialized randomized variant of the fractional cascading data structure. Additionally, we give another deterministic algorithm for 3SUM that runs in $O(n^2 \log\log n/\log n)$ time. The latter bound matches a recent independent bound by Freund [Algorithmica 2017], but our algorithm is somewhat simpler, due to a better use of the word-RAM model.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases 3SUM, k-SUM, Linear Degeneracy, Linear Decision Trees, Fractional Cascading

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.42

1 Introduction

The general 3SUM problem is formally defined as

3SUM: Given a finite set $A \subset \mathbb{R}$, determine whether there exist $a, b, c \in A$ such that $a + b + c = 0$.

An equivalent variant is that the input consists of three finite sets A, B, $C \subset \mathbb{R}$ of the same size, and the goal is to determine whether there are elements $a \in A$, $b \in B$, $c \in C$ such

* For the full version of this paper see [21]. Work on this paper has been supported by Grant 892/13 from the Israel Science Foundation, by Grant 2012/229 from the U.S.-Israeli Binational Science Foundation, by the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11), by the Blavatnik Research Fund in Computer Science at Tel Aviv University, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

† A full version of the paper is available at http://arxiv.org/abs/1512.05279.

1 An r-linear decision tree is one in which each branching is based on a sign test of a linear expression with at most r terms. The complexity of the tree is its depth.

that $a+b+c=0$. When the sets A, B, C are not of the same size, the problem is named unbalanced 3SUM.

The 3SUM problem and its variants are among the most fundamental problems in algorithm design. Although the 3SUM problem itself does not seem to have many compelling practical implications, it has been of wide interest due to numerous problems that can be reduced from it. The notion of 3SUM-Hardness is often used to describe such problems, namely, problems that are at least as hard as 3SUM. Thus, lower bounds on 3SUM imply lower bounds on dozens of other problems. Among them are fundamental problems in computational geometry [20, 3, 7, 30], dynamic graph algorithms [29, 1, 27], triangle enumeration [2, 27], and pattern matching [31, 5, 9, 27, 6].

In the last decades, starting with a study of Gajentaan and Overmars [20], it was conjectured that any algorithm for 3SUM requires $\Omega(n^2)$ time. However, a recent breakthrough by Grønlund and Pettie [22] showed that 3SUM can be solved in subquadratic time. Specifically, they gave a deterministic algorithm that runs in $O(n^2(\log\log n/\log n)^{2/3})$ time, and a randomized algorithm that runs in $O(n^2(\log\log n)^2/\log n)$ expected time and with high probability. Furthermore, they showed that there is a 4-linear decision tree for 3SUM with depth $O(n^{3/2}\sqrt{\log n})$ (i.e., the depth bounds the number of branching operations, each one is based on sign test of a linear expression with at most 4 terms). These results raised serious doubts on the optimality of many algorithms for 3SUM-Hard problems. For example, the following problems are known to be 3SUM-Hard. (1) Given an n-point set in $\mathbb{R}^2$, determine whether it contains three collinear points (Gajentaan and Overmars [20]). (2) Given n triangles in $\mathbb{R}^2$, determine whether their union contains a hole, or compute the area of their union [20]. (3) Given two n-point sets $X, Y \subset \mathbb{R}$, each of size n, determine whether all elements in $X+Y=\{x+y \mid x\in X,\ y\in Y\}$ are distinct (Barequet and Har-Peled [7]). (4) Given two n-edge convex polygons, determine whether one can be placed inside the other via translation and rotation [7].

Problems 1 and 2 are solvable in $O(n^2)$ time (see [20]). Problems 3 and 4 are solvable in $O(n^2\log n)$ time (see [7]). In face of the new 3SUM result of Grønlund and Pettie [22], it is natural to ask whether these bounds are optimal. However, no better bounds are currently known (in spite of the improvement in [22]). Problem 3 (or its stronger variant of sorting $X+Y$) has special importance, as it is used for basing the conditional lower bounds for the problems in [7] and in [23]; these problems are therefore also classified as "(Sorting $X+Y$)-Hard". It is a prominent long-standing open problem whether Problem 3 can be solved in $o(n^2\log n)$ time (see [14]).

In view of the results in [22], the 3SUM conjecture has been replaced by a relaxed, modern variant, asserting that 3SUM cannot be solved in *strongly subquadratic time* (even in expectation), i.e., in $O(n^{2-\epsilon})$ time, for any $\epsilon>0$. This conjecture is widely accepted and believed by the computer science community, and so are its implications for deriving lower bounds for other problems. Abboud and Vassilevska-Williams [2] argue, based on the collective computer science community efforts, that lower bounds that are based on the relaxed 3SUM conjecture should be at least as believable as any other known conditional lower bounds for a problem in P.

The 3SUM problem was also extensively studied in its generalized forms, k-SUM and k-variate linear degeneracy testing (k-LDT), formally defined as

***k*-LDT and *k*-SUM:** Given a k-variate linear function $\phi(x_1,\ldots,x_k)=\alpha_0+\sum_{i=1}^{k}\alpha_i x_i$, where $\alpha_0,\ldots,\alpha_k\in\mathbb{R}$, and a finite set $A\subset\mathbb{R}$, determine whether there exists $(x_1,\ldots,x_k)\in A^k$ such that $\phi(x_1,\ldots,x_k)=0$. When ϕ is $\sum_{i=1}^{k}x_i$ the problem is called k-SUM.

There are simple algorithms that solve k-LDT in time $O(n^{(k+1)/2})$ when k is odd, or $O(n^{k/2}\log n)$ when k is even; see [4]. These algorithms are based on straightforward reduc-

tions to a 2SUM problem or to an unbalanced 3SUM problem, depending on whether k is even or odd, respectively. These are currently the best known upper bounds for the running time of solving k-LDT. Erickson [16] showed that, for an even k, there is a k-linear decision tree with depth $O(n^{k/2})$, removing an $O(\log n)$ factor when comparing to the uniform model. The above bounds match with the seminal lower bound results of Erickson [16], and of Ailon and Chazelle [4], who showed that any k-linear decision tree for solving k-LDT must have depth $\Omega(n^{k/2})$ when k is even and $\Omega(n^{(k+1)/2})$ when k is odd. In particular, any 3-linear decision tree for 3SUM has depth $\Omega(n^2)$. Grønlund and Pettie [22] showed that using only one more variable per comparison leads to a dramatic improvement in the depth of the tree, which significantly beats the above lower bounds. Specifically, as will be reviewed below, they showed that there is a 4-linear decision tree for 3SUM with depth $O(n^{3/2}\sqrt{\log n})$, and by the reduction from k-LDT to unbalanced 3SUM, they concluded that there is a $(2k-2)$-linear decision tree for k-LDT with depth $O(n^{k/2}\sqrt{\log n})$, for any odd $k \geq 3$. Cardinal, Iacono, and Ooms [10] showed that if we allow arbitrarily many variables in a comparison (polynomial in n), then the linear decision tree complexity of k-SUM and k-LDT is $O(n^3 \log^3 n)$. This bound was recently improved by Ezra and Sharir [17] to $O(n^2 \log^2 n)$. A very recent breakthrough by Kane, Lovett, and Moran [26] significantly improves these results, not only by showing an $O(n \log^2 n)$ bound, but also by using only $2k$ variables in a comparison, namely, a $2k$-linear decision tree (see below for further details).

Apart from the many lower bounds obtained from the conjectured hardness of 3SUM and its variants, in recent years, many lower bounds were obtained also from two other plausible conjectures. The first is that computing the $(\min, +)$–product of two $n \times n$ matrices takes $\Omega(n^{3-o(1)})$ time (aka APSP-Hardness); see for examples [32, 2, 1]. The second is that CNF-SAT takes $\Omega(2^{(1-o(1))n})$ time. The latter is often referred to as the *Strong Exponential Time Hypothesis* (SETH) [24, 25]. A natural question is whether any of these conjectures (3SUM, SETH, APSP) are in fact equivalent, or whether they all derive from a basic unifying hypothesis. At the current state of knowledge, there is no strong relationship between any pair of these problems, so it may be possible that any one of them could be true or false, independently of the status of the others. A recent breakthrough by Carmosino, Gao, and Impagliazzo [11] provides evidence that such a relationship is *unlikely*, based on a nondeterministic variant of SETH; see [11] for details.

1.1 Our Results and Related Work

Before presenting our results, we recall the definition of the *randomized* r-linear decision tree complexity, for a particular target function f. Consider a probability distribution $\mathcal{P}$ over a set $\mathcal{T}$ of (deterministic) r-linear decision trees that compute f. For a particular input x, let $c(\mathcal{P}, x)$ be the expected number of branching operations a tree chosen from $\mathcal{T}$ will make on input x. Then, the randomized r-linear decision tree complexity of f is

$$\min_{\mathcal{P}} \max_{x} c(\mathcal{P}, x).$$

The following theorems capture our main results.

▶ **Theorem 1.** *The randomized 4-linear decision tree complexity of* 3SUM *is* $O(n^{3/2})$.

▶ **Theorem 2.** *The randomized* $(2k-2)$*-linear decision tree complexity of* k-SUM *and of* k-LDT *is* $O(n^{k/2})$*, for any odd* $k \geq 3$.

We show these results by giving a randomized algorithm that constructs a $(2k-2)$-linear decision tree whose expected depth is $O(n^{k/2})$. Theorems 1 and 2 improve (albeit in a

randomized setting) the respective $O(n^{3/2}\sqrt{\log n})$-depth and $O(n^{k/2}\sqrt{\log n})$-depth decision trees given by Grønlund and Pettie [22]. The aforementioned recent breakthrough by Kane, Lovett, and Moran [26] gives a 6-linear decision tree for 3SUM with depth $O(n\log^2 n)$, and in general, a $2k$-linear decision tree for k-SUM, and a $(2k+2)$-linear decision tree for k-LDT, both with depth $O(kn\log^2 n)$. These bounds nearly match the standard $\Omega(n\log n)$ lower bound. Viewing our (and Grønlund and Pettie's) k-SUM results for $(2k-2)$-linear decision tree, with respect to Erickson's $\Omega(n^{\lceil k/2\rceil})$ k-linear decision tree lower bound, and the $2k$-linear decision upper bound by Kane, Lovett, and Moran [26], shows that even adding only 1 or 2 terms for each linear comparison, can significantly improve the depth of the tree.

Our technique includes some new insights into the 3SUM problem, and uses a specialized data structure, based on an unusual randomized variant of fractional cascading in a grid.

Additionally, in the full version of this paper [21], we give an actual deterministic algorithm for 3SUM that runs in $O(n^2\log\log n/\log n)$ time.[2] The latter improves the $O(n^2(\log\log n/\log n)^{2/3})$-time bound of Grønlund and Pettie [22], and matches the bound given by a recent independent work of Freund [19]. Both algorithms, Freund's [19] and ours, have common high-level ideas, but ours makes a better use of the word-RAM model, and is hence somewhat simpler.[3]

Recently, Lincoln, Vassilevska-Williams, Wang, and Williams [28] showed a reduction result in which they apply our 3SUM algorithm (based on an initial version of this paper [21]) as a black-box, leading to a 3SUM algorithm that uses only $O\left(\sqrt{n\log n/\log\log n}\right)$ space, while preserving the time bound of our algorithm.

2 Methods and Lemmas

We give an overview of the techniques we use. Some of them were also used for some of the results mentioned above. This includes Fredman's prominent work from 1976 [18]. For our result, we will develop a special randomized variant of *fractional cascading* (Chazelle and Guibas [12, 13]). In this section we also briefly review the standard fractional cascading method, to set the infrastructure upon which we will later develop our specialized variant.

Throughout the paper we refer to the trivial (albeit ingenious) observation that $a+b<a'+b'$ iff $a-a'<b'-b$ as *Fredman's trick*. We denote by $[N]$ the first $\lceil N\rceil$ natural numbers succeeding zero $\{1,\ldots,\lceil N\rceil\}$, where N may or may not be an integer.

Fredman showed that, given n numbers whose sorted order is one of $\Pi\le n!$ realizable permutations, they can be sorted using a linear number of comparisons when Π is sufficiently small. More generally, we have:

▶ **Lemma 3** (Fredman 1976 [18]). *A list L of n numbers, whose sorted order is one of Π possible permutations, can be sorted with $2n+\log\Pi$ pairwise comparisons.*

Sorting Pairwise Sums and its Geometric Interpretation. Fredman describes the relation between the complexity of hyperplane arrangements and the decision tree complexity of sorting pairwise sums. Grønlund and Pettie [22] use similar arguments in their 3SUM decision

[2] We consider a simplified Real RAM model. Real numbers are subject to only two unit-time operations: addition and comparison. In all other respects the machine behaves like a $w=O(\log n)$-bit word RAM with the standard repertoire of unit-time AC^0 operations: bitwise Boolean operations, left and right shifts, addition, and comparison.

[3] The independent result of Freund [19] was brought to our attention after the completion of an initial version of this paper; see [21].

tree, where they sort pairwise sums. Specifically, given lists $A = (a_i)_{i\in[n]}$ and $B = (b_i)_{i\in[n]}$ of distinct real numbers, define the pairwise sum $A + B = \{a_i + b_j \mid i, j \in [n]\}$. The input A, B can be regarded as a point $p = (a_1, \ldots, a_n, b_1, \ldots, b_n) \in \mathbb{R}^{2n}$. The points in $\mathbb{R}^{2n}$ that agree with a fixed permutation of $A + B$ form a convex cone bounded by the set H of the $\binom{n^2}{2}$ hyperplanes $x_i + y_j - x_k - y_l = 0$, for $i, j, k, l \in [n]$, $(i, j) \neq (k, l)$. The number of possible sorted orders of $A + B$ is therefore bounded by the number of regions (of all dimensions) in the arrangement $\mathcal{A}(H)$ of H. As shown by Buck [8], the number of regions in an arrangement of m hyperplanes in $\mathbb{R}^d$ of dimension $k \leq d$ is at most

$$\binom{m}{d-k}\left(\binom{m-d+k}{0} + \binom{m-d+k}{1} + \cdots + \binom{m-d+k}{k}\right).$$

Thus, the number of regions of all dimensions is $O(m^d)$ (where the constant of proportionality is independent of d). Hence, the number of possible sorting permutations of $A + B$ is $O\left((n^4)^{2n}\right) = O(n^{8n})$. One can also construct the hyperplane arrangement explicitly in $O(m^d)$ time by a standard incremental algorithm [15]. The following lemma, taken from Grønlund and Pettie [22], extends this analysis by considering only a subset of these hyperplanes, and is an immediate consequence of these observations.

▶ **Lemma 4.** *Let $A = (a_i)_{i\in[n]}$ and $B = (b_i)_{i\in[n]}$ be two lists, each of n real numbers, and let $F \subseteq [n]^2$ be a set of positions in the $n \times n$ grid. The number of realizable orders of $(A + B)_{|F} := \{a_i + b_j \mid (i, j) \in F\}$ is $O\left(\binom{|F|}{2}^{2n}\right)$, and therefore $(A + B)_{|F}$ can be sorted with at most $2|F| + 4n \log |F| + O(1)$ comparisons.*

In Lemma 4, the case $F = [n]^2$ goes back to Fredman [18], who showed that $O(n^2)$ comparisons suffice to sort $A + B$.

For some of the algorithms presented and reviewed in this paper, it is important to assume that the elements of the pairwise sum are distinct, and therefore have a unique sorting permutation. When numbers do appear multiple times, a unique sorting permutation can be obtained by breaking ties consistently (see [22] for details).

Iterative Search and Fractional Cascading. In our decision tree construction for 3SUM, we aim to speed-up binary searches of the same number, in many sorted sets. We will use for this task a special randomized variant of *fractional cascading*, which will be described in Section 4. First, we briefly recall the standard fractional cascading technique, which was introduced by Chazelle and Guibas [12, 13], for solving the *iterative search problem*, defined as follows. Let U be an ordered universe of keys. Define a *catalog* as a finite ordered subset of U. Given a set of k catalogs $C_1, C_2, \ldots, C_k$ over U, such that $|C_i| = n_i$ for each $i \in [k]$, and $\sum_{i=1}^{k} n_i = n$, the iterative search problem is to provide a data structure that supports efficient execution of queries of the form: given a query $x \in U$, return the largest value less than or equal to x in each of the k catalogs.

Fractional cascading lets one preprocess the catalogs in $O(n)$ time, using $O(n)$ storage, and answer iterative search queries in $O(\log n + k)$ time per query. This is essentially optimal in terms of query time, storage size and preprocessing time. The idea is to maintain a sufficient number of pointers across catalogs, so that, once we have the answer c_i to a query in a catalog C_i, we can follow a pointer to an element in C_{i+1}, which is only $O(1)$ indices away from the answer $c_{i+1} \in C_{i+1}$.

In order to obtain optimal query time, the fractional cascading method expands each catalog C_i to an augmented catalog L_i, starting with L_k and proceeding backwards down to L_1. L_k is the same as C_k, and for each $1 \leq i < k$, L_i is formed by merging C_i with every

second element of L_{i+1}. The items in C_i that were not originally in the catalog are marked as synthetic keys. From each synthetic key in C_i we add a bridge (pointer) to the element in L_{i+1} on which it was based. Using these bridges and additional pointers, from each real key to the two consecutive synthetic keys nearest to it, one can follow directly from each element of L_i (real or synthetic) to the elements in L_{i+1} nearest to it, and by construction, the gap between these elements is 2. Thus, given a query number x, after spending $O(\log n)$ time for searching it in L_1, it takes only $O(1)$ time to locate x in each subsequent catalog, for a total of $O(\log n + k)$ time, as desired. Since the total number of elements that were copied to the catalogs form a convergent geometric series, one can show that the total number of elements that are copied through the catalogs is only $O(n)$, and that the cost of doing it is also $O(n)$.

Fractional cascading can also be extended to support a collection of catalogs stored at the vertices of a directed acyclic graph (DAG), and each query searches with some specified element x through the catalogs stored at the nodes of some specified path in the DAG. In more detail, a *catalog graph* is a DAG in which each vertex stores a catalog (ordered list of keys). A query consists of a key x and a path π in the graph, and the goal is to search with x in the catalog of each node of π. When the maximum in/out degree Δ of the catalog graph is constant, fractional cascading can be extended to this scenario, with the same bounds as before (albeit with larger constants of proportionality). Here too each catalog C_v at a node v, is expanded into an augmented catalog L_v, and each L_v passes to its predecessors every 2Δ-th element (instead of every second element in the earlier case, where Δ was 1). See [12, 13] for more details on the construction of the data structure, proof of correctness, and performance analysis.

In our algorithms we will present a special non-standard variant of this method, that lets us preserve the advantages of the other techniques (e.g., Fredman's trick) that we use.

The Quadratic 3SUM Algorithm. We next give a brief overview of the quadratic-time algorithm. We follow the implementation given by Grønlund and Pettie [22], which is slightly different from the standard approach, but is useful for the explanation of the results of [22] and of this paper. For later references, we present the algorithm for the more general three-set version of 3SUM, as defined in the first paragraph of Section 1.

The algorithm runs over each $c \in C$ and searches for $-c$ in the pairwise sum $A+B$. With a careful implementation, given below, each search takes $O(|A| + |B|)$ time, for a total of $O(|C|(|A| + |B|))$ time. We view $A + B$ as being a matrix whose rows correspond to the elements of A and columns to the elements of B, both listed in increasing order. To help visualizing some steps of the algorithms, we think of the rows arranged in increasing order from top to bottom, and of the columns from left to right.

1. Sort A and B in increasing order as $A(0), \ldots, A(|A|-1)$ and $B(0), \ldots, B(|B|-1)$.
2. For each $c \in C$,
2.1. Initialize lo $\leftarrow 0$ and hi $\leftarrow |B| - 1$.
2.2. Repeat:
2.2.1. If $-c = A(\text{lo}) + B(\text{hi})$, report witness "$(A(\text{lo}), B(\text{hi}), c)$".
2.2.2. If $-c > A(\text{lo}) + B(\text{hi})$ then increment lo, otherwise decrement hi.
2.3. Until lo $= |A|$ or hi $= -1$.
3. If no witnesses were found report "no witness."

The correctness easily follows from the fact that each row and column of $A + B$ is sorted in increasing order. Note that when a witness is discovered in Step 2.2.1, the algorithm can stop right there. However, in order to simplify future definitions and explanations, this

372	389	407	439	454	480	534	609	635	655
397	414	432	464	479	505	559	634	660	680
420	437	455	487	502	528	582	657	683	703
442	459	477	509	524	550	604	679	705	725
478	495	513	545	560	586	640	715	741	761
500	517	535	567	582	608	662	737	763	783
523	540	**558**	590	605	631	685	760	786	806
548	565	583	615	630	656	**710**	785	811	831
594	611	629	661	676	702	756	831	857	877
627	644	662	694	709	735	789	864	890	910

Figure 1 The sky-blue colored entries form CONTOUR(710), and the purple colored ones form CONTOUR(558); A shared cell is shown in green. The lighter colors (light purple and light sky-blue) depict their *partial contour*, that is, the positions of the contours where we chose to go down. All the elements in the matrix whose values are in $[558, 710)$ are enclosed between these two contours, excluding the partial contour of 558 and including the partial contour of 710.

implementation continues to search for more witnesses. After finding a witness we will always choose to decrement hi. This choice will be made throughout the paper.

Define the *contour* of x, CONTOUR$(x, A+B)$, (CONTOUR(x), when the context is clear) to be the sequence of positions (lo, hi) encountered while searching for x in $A+B$ in the preceding algorithm. Lemma 5 is straightforward.

▶ **Lemma 5.** *For $x < y \in \mathbb{R}$,* CONTOUR*(x) lies fully above* CONTOUR*(y); that is, for each $i, i', j \in \{0, \ldots, n-1\}$, if $(i, j) \in$* CONTOUR*(x) and $(i', j) \in$* CONTOUR*(y), then $i \le i'$.*

By Lemma 5 a pair of contours can overlap, but never cross. Moreover, Lemma 5 implies a weak total order relation $\prec$ on the contours, which corresponds to the order between the searched elements, such that $x < y$ iff CONTOUR$(x) \prec$ CONTOUR(y), where the latter relation means that the two contours satisfy the properties stated in the lemma; see Figure 1.

3 Grønlund and Pettie's Subquadratic 3SUM Decision Tree

In this section we give an overview of the subquadratic decision tree of Grønlund and Pettie [22]. In the following sections we show how their ideas can be extended and combined with additional techniques, to yield our improved results.

We give an overview of the subquadratic decision tree for 3SUM over a single input set A of size n, taken from [22], resulting in a 4-linear decision tree with depth $O(n^{3/2}\sqrt{\log n})$. This is shown by an algorithm that performs at most $O(n^{3/2}\sqrt{\log n})$ comparisons, where each comparison is a sign test of a linear expression with at most 4 terms.

1. Sort A in increasing order as $A(0), \ldots, A(n-1)$. Partition A into $\lceil n/g \rceil$ groups $A_1, \ldots, A_{\lceil n/g \rceil}$, each of at most g consecutive elements, where g is a parameter that we will fix later, by setting $A_i := \{A((i-1)g), \ldots, A(ig-1)\}$, for each $i = 1, \ldots, \lceil n/g \rceil - 1$, where $A_{\lceil n/g \rceil}$ may be shorter. The first and last elements of A_i are $\min(A_i) = A((i-1)g)$ and $\max(A_i) = A(ig-1)$.
2. Sort $D := \bigcup_{i \in [n/g]} (A_i - A_i) = \{a - a' \mid a, a' \in A_i \text{ for some } i\}$.
3. For all $i, j \in [n/g]$, sort $A_{i,j} := A_i + A_j = \{a + b \mid a \in A_i \text{ and } b \in A_j\}$.
4. For k from 1 to n,

4.1. Initialize lo $\leftarrow 1$ and hi $\leftarrow \lceil n/g \rceil$.

4.2. Repeat:

4.2.1. If $-A(k) \in A_{\text{lo,hi}}$, report "solution found" and halt.

4.2.2. If $\max(A_{\text{lo}}) + \min(A_{\text{hi}}) > -A(k)$ then decrement hi, otherwise increment lo.

4.3. Until $\text{lo} = \lceil n/g \rceil + 1$ or $\text{hi} = 0$.

5. Report "no solution" and halt.

This algorithm can be generalized in a straightforward way to solve the (unbalanced) three-set version of 3SUM. For the easy argument concerning the correctness of the algorithm, see [22].

With a proper choice of g, the decision tree complexity of the algorithm is $O(n^{3/2}\sqrt{\log n})$. Step 1 requires $O(n \log n)$ comparisons. By Lemma 4, Step 2 requires $O(n \log n + |D|) = O(n \log n + gn)$ comparisons to sort D. By Fredman's trick, if $a, a' \in A_i$ and $b, b' \in A_j$, $a+b < a'+b'$ holds iff $a-a' < b'-b$, and both sides of this inequality are elements of D. Thus, Step 3 does not requires any real input comparisons, given the sorted order on D. For each iteration of the outer loop (in Step 4) there are at most $2\lceil n/g \rceil$ iterations of the inner loop (Step 4.2), since each iteration ends by either incrementing lo or decrementing hi. In Step 4.2.1 we can determine whether $-A(k)$ is in $A_{\text{lo,hi}}$ using binary search, in $\log |A_{\text{lo,hi}}| = O(\log g)$ comparisons. The total number of comparisons is thus $O(n \log n + gn + (n^2 \log g)/g)$, which becomes $O(n^{3/2}\sqrt{\log n})$ when $g = \sqrt{n \log n}$.

4 Improved Decision Trees for 3SUM, k-SUM, and k-LDT

In this section we show that the randomized decision tree complexity of 3SUM is $O(n^{3/2})$, and more generally, that the randomized decision tree complexity of k-LDT is $O(n^{k/2})$, for any odd $k \geq 3$. This bound removes the $O(\sqrt{\log n})$ factor in Grønlund and Pettie's decision tree bound. We show these results by giving a randomized algorithm that constructs a $(2k-2)$-linear decision tree whose expected depth is $O(n^{k/2})$.

To make the presentation more concise, we present it for the variant where we have three different sets A, B, C of n real numbers each, and we want to determine whether there exist $a \in A$, $b \in B$, $c \in C$, such that $a + b + c = 0$.

As in the previous section, we partition each of the sorted sets A and B into $\lceil n/g \rceil$ blocks, each consisting of g consecutive elements, denoted by $A_1, \ldots, A_{n/g}$, and $B_1, \ldots, B_{n/g}$, respectively. As above, but with a slightly different notation, we consider the $n \times n$ matrix $M = M^{AB}$, whose rows (resp., columns) are indexed by the (sorted) elements of A (resp., of B), so that $M(k, \ell) = a_k + b_\ell$, for $k, \ell \in [n]$. The partitions of A and of B induce, as before, a partition of M into n^2/g^2 boxes $M_{i,j}$, for $i, j \in [n/g]$, where $M_{i,j}$ is the portion of M with rows in A_i and columns in B_j.

As above, Fredman's trick allows us to sort all the boxes $M_{i,j}$ with $O(n \log n + ng)$ comparisons. Since the problem is fully symmetric in A, B, C, we can also define analogous matrices M^{AC} and M^{BC}, constructed in the same manner for the pairs A, C and B, C, respectively, partition each of them into n^2/g^2 boxes, and obtain the sorted orders of all the corresponding boxes, with $O(n \log n + ng)$ comparisons.

The crucial (costliest) step in Grønlund and Pettie's algorithm, which we are going to improve, is the searches of the elements of $-C$ in M^{AB}. For each $c \in C$, let $\sigma(c) = \text{CONTOUR}(-c)$ denote the staircase path contour of $-c$, as defined before Lemma 5. The length of $\sigma(c)$ is thus at most $2n$. Each of the paths $\sigma(c)$ visits some (at most $2\lceil n/g \rceil$) of the boxes $M_{i,j}$, and the index pairs (i, j) of these boxes also form a staircase pattern, as in the preceding sections. For each $c \in C$, the sequence of boxes that $\sigma(c)$ visits can be obtained by invoking (an appropriate variant of) Step 4 of the algorithm in Section 3, excluding the binary search in Step 4.2.1. The total running time of this step, over all $c \in C$, is $O(n^2/g)$.

The paths $\sigma(c)$, being contours, have the structure given in Lemma 5, including the weak total order $\prec$ between them. Thus, we obtain the following.

▶ **Corollary 6.** *For each box $M_{i,j}$, let $C_{i,j}$ denote the set of elements of C whose paths $\sigma(c)$ traverse $M_{i,j}$. Then $C_{i,j}$ is a contiguous subsequence of (the sorted) C.*

Put $\kappa_{i,j} := |C_{i,j}|$. Then we clearly have $\sum_{i,j\in[n/g]} \kappa_{i,j} = O(n^2/g)$. That is, the average number of elements of C that visit a box is $O(g)$, and, for each box, these elements form a contiguous subsequence of C, as just asserted in Corollary 6. Let $C^*_{i,j}$ denote the contiguous sequence of *indices* in C of the elements of $C_{i,j}$. That is, $C_{i,j} = \{c_\ell \mid \ell \in C^*_{i,j}\}$. With all these observations, we next proceed to derive the mechanism by which, for each box $M_{i,j}$, we can efficiently search in $M_{i,j}$ with the (negations of the) $\kappa_{i,j}$ corresponding elements of $C_{i,j}$.

We apply a special variant of fractional cascading. The twist is in the way in which we construct the augmented catalogs. Note that in each box $M_{i,j}$, we have g^2 elements of the form $a_k + b_\ell$, but only $2g$ indices k, ℓ. We want to sample elements from a box, and then copy and merge them into its neighbor boxes. However, in order to be able to use Fredman's trick, we have to preserve the property that the number of element-indices (rows and columns) in each augmented box stays $O(g)$ (unlike a naive implementation of fractional cascading, where it is enough that each augmented box be of size $O(g^2)$).

Thus, we sample elements from A (row elements) and elements from B (column elements) separately. We construct augmented sets $A'_1, \ldots, A'_{\lceil n/g\rceil}$. Starting with $A'_{\lceil n/g\rceil} = A_{\lceil n/g\rceil}$, we sample each element in $A'_{\lceil n/g\rceil}$ with probability $p = \frac{1}{4}$. Each sampled element is copied and merged with $A_{\lceil n/g\rceil-1}$, and we denote by $A'_{\lceil n/g\rceil-1}$ the new augmented set. Then we sample each element from $A'_{\lceil n/g\rceil-1}$ with the same probability p, copy and merge the sampled elements with $A_{\lceil n/g\rceil-2}$, obtaining $A'_{\lceil n/g\rceil-2}$, and continue this process until the augmented set A'_1 is constructed. Similarly, we construct the augmented sets $B'_1, \ldots, B'_{\lceil n/g\rceil}$, but we do it in the opposite direction, starting from $B'_1 = B_1$ and ending with $B'_{\lceil n/g\rceil}$. Clearly, similar to standard fractional cascading, the expected size of each of the augmented sets is $O(g)$, as the expected numbers of additional elements placed in each box form a convergent geometric series. Now we sort

$$D_{A'} = \bigcup_{i\in[n/g]} (A'_i - A'_i) = \{a - a' \mid a, a' \in A'_i \text{ for some } i\}.$$

In each $A'_i - A'_i$, the expected number of elements $a_k - a_{k'}$ is $O(g^2)$, and the expected number of element indices k, k' is only $O(g)$. Thus, by Lemma 4, we can sort $D_{A'}$ with expected $O(n \log n + ng)$ comparisons. Similarly, we sort $D_{B'} = \bigcup_{j\in[n/g]} (B'_j - B'_j)$ with the same expected number of comparisons. Then, we form the union $D' = D_{A'} \cup D_{B'}$ and obtain its sorted order by merging $D_{A'}$ and $D_{B'}$. This costs additional expected $O(ng)$ comparisons. By Fredman's trick, from the sorted order of D', we can, and do, obtain the sorted order of the augmented boxes $A'_i + B'_j$, for each $i, j \in [n/g]$, without further comparisons.

With these augmentations of the row and column blocks, the matrix M^{AB} itself is now augmented, such that each modified box $M_{i,j} = A'_i + B'_j$ receives some fraction of the rows from the box $M_{i+1,j}$ below it, and a fraction of the columns from the box $M_{i,j-1}$ to its left. Each box $M_{i,j}$ corresponds to a vertex in the catalog graph, and it has (at most) two outgoing edges, one to the vertex that corresponds to $M_{i+1,j}$ and one to the vertex that corresponds to $M_{i,j-1}$ (it also has at most two incoming edges). Clearly this is a DAG with maximum in/out degree $\Delta = 2$, which is why we sampled $\frac{1}{2\Delta} = \frac{1}{4}$ of the rows/columns in each step. We complete the construction of this special fractional cascading data structure, by adding the appropriate pointers, similar to what is done in a standard implementation of

60	70	80	90	100	110	120	130	140	150
160	170	180	190	200	210	220	230	240	250
260	270	280	290	300	310	320	330	340	350

Figure 2 An expensive step in the fractional cascading search: Assume that only the first and third rows (appearing in gray) are sent to the preceding box (above the current one), and that we search with $-c = 205$. The previous search locates $-c$ between $\xi^- = 150$ and $\xi^+ = 260$, say, and now we have to examine the entire second row to locate $-c$ in the current box.

fractional cascading (see Section 2). This does not require any further comparisons, since the pointers from synthetic keys (the sampled elements) to real keys, and pointers from real keys to synthetic keys, depend only on the sorted order of the augmented sets $M_{i,j}$, which we already computed. So the overall expected number of comparisons needed to construct this data structure is still $O(n \log n + ng)$.

Consider now the search with $-c$, for some $c \in C$. Assume that the search has just visited some box $M_{i',j'}$, and now proceeds to search in box $M_{i,j}$, where either $(i,j) = (i'+1, j')$ or $(i,j) = (i', j'-1)$. Assume, without loss of generality, that $(i,j) = (i'+1, j')$; a symmetric argument applies when $(i,j) = (i', j'-1)$, using columns instead of rows. In this case, the fractional cascading mechanism has sampled, in a random manner, an expected quarter of the rows of (the already augmented) $M_{i,j}$ and has sent them to $M_{i',j'} = M_{i-1,j}$. The output of the search at $M_{i-1,j}$, if $-c$ was not found there, includes two pointers to the largest element ξ^- of $M_{i,j}$ that is smaller than $-c$, and to the smallest element ξ^+ of $M_{i,j}$ that is larger than or equal to $-c$. We need to go over the elements in the sorted order of $M_{i,j}$ that lie between ξ^- and ξ^+, and locate $-c$ among them. If we do not find it, we get the two consecutive elements that enclose $-c$, retrieve from them two corresponding pointers to a pair of elements in the next box to be searched, that enclose $-c$ between them, and continue the fractional cascading search in the next box, in between these elements.

The main difficulty in this approach is that the number of elements of $M_{i,j}$ between ξ^- and ξ^+ might be large, because there might be many elements between ξ^- and ξ^+ in rows that we did not sample, and then we have to inspect them all, slowing down the search.

Concretely, in this case we sample, in expectation, a quarter of the rows of $M_{i,j}$ (recall that, we actually sample the rows from an augmented box that has already received data from previous boxes, but let us ignore this issue for now). Collectively, these rows contain (in expectation) $\Theta(g^2)$ elements of $M_{i,j}$, but we have no good control over the size of the gaps of non-sampled elements between consecutive pairs of sampled ones. This is because there might be rows that we did not sample which contain many elements between ξ^- and ξ^+, and searching through such large gaps could slow down the procedure considerably. See Figure 2 for an illustration. (For a normal fractional cascading, this would not be an issue, but here the peculiar and implicit way in which we sample elements has the potential for creating this problem.)

We handle this problem as follows. Consider any gap of non-sampled elements of $M_{i,j}$ between a consecutive pair $\xi^- < \xi^+$ of sampled ones. We claim that the expected number of rows to which these elements belong is $O(1)$. Indeed, the probability to have k distinct rows in such a gap, conditioned on the choice of the row containing ξ^-, is $\frac{1}{4}\left(\frac{3}{4}\right)^k$, which follows since each row is sampled *independently* with probability $1/4$. Hence, the (conditionally)

expected row-size of a gap is

$$\sum_{k\geq 0} k\frac{1}{4}\left(\frac{3}{4}\right)^k = O(1),$$

as claimed. Denote this expected value as β. In other words, for each $c \in C_{i,j}$, let R_c be the set of rows that show up in the gap between the corresponding elements ξ^- and ξ^+ for c. The overall expected size $\sum_{c\in C_{i,j}} |R_c|$ is thus $\beta|C_{i,j}|$.

Fix a box $M_{i,j}$. For each $\ell \in C^*_{i,j}$ and for each $k \in R_{c_\ell}$, we need to locate $-c_\ell$ among the elements in row k of $M_{i,j}$. That is, we need to locate $-c_\ell$ among the elements of the set $a_k + B'_j$. This however is equivalent to locating $-a_k - c_\ell$ among the elements of B'_j.

We therefore collect the set S of all the sums $-a_k - c_\ell$, for $\ell \in C^*_{i,j}$ and $k \in R_{c_\ell}$, and recall that in expectation we have $|S| = O(|C_{i,j}|)$. The crucial observation is that we already (almost) know the order of these sums. To make this statement more precise, partition, in the usual manner, the sorted sequence C into $\lceil n/g\rceil$ blocks $C_1, C_2, \ldots, C_{\lceil n/g\rceil}$, each consisting of g consecutive elements in the sorted order. As mentioned earlier, a symmetric application of Fredman's trick allows us to obtain the sorted order of each box of the form $A'_i + C_j$, using a total of $O(ng)$ comparisons.

The number of (consecutive) blocks C_s of C that overlap $C_{i,j}$ is $t_{i,j} \leq \lceil \kappa_{i,j}/g\rceil + 2$. Moreover, each sum in S belongs to $-(A'_i + C_s)$ for one of these $t_{i,j}$ blocks. Since each of these sets is already sorted, we extract from them (with no extra comparisons) the elements of S as the union of $t_{i,j}$ sorted sequences $S_{i,s}$, where $S_{i,s} \subset -(A'_i + C_s)$ for each s. Arguing as above, the expected size of $S_{i,s}$ is $\beta|C_s| = O(g)$. We now merge each of the sorted sequences $S_{i,s}$ with B'_j, using an expected $O(g)$ comparisons for each merge. As a result, each sum $-a_i - c_\ell$ is located between two consecutive elements $b^-_{i,\ell} < b^+_{i,\ell}$ of B'_j. In other words, for each $c_\ell \in C_{i,j}$, we have at most $|R_{c_\ell}|$ candidates for being the largest element of $M_{i,j}$ that is smaller than $-c_\ell$ (these are the elements $a_i + b^-_{i,\ell}$, for $i \in R_{c_\ell}$), and we select the largest of them, requiring no comparisons, as these are all elements of the already sorted $A'_i + B'_j$. In the same manner, we find the smallest element of $M_{i,j}$ that is larger than $-c_\ell$. Having found these two elements, we can proceed to search $-c_\ell$ in the next box, using the appropriate pointers created by the fractional cascading mechanism (see Section 2).

The overall number of merges is

$$\sum_{i,j\in[n/g]} t_{i,j} \leq \sum_{i,j\in[n/g]} (\kappa_{i,j}/g + 2) = O(n^2/g^2),$$

and each of them costs $O(g)$ expected comparisons, for a total of $O(n^2/g)$ expected comparisons. Thus, the overall number of expected comparisons is $O(n\log n + ng + n(\log g + n/g))$, which is $O(n^{3/2})$, when $g = \sqrt{n}$. This completes the proof of Theorem 1. ◀

4.1 k-SUM and Linear Degeneracy Testing

The standard algorithm for k-variate linear degeneracy testing (k-LDT) for odd $k \geq 3$, is based on a straightforward reduction to an instance of unbalanced 3SUM, where $|A| = |B| = n^{(k-1)/2}$ and $|C| = n$; see [4] and [22]. The analysis of this section also applies for unbalanced 3SUM, and directly implies that it can be solved by using an expected number of

$$O\left(|A|\log|A| + |B|\log|B| + |C|\log|C| + g\left(|A| + |B| + |C|\right) + |C|\left((|A| + |B|)/g + \log g\right)\right)$$

comparisons, where the first four terms come from the cost of sorting the blocks of (the augmented) M^{AB}, M^{AC}, and M^{BC}, and where the last term is the cost of the fractional cascading searches. We have $|A| = |B| = n^{(k-1)/2}$, $|C| = n$, so by choosing $g = \sqrt{n}$, the

bound becomes $O(n^{k/2})$. Thus, the randomized decision tree complexity of k-LDT (and thus of k-SUM) is $O(n^{k/2})$, for any odd $k \geq 3$, as stated in Theorem 2.

References

1. A. Abboud and V. Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. 55th Annu. Sympos. on Foundations of Computer Science (FOCS)*, pages 434–443, 2014.
2. A. Abboud, V. Vassilevska Williams, and H. Yu. Matching triangles and basing hardness on an extremely popular conjecture. In *Proc. 47th Annu. ACM on Sympos. on Theory of Computing (STOC)*, pages 41–50, 2015.
3. O. Aichholzer, F. Aurenhammer, E. D. Demaine, F. Hurtado, P. Ramos, and J. Urrutia. On k-convex polygons. *Comput. Geom.*, 45(3):73–87, 2012.
4. N. Ailon and B. Chazelle. Lower bounds for linear degeneracy testing. *J. ACM*, 52(2):157–171, 2005.
5. A. Amir, T. M. Chan, M. Lewenstein, and N. Lewenstein. On hardness of jumbled indexing. In *Proc. 41st Int'l Colloq. on Automata, Languages, and Programming (ICALP)*, pages 114–125, 2014.
6. A. Amir, T. Kopelowitz, A. Levy, S. Pettie, E. Porat, and B. Riva Shalom. Mind the gap: Essentially optimal algorithms for online dictionary matching with one gap. In *Proc. 27th Int'l Sympos. on Algorithms and Computation (ISAAC)*, pages 12:1–12:12, 2016.
7. G. Barequet and S. Har-Peled. Polygon containment and translational min-Hausdorff-distance between segment sets are 3SUM-hard. *Int. J. Comput. Geometry Appl.*, 11(4):465–474, 2001.
8. R. C. Buck. Partition of space. *Amer. Math. Monthly*, 50:541–544, 1943.
9. A. Butman, P. Clifford, R. Clifford, M. Jalsenius, N. Lewenstein, B. Porat, E. Porat, and B. Sach. Pattern matching under polynomial transformation. *SIAM J. Comput.*, 42(2):611–633, 2013.
10. J. Cardinal, J. Iacono, and A. Ooms. Solving k-SUM using few linear queries. In *Proc. 24th Annu. European Sympos. on Algorithms (ESA)*, pages 25:1–25:17, 2016.
11. M. L. Carmosino, J. Gao, R. Impagliazzo, I. Mihajlin, R. Paturi, and S. Schneider. Nondeterministic extensions of the strong exponential time hypothesis and consequences for non-reducibility. In *Proc. 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 261–270, 2016.
12. B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.
13. B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1(2):163–191, 1986.
14. E. D. Demaine, J. S. B. Mitchell, and J. O'Rourke. The open problems project. Accessed: 2015-10-28.
15. H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM J. Comput.*, 15(2):341–363, 1986.
16. J. Erickson. Bounds for linear satisfiability problems. *Theor. Comput. Sci*, 8:388–395, 1999.
17. E. Ezra and M. Sharir. A nearly quadratic bound for the decision tree complexity of k-SUM. *To Appear in Proc. 33st Int'l Sympos. on Computational Geometry (SoCG)*, 2017.
18. M. L. Fredman. How good is the information theory bound in sorting? *Theor. Comput. Sci*, 1(4):355–361, 1976.
19. A. Freund. Improved subquadratic 3SUM. *Algorithmica*, 77(2):440–458, 2017.
20. A. Gajentaan and M. H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Comput. Geom.*, 5:165–185, 1995.

21 O. Gold and M. Sharir. Improved bounds for 3SUM, k-SUM, and linear degeneracy. *CoRR*, abs/1512.05279, 2015. URL: `http://arxiv.org/abs/1512.05279`.

22 A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proc. 55th Annu. Sympos. on Foundations of Computer Science (FOCS)*, pages 621–630, 2014.

23 A. Hernández-Barrera. Finding an $o(n^2 \log n)$ algorithm is sometimes hard. In *Proc. 8th Canadian Conference on Computational Geometry*, pages 289–294, 1996.

24 R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *J. Comput. Syst. Sci.*, 62(2):367–375, March 2001.

25 R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001.

26 D. M. Kane, S. Lovett, and S. Moran. Near-optimal linear decision trees for k-SUM and related problems. *CoRR*, abs/1705.01720, 2017.

27 T. Kopelowitz, S. Pettie, and E. Porat. Higher lower bounds from the 3SUM conjecture. In *Proc. 27th Annu. ACM-SIAM Sympos. on Discrete Algorithms (SODA)*, pages 1272–1287, 2016.

28 A. Lincoln, V. Vassilevska Williams, J. R. Wang, and R. Williams. Deterministic time-space trade-offs for k-SUM. In *Proc. 43rd Int'l Colloq. on Automata, Languages, and Programming (ICALP)*, pages 58:1–58:14, 2016.

29 M. Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *Proc. 42nd ACM Sympos. on Theory of Computing (STOC)*, pages 603–610, 2010.

30 M. A. Soss, J. Erickson, and M. H. Overmars. Preprocessing chains for fast dihedral rotations is hard or even impossible. *Comput. Geom.*, 26(3):235–246, 2003.

31 O. Weimann, A. Abboud, and V. Vassilevska Williams. Consequences of faster sequence alignment. In *Proc. 41st Int'l Colloq. on Automata, Languages, and Programming (ICALP)*, pages 39–51, 2014.

32 V. Vassilevska Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In *Proc. 51st Annu. IEEE Sympos. on Foundations of Computer Science (FOCS)*, pages 645–654, 2010.

Profit Sharing and Efficiency in Utility Games*

Sreenivas Gollapudi[1], Kostas Kollias[2], Debmalya Panigrahi[3], and Venetia Pliatsika[4]

1 **Google Research, Mountain View, USA**
2 **Google Research, Mountain View, USA**
3 **Duke University, Durham, USA**
4 **University of Pennsylvania, Philadelphia, USA**

Abstract

We study utility games (Vetta, FOCS 2002) where a set of players join teams to produce social utility, and receive individual utility in the form of payments in return. These games have many natural applications in competitive settings such as labor markets, crowdsourcing, etc. The efficiency of such a game depends on the profit sharing mechanism – the rule that maps utility produced by the players to their individual payments. We study three natural and widely used profit sharing mechanisms – egalitarian or equal sharing, marginal gain or value addition when a player joins, and marginal loss or value depletion when a player leaves. For these settings, we give tight bounds on the price of anarchy, thereby allowing comparison between these popular mechanisms from a (worst case) social welfare perspective.

1998 ACM Subject Classification J.4 Social and Behavioral Sciences: Economics

Keywords and phrases Price of anarchy, submodular maximization, coverage functions

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.43

1 Introduction

In *utility games* (introduced by Vetta [20], see also [14]), individual agents (e.g., employees) offer their services to entities (e.g., employers) to create social utility, and receive individual utility in the form of payments in return. It is natural to expect the agents to behave strategically, i.e., offer their services to the entity giving them the highest payment. This represents a game where each agent (called a *player*) selects one of the available entities (called *teams*) to maximize their individual payments (called *payoffs*), but the overall *social welfare* is the total utility cumulatively produced by all the teams. A stable outcome, called a *Nash equilibrium* or NE, is achieved when no player can unilaterally change her team and increase her payoff. The goal of this paper is to study the (in)efficiency of such stable outcomes – the maximum (worst-case) ratio between the social welfare produced by an optimal allocation and that in an NE, called the Price of Anarchy or POA of the game. Clearly, the social welfare produced at equilibrium depends on the *profit sharing* mechanism in use, i.e., the payoff of the players as a function of the utility produced by them. We consider three natural and widely used profit sharing rules:

- *egalitarian*: any unit of utility produced by a team is divided equally among the team members who helped produce it.

* A full version of this paper is available at `http://theory.stanford.edu/~kkollias/profit_sharing.pdf`

- *marginal gain*: the payoff of a player in a team is the utility that the team gained when she joined.
- *marginal loss*: the payoff of a player in a team is the utility that the team would lose if she were to leave.

The main motivation for this work is to compare these popular profit sharing mechanisms in terms of their impact on (worst case) social utility.

Formally, there is a set of *players* N and a set of *teams* T that they can join. The utility produced by a team is given by a *weighted coverage function* over the players in the team. To interpret this, consider a set of tasks S with respective utilities v_s for $s \in S$. If each player $i \in N$ can perform a subset of tasks $S_i \subseteq S$, then the tasks completed by a team $t \in T$ is given by $S_t = \cup_{i \in t} S_i$, and the utility produced is $U_t = \sum_{s \in S_t} v_s$. This is precisely a weighted coverage function over the team members; we say that team t *covers* task s if $s \in S_t$.

From a social perspective, the goal is to maximize the total utility produced by all teams,

$$U = \sum_{t \in T} U_t = \sum_{t \in T} \sum_{s \in S_t} v_s.$$

We will call U the *social welfare* or *objective value*.

Profit Sharing Mechanisms

Each player i is interested in maximizing her own payoff, denoted u_i, which depends on the profit sharing mechanism. We consider the following popular profit sharing mechanisms.

Egalitarian Profit Sharing. For every task that is covered in a team, the utility of the task is equally shared among the members of the team who perform the task. We may note that the egalitarian model is an instantiation of the *Shapley value* utility sharing method [19]. This is defined as the expected contribution of a player to her team's utility, assuming players are sequentially added to the team using a uniformly random ordering.

For egalitarian sharing, we exactly determine the POA to be 1.6 by proving matching upper and lower bounds on the POA. The upper bound employs the smoothness framework due to Roughgarden [18]. However, unlike the standard approach of applying the smoothness inequality for every resource, (in our setting, for every team), we apply the smoothness inequality across all teams and players simultaneously for a single task. The matching lower bound of 1.6, on the other hand, uses a careful combinatorial construction of the worst case POA instance, using symmetrization techniques to argue stability of the solution. Both these results appear in Section 2, thereby proving the following theorem.

▶ **Theorem 1.** *The* POA *of egalitarian profit sharing is* 1.6.

Marginal Gain Profit Sharing. In this model, players have an order of arrival and each player's utility is the value added to the team when that player joins. The marginal gain method is an instantiation of an *ordinal Shapley value*, which is a variation on the Shapley value discussed above where player ordering is not random but predefined.

For marginal gain profit sharing, we show an upper bound of 1.71 and a lower bound of 1.58 on the POA. The lower bound can be established by hardness of approximation results (see [11]) assuming $\mathbf{P} \neq \mathbf{NP}$. We give an alternative proof of this result based on an explicit construction that does not rely on complexity theoretic assumptions in our full paper. On the other hand, our upper bound, which appears in Section 3, is based on a charging argument, which carefully matches tasks that are not covered in the NE to covered tasks.

▶ **Theorem 2.** *The* POA *of marginal gain profit sharing is at most* $1 + \frac{1}{\sqrt{2}} \simeq 1.71$ *and at least* $\frac{e}{e-1} \simeq 1.58$.

Marginal Loss Profit Sharing. In this model, the utility of a player is the value lost if she were to leave the team. The marginal loss model is an instantiation of the *marginal contribution* method, where each player is rewarded with her marginal contribution to the utility of her team [19, 10]. One interesting point of difference between the this and the previous two models is that the sum of individual payoffs in this case may be strictly smaller than the overall social utility, whereas this sum was exactly equal to the social utility in the previous cases. This is because the only tasks whose utility is awarded as payoff in this model are those that are uniquely performed by a single team member.

For marginal loss profit sharing, we prove the POA is exactly 2. An upper bound of 2 follows from the work of Vetta [20]. We show a matching lower bound via an explicit construction with ideas that are similar to the lower bound construction in Theorem 1. Due to space constraints, we present this lower bound construction in our full paper, thereby proving the next theorem.

▶ **Theorem 3.** *The* POA *of marginal loss profit sharing is* 2.

Existence of NE**.** Omitting further details, we briefly mention why existence of a NE is guaranteed in all three profit sharing models. Egalitarian profit sharing induces a congestion game [17, 15], which implies existence of a NE is guaranteed. In marginal gain profit sharing, the property follows by equivalence to the process of having players appear online and letting each player select the team yielding the higher profit at the time of her arrival. In marginal loss profit sharing, the optimal solution is always a NE since any beneficial deviation by a player by definition increases the objective value.

Extensions

Submodular Utilities. While we primarily consider utility functions that are weighted coverage functions of the players in a team, Vetta [20] has originally proposed the utility game framework for more general *submodular* utility functions. The marginal gain and marginal loss profit sharing models naturally extend to this setting with the same definitions. In both cases, we show that the POA is 2, i.e., that Vetta's upper bound is tight. For the marginal loss model, this follows as a corollary of Theorem 3 since weighted coverage functions are a special case of submodular functions. For the marginal gain mode, however, this establishes a separation between the POA for general submodular functions and the special case of weighted coverage functions. Our lower bound construction for general submodular functions makes use of a knapsack welfare function, and appears in our full paper.

▶ **Theorem 4.** *The* POA *of marginal gain and marginal loss profit sharing with submodular utilities is* 2.

One can also extend the egalitarian profit sharing model to the case of general submodular functions by using the analogy with Shapley profit sharing. In particular, for submodular functions, the payoff of a player in the egalitarian case is her expected contribution to the utility of the team, assuming a uniform random order of arrival of players. Determining the POA in this case is an interesting problem for future work.

Asymmetric Utilities. We can also consider an *asymmetric* setting, where the utility functions of teams are not necessarily identical. This is the case, e.g., if a task produces different utility to different teams. In this case, even for weighted coverage utility functions (and therefore, also for submodular utilities), we show that the POA is 2, i.e., Vetta's upper bound is tight. For the marginal loss model, this follows from Theorem 3 since symmetric utilities are a special case, but for the egalitarian and symmetric gain models, we require new lower bound constructions that are given in our full paper.

▶ **Theorem 5.** *The* POA *in the egalitarian, marginal gain, and marginal loss models for asymmetric teams is 2.*

Related Work

Utility games were introduced by Vetta [20] to model strategic agents who produce submodular social welfare in a team, and seek to maximize their individual utility or payoff in return. For this general setting, Vetta showed an upper bound of 2 on the POA, subject to some mild conditions on the agent payoff functions that are satisfied in all our models above. The profit sharing models that we study in this paper are inspired by standard cost sharing rules from the economics literature such as Shapley and marginal contribution costs as mentioned earlier. The POA of these cost sharing models has been studied in several resource selection problems with negative externalities among players. Specifically, Marden and Wierman [14] studied utility sharing methods in a general distributed utility maximization model, and Harks and Miller [9] studied cost sharing methods in networking applications. Bachrach *et al.* [1] considered the effect of *positive* externalities among players working on multiple projects simultaneously, but restricted by an effort budget. In [13], the authors study a special case of utility games where the welfare produced by a resource is a function of the number of players on it, and prove that under certain conditions (such as symmetric players), the POA drops below 2.

Utility games with coverage utility functions are also related to congestion games [17, 15]; in fact, utility games in the egalitarian profit sharing model *are* congestion games. The POA of cost sharing methods in generalizations of congestion games has been extensively studied [3, 12, 8]. Gairing [7] studied a congestion game with a coverage utility function and showed how to modify the payoffs of the tasks so that better-response dynamics reaches an equilibrium with an inefficiency of $1 - \frac{1}{e}$ or better in polynomial time.

Finally, we mention known results for the corresponding optimization problem – make an assignment of players to teams to maximize overall social utility, where the utility on every team is given by a weighted coverage function. This problem is called *submodular welfare maximization with coverage functions*. The best approximation ratio in the general case is $\frac{e}{e-1} \simeq 1.58$ [2, 5, 6]. Algorithmic results on combinatorial auctions, which are similar to our setting (teams are bidders and players are items) include a $1 - \frac{1}{e}$ approximation algorithm for submodular valuations [4], a proof of optimality of the greedy algorithm in various online and offline settings [16], and a (matching) hardness of approximation result [11].

2 Egalitarian Payoffs

In this section, we prove Theorem 1, i.e., show that the POA for the egalitarian profit sharing model is exactly 1.6. This comprises two parts: a lower bound of 1.6 (in Section 2.1) and a matching upper bound of 1.6 (in Section 2.2).

2.1 Lower Bound for Egalitarian Payoffs

▶ **Lemma 6.** *The* POA *of egalitarian profit sharing is at least* 1.6.

We construct of an instance of the egalitarian model and an assignment of players to teams that is an NE and whose social welfare is a $\frac{5}{8}$ fraction of the optimal solution. We begin with an overview of this construction, and then give details of each step. First, we create a simple instance parameterized by integers x and y (we will precisely define these integers later), and an assignment of players to teams with utility $\frac{x+y}{2x+y}$ times the optimal. Our assignment in this preliminary game will *not* be an NE. We then modify the instance in two stages, where we preserve the ratio $\frac{x+y}{2x+y}$ w.r.t. the optimal solution, while creating sufficient structure to argue that the final assignment is an NE for appropriate values of x and y. The worst case among these equilibrium-inducing (x, y) values will yield the POA lower bound of 1.6.

Checking whether the final assignment is an NE can be a complicated task in general, since there will eventually be a large number of players and possible deviations in the game. Our two-stage transformation will ensure, however, that this task reduces to verifying a single inequality. This will be achieved by imposing symmetry across players (first transformation) and symmetry across possible deviations of a player (second transformation). We now present the four stages of our proof (initialization, imposing player symmetry, imposing deviation symmetry, and picking the values of x, y) in detail.

Stage 1: Initialization. Our preliminary game uses the parameter $k = 2x + y$. There are k tasks $s_1, s_2, \ldots, s_k$, and k *types* of players where a player of type i can *only* perform task s_i. There are k players for each type, i.e., a total of k^2 players. The number of teams is also k. The utility produced by covering any single task in a team is 1.

We will crucially maintain two properties of the assignment. The first property imposes symmetry over how players are divided among teams.

▶ **Property 7.** *Note that there are $k = 2x+y$ players who can perform a task. Our assignment will ensure that every task is covered by 2 players in x teams, by 1 player in y teams, and remains uncovered in x teams. We will also ensure that every team has $k = 2x + y$ players. These k players in any team will cover tasks as follows: x tasks will be covered by 2 players, y tasks will be covered by 1 player, and x tasks will remain uncovered.*

Note that the above property ensures that every team only covers $x + y$ tasks out of the total of $k = 2x + y$ tasks. Similarly, every task is covered in only $x + y$ teams out of the total of $k = 2x + y$ teams.

The second property relates our assignment to an optimal assignment (call it OPT). To encode OPT, let us use k colors $c_1, c_2, \ldots, c_k$, where all players assigned to team i by OPT are said to have color c_i.

▶ **Property 8.** OPT *will satisfy the property that there is exactly one player with color c_i who can perform a specific task s_j, for any i and j. In other words, the $k = 2x + y$ players who can perform any specific task will be divided among the $k = 2x + y$ teams, thereby ensuring that all tasks in all teams are covered. Contrast this to our assignment that only covers $x + y$ tasks in every team, and $x + y$ teams cover every task, according to Property 7. Finally, in our assignment, there will be exactly one player of each color c_i in every team t. In other words, the overlap between any team in our assignment and any team in* OPT *will be exactly one player.*

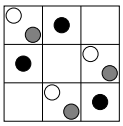


Figure 1 The preliminary assignment for $x = y = 1$. Teams are rows, tasks are columns. Cell (i, j) corresponds to team i, task s_j. OPT has all white players in team 1, all gray players in team 2, and all black players in team 3. The occupancy is symmetric across rows and columns and each color appears once per row and column.

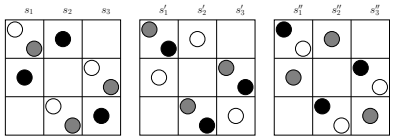


Figure 2 The intermediate assignment for $x = y = 1$. The first copy is the original preliminary assignment. In the second one, white becomes gray, gray becomes black, black becomes white. In the third we shift the colors again. All three together form the intermediate assignment.

As noted above, Property 7 implies that the coverage of our assignment is $\frac{x+y}{2x+y}$ times the total number of tasks, while Property 8 ensures that the optimal solution covers every task. However, it is not aprioiri clear that these properties can be satisfied by an assignment: the next lemma asserts this.

▶ **Lemma 9.** *Given k teams, k tasks, and, for each task, k players who can perform only that task, there is an assignment of the players to the teams and a coloring that satisfies Properties 7 and 8.*

Proof. (See Fig. 1 for an illustration of the $x = y = 1$ case.) The first team's structure is as follows: tasks $s_1, s_2, \ldots, s_x$ are covered by two players, tasks $s_{x+1}, s_{x+2}, \ldots, s_{x+y}$ are covered by one player each, and tasks $s_{x+y+1}, s_{x+y+2}, \ldots, s_k$ are left uncovered. For this first team, we use any coloring that has a different color for each of the k players. The structure and coloring of the second team is obtained by performing a left circular shift to the first team's structure, i.e., $s_k, s_1, s_2, \ldots, s_{x-1}$, are covered by 2 players, $s_x, s_{x+1}, \ldots, s_{x+y-1}$ are covered by 1 player, and $s_{x+y}, s_{x+y+1}, \ldots, s_{k-1}$, are left uncovered. Colors are also shifted, i.e., the color(s) of the player(s) covering s_i in the first team is applied to the player(s) covering s_{i-1} (s_k, if $i = 1$) in the second team. We continue with similar left circular shifts to define the remaining teams. This assignment and coloring satisfies Properties 7 and 8. ◀

We have now completed the first stage; we will call this the *preliminary assignment*. By Property 8, the optimal assignment covers all tasks; hence, the ratio of the coverage of this preliminary assignment to the optimum is $\frac{x+y}{2x+y}$. However, this assignment is not an NE, since players sharing a task have unilateral incentive to deviate to a team where the corresponding task is not covered. We now proceed to the next stages, which will modify this assignment to an NE.

Stage 2: Imposing player symmetry. During this stage, we will augment the game by adding new tasks. In our preliminary assignment, not all players have the same payoff since some of them share a task with a teammate while others do not. In this stage, we impose symmetry across players: every player will share exactly $2x$ tasks with another player and will cover exactly y tasks by herself. To do this, we create k copies of our preliminary assignment, and exchange roles between players in the different copies in a way that they all end up being symmetric. We will call this the *intermediate assignment*.

The first copy is identical to the preliminary assignment. In the second copy, we take the preliminary assignment and perform a circular shift on the colors, i.e., we change color c_i to

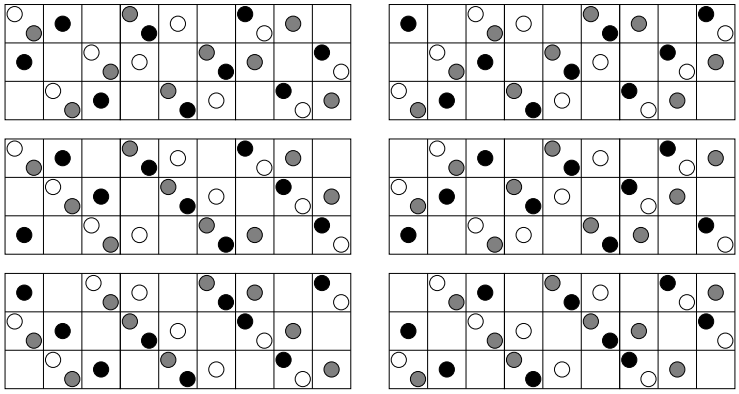


Figure 3 All 6 versions of the intermediate assignment for $x = y = 1$. The first one is the original intermediate assignment. The rest are all possible permutations of the team structures (i.e., rows). All 6 together form the final assignment with 9 players: one white, one gray, and one black player in each team.

color c_{i+1} (color c_k changes to c_1). Next, we rename the tasks so that they are distinct from those in the first copy. We continue this process of doing a circular shift on the colors and renaming the tasks in each subsequent copy until we have k copies in total. (See Fig. 2 for all the copies of the $x = y = 1$ case.) The intermediate assignment is constructed by appending all k copies (recall that the tasks are distinct in the copies), and merging all players in the same team with the same color into a single player.

Note that properties 7 and 8 continue to hold; in particular, this implies that the intermediate assignment covers a $\frac{x+y}{2x+y}$ fraction of tasks in each team, while OPT covers every task in every team. Moreover, since every color assumes the role of every other color in the preliminary assignment in one of the copies, it follows that every player covers $2x$ tasks with another player and y tasks by herself in the intermediate assignment. This implies that the players are symmetric in their coverage and payoff in their current team. However, the possible deviations of a player to another team are not symmetric, i.e., the payoff of a player depends on the team that the player moves to. In the next stage, we impose symmetry on the deviations of players, thereby reducing the equilibrium condition to a single inequality.

Stage 3: Imposing Deviation Symmetry. In this stage, we repeatedly perform an operation that we call *team structure switch*. Switching the structure of team t to that of t' involves taking each player in t, stripping her of her existing tasks, and granting her the tasks of the player in t' with the same color. By Property 8, this player in t' is uniquely defined given a specific player in t. A *team structure permutation* is said to be performed when we switch the structure of every team t to the structure of team $\pi(t)$, where π is a permutation on the teams T.

For every possible permutation π, we generate a copy of the intermediate assignment and perform a team structure permutation based on π. As we did in the previous stage, we rename tasks so that they are different for each permuted copy and incorporate all $k!$ copies into our game by merging players in the same team with the same color into a single player with $k \cdot k!$ tasks. This generates our *final assignment*. (See Fig. 3 for the copies corresponding to the six permutations for the $x = y = 1$ case.)

Again, note that Properties 7 and 8 continue to hold; as a consequence, each team only covers a $\frac{x+y}{2x+y}$ of the tasks in the final assignment whereas OPT covers every task in every

team. Additionally, every deviation of a player to another team now result in exactly the same utility; therefore, not only are the players symmetric in their current team, but their deviation to any other team is also symmetric.

▶ **Lemma 10.** *In the final assignment, the utility of any player i who deviates to a team t' that is not her assigned team t is given by*

$$\left[\left(\frac{x-1}{3}+\frac{y}{2}+x\right)2x+\left(\frac{x}{3}+\frac{y-1}{2}+x\right)y\right]k(k-2)!.$$

Proof. Consider a player i, and call her assigned team t. Fix some structure for t in an intermediate assignment, and focus on all versions of the intermediate assignment in which team t has that structure. There will be $(k-1)!$ such versions. Consider any task s that is covered by i and another player in this structure. Our first goal is to determine the coverage of task s in any other team t' in each of the $(k-1)!$ versions of the intermediate assignment that we are considering.

The copies of t' in the $(k-1)!$ versions we are considering can assume one of $k-1$ possible structures, excluding the structure that we have fixed for t. Each one of these $k-1$ possible structures for t' appears an equal number of times, i.e., $(k-2)!$ times. By Property 7, $x-1$ of these $k-1$ structures have 2 players covering task s. (Note that t itself has 2 players covering s, hence the number is $x-1$ and not x.) Similarly, in y of these structures, s is covered by a single player, and it is not covered at all in x structures. This implies that the payoff of i due to s, if she deviates to t', will be $\frac{x-1}{3}+\frac{y}{2}+x$ when we sum across one copy of each structure of t'. For the overall payoff of i after deviation to t' due to tasks shared with another player in t, we need to multiply this expression by:

- $2x$, which represents the number of different tasks s that are covered by i and another player in t,
- $(k-2)!$, which represents the number of copies with the same structure of t', given a fixed structure of t, and
- k, which is the number of different structures of t.

This yields a payoff of:

$$\left(\frac{x-1}{3}+\frac{y}{2}+x\right)2xk\cdot(k-2)!. \tag{1}$$

In a similar manner, we can calculate the payoff that i would get by deviating to t' due to the tasks she uniquely covers in t. This comes out to:

$$\left(\frac{x}{3}+\frac{y-1}{2}+x\right)yk\cdot(k-2)!. \tag{2}$$

The total payoff after deviation for player i is then given by:

$$\left[\left(\frac{x-1}{3}+\frac{y}{2}+x\right)2x+\left(\frac{x}{3}+\frac{y-1}{2}+x\right)y\right]k\cdot(k-2)!, \tag{3}$$

which is independent of i, t and t'. This completes the proof of the lemma. ◀

Stage 4: Choice of the parameters x and y. Note that the payoff of a player in the final assignment is $(x+y)k!$, since the payoff in every copy of the intermediate assignment is $x+y$ and there are $k!$ copies. Therefore, by Lemma 10, the equilibrium condition is:

$$\left[\left(\frac{x-1}{3}+\frac{y}{2}+x\right)2x+\left(\frac{x}{3}+\frac{y-1}{2}+x\right)y\right]k(k-2)!\leq(x+y)k!.$$

Since $k = 2x + y$, this simplifies to:

$$(x+y)(2x+y-1) \geq \left(\frac{x-1}{3} + \frac{y}{2} + x\right) 2x + \left(\frac{x}{3} + \frac{y-1}{2} + x\right) y.$$

We can verify that this equilibrium condition holds if we set $y = \frac{2+\epsilon}{3}x$, with $\epsilon > 0$ arbitrarily small, and let $x \to \infty$. We then get a $\frac{2x+y}{x+y}$ lower bound on the POA, which is arbitrarily close to 1.6. This completes the proof of Lemma 6.

2.2 Upper Bound for Egalitarian Payoffs

▶ **Lemma 11.** *The* POA *of egalitarian profit sharing is at most* 1.6.

We will apply the (λ, μ) smoothness framework of Roughgarden [18]. For the purposes of this proof, we extend the game by introducing a new strategy for each player, which is to split herself into $|T|$ fractions of $\frac{1}{|T|}$ each, and assign each fraction to a different team; call this the *fractional* strategy. We define the payoff of a $\frac{1}{|T|}$-sized fractional player sharing a task s with another n (integral) players in a team as $\frac{1}{|T|} \cdot \frac{v_s}{n+1}$. If every player plays her fractional strategy, then we denote the outcome OPT-FR. Let N_s be the set of players who can perform task s. We define the utility of a task s in team t in the outcome OPT-FR as $v_s \cdot \min\left\{\frac{|N_s|}{|T|}, 1\right\}$.

We prove two important properties of this augmented game. The first property is that OPT-FR represents an optimal fractional solution to the optimization problem maximizing the total utility; therefore, its utility is at least that of OPT, which is the optimal integral solution to the same problem.

▶ **Property 12.** *The total utility (social welfare) of* OPT-FR *is at least that of* OPT.

This property allows to compare the utility in any NE with that in OPT-FR instead of OPT in order to obtain an upper bound on the POA. Since OPT-FR is highly symmetric, this is a simpler comparison that does not require delving into the structure of OPT.

The second property establishes that any NE in the original game is also an NE in the augmented game, i.e., no player has an incentive to deviate to her fractional strategy. This property holds because a deviation to the fractional strategy would produce payoff for the player that is a convex combination of her current payoff and the payoffs produced by deviating (integrally) to the other teams. Since none of these integral deviations produces a higher payoff, neither does the deviation to the fractional strategy.

▶ **Property 13.** *If* NASH *is an* NE *in the original game, then no player has an incentive to deviate to her fractional strategy.*

Let u_i^d be the payoff of player i if she unilaterally deviates from her team in NASH to her fractional strategy. Also, let U be the total utility (social welfare) in NASH and U^* be the total utility in OPT-FR. Our goal will be to identify positive parameters λ and μ such that for any equilibrium NASH,

$$\sum_{i \in N} u_i^d \geq \lambda U^* - \mu U. \tag{4}$$

Using Property 13, we have: $U = \sum_{i \in N} u_i \geq \sum_{i \in N} u_i^d \geq \lambda U^* - \mu U$. By rearranging the terms, we get $\frac{U^*}{U} \leq \frac{\mu+1}{\lambda}$. A POA bound of $\frac{\mu+1}{\lambda}$ now follows from Property 12. We will show Eq. (4) with $\lambda = \frac{5}{6}$ and $\mu = \frac{1}{3}$, which will then give us the desired upper bound of 1.6.

Our task, therefore, is to prove Eq. (4) with $\lambda = \frac{5}{6}$ and $\mu = \frac{1}{3}$. Let us initially focus on a single task s. Let n be the number of players who can perform task s, k be the number of teams, and h be the total utility produced by task s across all the teams in OPT-FR. To compare this utility with that in the equilibrium NASH, we use γ to denote the ratio of utilities for task s in the two assignment OPT-FR and NASH. In other words, γh denotes the total utility produced by task s in NASH. We examine two cases: $\gamma < \frac{1}{2}$ and $\gamma \geq \frac{1}{2}$. (Since we argue Eq. (4) for each task separately, we assume wlog that $v_s = 1$.)

Case 1: $\gamma < \frac{1}{2}$. When a player $i \in N_s$ deviates to her fractional strategy unilaterally, her payoff from task s is the sum of payoffs from the $k - \gamma h$ teams that do not cover task s and the γ_h teams that already cover s. This is given by:

$$\frac{1}{k}(k - \gamma h) + \frac{1}{k} \sum_{t:n_{s,t}>0} \frac{1}{n_{s,t}+1} \geq \frac{1}{k}(k - \gamma h) + \frac{1}{k} \cdot \gamma h \cdot \frac{1}{\frac{n}{\gamma h}+1} \quad \text{(by convexity)}. \tag{5}$$

Summing over the n players who can perform task s gives:

$$\frac{n}{k}(k - \gamma h) + \frac{n}{k} \cdot \frac{(\gamma h)^2}{n + \gamma h} \geq (1 - \gamma)h + \frac{(\gamma h)^2}{h + \gamma h}. \tag{6}$$

The inequality follows by replacing n and k with their smaller or equal number h, since the left-hand side is increasing as a function of n and k. We can then verify that for our values of $\lambda = \frac{5}{6}$ and $\mu = \frac{1}{3}$ and for any $\gamma < \frac{1}{2}$, the last expression from (6) satisfies,

$$(1 - \gamma)h + \frac{(\gamma h)^2}{h + \gamma h} \geq \lambda h - \mu \gamma h. \tag{7}$$

Case 2: $\gamma \geq \frac{1}{2}$. Similar to Case 1, the sum of payoffs for deviating from NASH to the fractional strategies is at least

$$\frac{n}{k}(k - \gamma h) + \frac{n}{k} \sum_{t:n_{s,t}>0} \frac{1}{n_{s,t}+1}. \tag{8}$$

Note that the sum of all $n_{s,t}$ values must be equal to n. Also, note that an adversary minimizing $\sum_{t:n_{s,t}>0} \frac{1}{n_{s,t}+1}$ sets all $n_{s,t}$ values equal and, if it turns out to be non-integral, then rounds some of them up and some down to keep their sum at n. Now consider the expression $n \sum_{t:n_{s,t}>0} \frac{1}{n_{s,t}+1}$ and suppose that the $n_{s,t}$ values have been picked by the adversary as above. Consider increasing n by one. Then the adversary will also increase exactly one of the $n_{s,t}$ values to restore the property that they sum to n. This will clearly increase the value of the expression $n \sum_{t:n_{s,t}>0} \frac{1}{n_{s,t}+1}$. Hence, we again get a lower bound on (8) by substituting n and k with their smaller number h, and letting the adversary pick $n_{s,t}$ values summing to h.

At this point, we know that the sum of $n_{s,t}$ values will be equal to h and that the number of $n_{s,t}$ variables is γh, with $\gamma \geq \frac{1}{2}$. Therefore, each $n_{s,t}$ value chosen by the adversary will be either 1 or 2. Since the average of the $n_{s,t}$ values must be equal to $\frac{1}{\gamma}$, there is also the constraint that $2\beta + 1(1 - \beta) = \frac{1}{\gamma}$, where β is the fraction of $n_{s,t}$ variables with value 2. After solving, we get $\beta = \frac{1}{\gamma} - 1$. Then, the inequality corresponding to (7) in Case 1 becomes

$$(1 - \gamma)h + \left(\frac{1}{\gamma} - 1\right)\gamma h \frac{1}{3} + \left(2 - \frac{1}{\gamma}\right)\gamma h \frac{1}{2} \geq \lambda h - \mu \gamma h, \tag{9}$$

which is always true for $\lambda = \frac{5}{6}$ and $\mu = \frac{1}{3}$.

Combining the above two cases, and summing over all tasks s, we can conclude that (4) holds as desired.

3 Marginal Gain Payoffs

In this section, we prove our upper bound from Theorem 2, i.e., show that the POA for marginal gain is at most $1 + \frac{1}{\sqrt{2}}$. We note that the lower bound of Theorem 2 can be established by hardness of approximation results (see [11]) assuming $\mathbf{P} \neq \mathbf{NP}$ but we also provide an explicit construction in our full paper.

▶ **Lemma 14.** *The* POA *of marginal gain is at most* $1 + \frac{1}{\sqrt{2}}$.

We will assume wlog that all tasks have unit utility, since any task can be decomposed into multiple unit-utility tasks. We say that player i has a *hit* on a task that she can perform if she receives payoff for it, i.e., is the first person in her team to perform the task; if not, we call it a *waste* of the task. We will write OPT for the optimal outcome and NASH for some given NE. Consider some task $s \in S$ and denote the number of hits on this task in OPT (resp., NASH) by h_s^* (resp., h_s) and the number of wastes by w_s^* (resp., w_s). Consider the quantity $w_s^+ = w_s - w_s^*$. For tasks that have $w_s^+ > 0$, we will arbitrarily select w_s^+ of the wastes in NASH and label them as the *additional* wastes of NASH against OPT. Note that since $h_s^* + w_s^* = h_s + w_s$, w_s^+ is precisely the difference in social utilities of NASH and OPT due to task s. For ease of exposition we make the following modification to the values of h_s^* and w_s^*: for tasks with $w_s^+ < 0$, we raise h_s^* (and accordingly lower w_s^*) until $w_s^+ = 0$. These changes improve the situation for OPT, and so an upper bound after the modification also holds for the original scenario.

In what follows, let $k = |T|$ be the number of teams and $m = |S|$ be the number of tasks. Now focus on any of the additional wastes w, which was a waste of s by player i in team t. We can charge this waste to k hits as follows:

1. Task s was already covered in team t when i appeared; hence, we can infer that a hit occurred for task s in team t in a previous arrival. We charge to that hit.
2. For every team $t' \neq t$:
   a. either has s covered (a hit from a previous arrival),
   b. or has some other task s covered, for which i received payoff in t (again a hit from a previous arrival). If not, player i would have chosen t' over t.

   We charge to these hits in teams $t' \neq t$.

We now need to bound the maximum number of times that a hit can be charged in the above scheme. Whenever some hit h for some task s is charged for an additional waste w, one of the following is true:

- Another hit h', on the same task s as h, is happening at the same time as w. This is true for charging arguments of the form (2b).
- w is a waste of the same task s that is a hit in h. This is true for charging arguments of the form (1) and (2a).

Hence, a hit h on task s may be charged in the above scheme only if, at the time of the charging, there is a hit h' on the same task s or a waste w of the same task s. We also note that if a player i incurs multiple wastes in her selected team t, then for each team $t' \neq t$ and for each of these wastes (that are labeled as additional), we can find a distinct hit to charge with an argument of the form (2a) or (2b).

It follows that the first hit on s can be charged at most $h_s - 1 + w_s^+ = h_s^* - 1$ times, the second hit on s can be charged at most $h_s^* - 2$ times, and so on. Recall that the total number of hits on task s in NASH is h_s. Therefore, the total number of times that a hit on s can be

charged, denoted χ_s, is upper bounded as

$$\chi_s \leq (h_s^* - 1) + (h_s^* - 2) + (h_s^* - 3) + \ldots + (h_s^* - h_s) = \sum_{j=1}^{h_s} (h_s^* - j)\,.$$

Then, it follows that the total number of times all hits are charged is upper bounded as follows, with U (resp., U^*) denoting the total utility of NASH (resp., OPT).

$$\begin{aligned}
\sum_{s\in S}\chi_s &\leq \sum_{s\in S}\sum_{j=1}^{h_s}(h_s^* - j) = \sum_{s\in S} h_s h_s^* - \sum_{s\in S}\left(\frac{1}{2}h_s\,(h_s+1)\right) = \sum_{s\in S} h_s h_s^* - \frac{1}{2}\sum_{s\in S} h_s^2 - \frac{U}{2} \\
&\leq \sqrt{\sum_{s\in S} h_s^2}\sqrt{\sum_{s\in S} {h_s^*}^2} - \frac{1}{2}\sum_{s\in S} h_s^2 - \frac{U}{2} \quad \text{(Cauchy-Schwarz inequality)} \\
&\leq \sqrt{m}\frac{U}{m}\cdot\sqrt{m}\frac{U^*}{m} - \frac{1}{2}m\cdot\frac{U^2}{m^2} - \frac{U}{2} = \frac{U\cdot U^*}{m} - \frac{U^2}{2m} - \frac{U}{2}. \qquad (10)
\end{aligned}$$

Eqn. (10) follows from the following facts: (a) the sum of squares of m nonnegative numbers with a given sum (here the sum of all h_s is U and the sum of all h_s^* is U^*) is minimized when they are all equal, and (b) the expression is decreasing as a function of the sum of all h_s and as a function of the sum of all h_s^*.

We also know that the total number of times a hit is charged is k times the number of additional wastes. Hence,

$$\sum_{s\in S}\chi_s = k\sum_{s\in S} w_s^+ = k\sum_{s\in S}(w_s - w_s^*) = k\sum_{s\in S}(h_s^* - h_s) = k(U^* - U). \qquad (11)$$

From (10) and (11) we get that:

$$\frac{U\cdot U^*}{m} - \frac{U^2}{2m} - \frac{U}{2} \geq k(U^* - U) \qquad (12)$$

Now let $\gamma = \frac{U}{U^*}$. Note that upper bounding $\frac{1}{\gamma}$ gives an upper bound on the POA. Substituting in (12) and using the fact that $U^* \leq mk$, we get

$$-\frac{k}{2}\gamma^2 + \frac{4k-1}{2}\gamma - k \geq 0$$

Since, by definition, $\gamma \in [0,1]$, the expression is increasing in γ and, hence, for the inequality to hold, it must be the case that γ is greater than or equal to the unique root in $[0,1]$. This gives the following upper bound for the POA:

$$\frac{U^*}{U} \leq \frac{2k}{4k - 1 - \sqrt{8k^2 - 8k + 1}}.$$

This is increasing in k and as k goes to ∞, the limit is $1 + \frac{1}{\sqrt{2}}$. This completes the proof.

4 Price of Stability

Studying the efficiency of the best NE in our setting is an interesting direction. The more optimistic metric that corresponds to the price of anarchy in this framework is the *price of stability*, i.e., the worst case ratio of the efficiency in OPT over the efficiency in the best NE. We conclude the paper with a brief discussion on the topic. For marginal loss profit sharing, we observe that any beneficial unilateral deviation also improves the social objective,

hence, OPT is also a NE and the price of stability is 1. For marginal gain profit sharing, it is possible to take any given NE, NASH, and modify the instance so that NASH becomes the unique NE in the modified instance. The modification is performed by means of making a very large number of copies of each task and introducing new unit tasks for tie-breaking purposes. Then, we get that the price of stability for marginal loss profit sharing is equal to the price of anarchy. We omit the exact details of this modification process. In contrast to the two previous models, determining the price of stability for egalitarian profit sharing appears to be a challenging question that invites future research.

References

1 Y. Bachrach, V. Syrgkanis, and M. Vojnovic. Incentives and efficiency in uncertain collaborative environments. In *WINE*, pages 26–39, 2013.

2 G. Calinescu, C. Chekuri, M. Pal, and J. Vondrak. Maximizing a submodular set function subject to a matroid constraint (extended abstract). In *IPCO*, pages 182–196, 2007.

3 H. Chen, T. Roughgarden, and G. Valiant. Designing network protocols for good equilibria. *SIAM Journal on Computing*, 39(5):1799–1832, 2010.

4 S. Dobzinski and M. Schapira. An improved approximation algorithm for combinatorial auctions with submodular bidders. In *SODA*, pages 1064–1073, 2006.

5 Y. Filmus and J. Ward. The Power of Local Search: Maximum Coverage over a Matroid. In *STACS*, pages 601–612, 2012.

6 Y. Filmus and J. Ward. Monotone submodular maximization over a matroid via non-oblivious local search. *SIAM Journal on Computing*, 43(2):514–542, 2014.

7 M. Gairing. Covering games: Approximation through non-cooperation. In *WINE*, pages 184–195, 2009.

8 V. Gkatzelis, K. Kollias, and T. Roughgarden. Optimal cost-sharing in weighted congestion games. In *WINE*, 2014.

9 T. Harks and K. Miller. The worst-case efficiency of cost sharing methods in resource allocation games. *Operations Research*, 59(6):1491–1503, 2011.

10 E. Kalai and D. Samet. On weighted Shapley values. *International Journal of Game Theory*, 16(3):205–222, 1987.

11 S. Khot, R. J. Lipton, E. Markakis, and A. Mehta. Inapproximability results for combinatorial auctions with submodular utility functions. *Algorithmica*, 52(1):3–18, 2008.

12 K. Kollias and T. Roughgarden. Restoring pure equilibria to weighted congestion games. In *ICALP*, 2011.

13 J. R. Marden and T. Roughgarden. Generalized efficiency bounds in distributed resource allocation. In *CDC*, pages 2233–2238. IEEE, 2010.

14 J. R. Marden and A. Wierman. Distributed welfare games. *Operations Research*, 61(1):155–168, 2013.

15 D. Monderer and L. S. Shapley. Potential games. *Games and Economic Behavior*, 14(1):124–143, 1996.

16 I. Post, M. Kapralov, and J. Vondrak. Online submodular welfare maximization: Greedy is optimal. In *SODA*, pages 1216–1225, 2013.

17 R. W. Rosenthal. A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory*, 2(1):65–67, 1973.

18 T. Roughgarden. Intrinsic robustness of the price of anarchy. In *STOC*, pages 513–522, 2009.

19 L. S. Shapley. *Additive and Non-Additive Set Functions*. PhD thesis, Department of Mathematics, Princeton University, 1953.

20 A. Vetta. Nash equilibria in competitive societies, with applications to facility location, traffic routing and auctions. In *FOCS*, 2002.

Improved Guarantees for Vertex Sparsification in Planar Graphs*†

Gramoz Goranci‡[1], Monika Henzinger[2], and Pan Peng[3]

1 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
gramoz.goranci@univie.ac.at
2 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
monika.henzinger@univie.ac.at
3 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
pan.peng@univie.ac.at

Abstract

Graph Sparsification aims at compressing large graphs into smaller ones while (approximately) preserving important characteristics of the input graph. In this work we study Vertex Sparsifiers, i.e., sparsifiers whose goal is to reduce the number of vertices. Given a weighted graph $G = (V, E)$, and a terminal set K with $|K| = k$, a quality-q vertex cut sparsifier of G is a graph H with $K \subset V_H$ that preserves the value of minimum cuts separating any bipartition of K, up to a factor of q. We show that planar graphs with all the k terminals lying on the same face admit quality-1 vertex cut sparsifier of size $O(k^2)$ that are also planar. Our result extends to vertex flow and distance sparsifiers. It improves the previous best known bound of $O(k^2 2^{2k})$ for cut and flow sparsifiers by an exponential factor, and matches an $\Omega(k^2)$ lower-bound for this class of graphs.

We also study vertex reachability sparsifiers for directed graphs. Given a digraph $G = (V, E)$ and a terminal set K, a vertex reachability sparsifier of G is a digraph $H = (V_H, E_H)$, $K \subset V_H$ that preserves all reachability information among terminal pairs. We introduce the notion of reachability-preserving minors, i.e., we require H to be a minor of G. Among others, for general planar digraphs, we construct reachability-preserving minors of size $O(k^2 \log^2 k)$. We complement our upper-bound by showing that there exists an infinite family of acyclic planar digraphs such that any reachability-preserving minor must have $\Omega(k^2)$ vertices.

1998 ACM Subject Classification G.2.2 Graph Theory, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Vertex Sparsification, Graph Sparsification, Planar Graphs, Metric Embedding, Reachability

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.44

1 Introduction

Very large graphs or networks are ubiquitous nowadays, from social networks to information networks. One natural and effective way of processing and analyzing such graphs is to compress or sparsify the graph into a smaller one that well preserves certain properties of the original graph. Such a sparsification can be obtained by reducing the number of

* A full version of the paper is available at https://arxiv.org/abs/1702.01136.
† The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.
‡ Partially supported by the Doctoral Programme "Vienna Graduate School on Computational Optimization" which is funded by Austrian Science Fund (FWF, project no. W1260-N35)

edges. Typical examples include cut sparsifiers [3], spectral sparsifiers [36], spanners [40] and transitive reductions [1], which are subgraphs defined on the same vertex set of the original graph G while having much smaller number of edges and still well preserving the cut structure, spectral properties, pairwise distances, transitive closure of G, respectively. Another way of performing sparsification is by reducing the number of *vertices*, which is most appealing when only the properties among a subset of vertices (which are called *terminals*) are of interest (see e.g., [34, 2, 26]). We call such small graphs *vertex sparsifiers* of the original graph. In this version of the paper, we will focus on vertex cut and reachability sparsifiers.

More specifically, given a capacitated undirected graph $G = (V, E, c)$, and a set of terminals K, we are looking for a graph $H = (V_H, E_H, c_H)$ with as few vertices as possible and $K \subseteq V_H$ such that the properties (e.g., reachability) or quantities (e.g., cut value, multi-commodity flow, distance) among vertices in K in H are the same as or close to the corresponding properties or quantities in G. If $|K| = k$, we call the graph G a *k-terminal graph*. We say H is a *quality-q (vertex) cut sparsifier* of G, if for every bipartition $(U, K \setminus U)$ of the terminal set K, the value of the minimum cut separating U from $K \setminus U$ in G is within a factor of q of the value of minimum cut separating U from $K \setminus U$ in H. If H is a quality-1 cut sparsifier, then it will be also called a *mimicking network* [22]. Similarly, we define vertex flow and distance sparsifiers that (approximately) preserve multicommodity flows and distances among terminal pairs, respectively (formal definitions are deferred to the full version). These type of sparsifiers have proven useful in approximation algorithms [34] and also find applications in network routing [12].

Vertex reachability sparsifiers in *directed* graphs is another important and fundamental notion in Graph Sparsification, which has been implicitly studied in the dynamic graph algorithms community [37, 15], and explicitly in [24]. Specifically, given a digraph $G = (V, E)$, $K \subset V$, a digraph $H = (V_H, E_H)$, $K \subset V_H$ is a *vertex reachability sparsifier* of G if for any $x, x' \in K$, there is a directed path from x to x' in H iff there is a directed path from x to x' in G. Note that any k-terminal digraph G always admits a trivial vertex reachability sparsifier H, which corresponds to the transitive closure restricted to the terminals. In this work, we initiate the study of *reachability-preserving minors*, i.e., vertex reachability sparsifiers with H required to be a minor of G. The restriction on H being a minor of G is desirable as it makes sure that H is structurally similar to G, e.g., any minor of a planar graph remains planar. We ask the question whether general graphs admit reachability-preserving minors whose size can be bounded independently of the input graph G, and study it from both the lower- and upper-bound perspective.

Our Results. We provide new constructions for quality-1 (exact) cut, flow and distance sparsifiers for k-terminal planar graphs, where all the terminals are assumed to lie on the same face. We call such k-terminal planar graphs *Okamura-Seymour (OS) instances*. They are of particular interest in the algorithm design and optimization community, due to the classical Okamura-Seymour theorem that characterizes the existence of feasible concurrent flows in such graphs (see e.g., [35, 8, 9, 30]).

We show that the size of quality-1 sparsifiers can be as small as $O(k^2)$ for such instances, for which only exponential (in k) size of cut and flow sparisifiers were known before [27, 2]. Formally, we have the following theorem.

▶ **Theorem 1.** *For any OS instance G, i.e., a k-terminal planar graph in which all terminals lie on the same face, there exist quality-1 vertex cut, flow and distance sparsifers of size $O(k^2)$. Furthermore, the resulting sparsifiers are also planar.*

We remark that all the above sparsifiers can be constructed in polynomial time (in n and k), but we will not optimize the running time here. As we mentioned above, previously the only known upper bound on the size of quality-1 cut and flow sparsifiers for OS instance was $O(k^2 2^{2k})$, given by [27, 2]. Our upper bound for cut sparsifier also matches the lower bound of $\Omega(k^2)$ for OS instance given by [27]. More specifically, in [27], an OS instance (that is a grid in which all terminals lie on the boundary) is constructed, and used to show that any mimicking network for this instance needs $\Omega(k^2)$ edges, which is thus a lower bound for planar graphs (see the table below for an overview). Note that that even though our distance sparsifier is not necessarily a minor of the original graph G, it still shares the nice property of being planar as G. It is worth mentioning that in [29], it is proven that there exists a k-terminal planar graph G (not necessarily an OS instance), such that any quality-1 distance sparsifier of G that is planar requires at least $\Omega(k^2)$ vertices.

Type of sparsifier	Graph family	Upper Bound	Lower Bound
Cut	Planar	$O(k^2 2^{2k})$ [27]	
Cut	Planar OS	$O(k^2)$ [**new**]	$\lvert E(G')\rvert \geq \Omega(k^2)$ [27]
Flow	Planar OS	$O(k^2 2^{2k})$ [2]	follows from cut
Flow	Planar OS	$O(k^2)$ [**new**]	follows from cut
Distance (minor)	Planar OS	$O(k^4)$ [26]	$\Omega(k^2)$ [26]
Distance (planar)	Planar OS	$O(k^2)$ [**new**]	

Our second main contribution is the study of reachability-preserving minors. Although reachability is a weaker requirement in comparison to shortest path distances, directed graphs are usually much more cumbersome to deal with from the perspective of graph sparsification. Surprisingly, we show that general digraphs admit reachability-preserving minors with $O(k^4)$ vertices (see Corollary 20), thus matching the bound of Krauthgamer et al. [26] for distances in undirected graphs. A tight integration of our techniques with the compact distance oracles for planar graphs by Thorup [39] yields the following theorem.

▶ **Theorem 2.** *Given a k-terminal planar digraph, there exists a reachability-preserving minor H of G with size $O(k^2 \log^2 k)$.*

We complement the above result by showing that there exist instances where the above upper-bound is tight up to a $O(\log^2 k)$ factor. The proof is deferred to the full version.

▶ **Theorem 3.** *For infinitely many $k \in \mathbb{N}$ there exists a k-terminal acyclic directed grid G such that any reachability-preserving minor of G must use $\Omega(k^2)$ non-terminals.*

Our third contribution is a lower bound on the size of any *data structure* (not necessarily a graph) that approximately preserves pairwise terminal distances of *general* k-terminal graphs, which provides a trade-off between the distance stretch and the space complexity. The proof is deferred to the full version.

▶ **Theorem 4.** *For any $\varepsilon > 0$ and $t \geq 2$, there exists a (sparse) k-terminal n-vertex graph such that $k = o(n)$, and any data structure that approximates pairwise terminal distances within a multiplicative factor of $t - \varepsilon$ or an additive error $2t - 3$ must use $\Omega(k^{1+1/(t-1)})$ bits.*

▶ Remark. Recently and independently of our work, Krauthgamer and Rika [28] constructed quality-1 cut sparsifiers of size $O(\gamma^5 2^{2\gamma} k^4)$ for planar graphs whose terminals are incident to at most $\gamma = \gamma(G)$ faces. In comparison with our upper-bound which only considers the case $\gamma = 1$, the size of our sparsifiers from Theorem 1 is better by a $\Omega(k^2)$ factor.

Our Techniques. We construct our quality-1 cut and distance sparsifiers by repeatedly performing *Wye-Delta transformations*, which are local operations that preserve cut values and distances and have proven very powerful in analyzing electrical networks and in the theory of circular planar graphs (see e.g., [14, 18]). Khan and Raghavendra [25] used Wye-Delta transformations to construct quality-1 cut sparsifiers of size $O(k)$ for trees, while our case (i.e., the planar OS instances) is more general and complicated and previously it was not clear at all how to apply such transformations to a more broad class of graphs.

Our approach is as follows. Given a k-terminal planar graph with terminals lying on the same face, we first embed it into some large grid with terminals lying on the boundary of the grid. Next, we show how to embed this grid into a "more suitable" graph, which we will refer to as "half-grid". Finally, using the Wye-Delta operations, we reduce the "half-grid" into another graph whose number of vertices can be bounded by $O(k^2)$. Since we argue that the above graph reductions preserve exactly all terminal minimum cuts, our result follows. Gitler [19] proposed a similar approach for studying the reducibility of multi-terminal graphs with the goal to classify all Wye-Delta reducible graphs, which is very different from our motivation of constructing small vertex sparsifiers with good quality.

The distance sparsifiers can be constructed similarly by slightly modifying the Wye-Delta operation. Our flow sparsifiers follow from the construction of cut sparsifiers and the flow/cut gaps for OS instances (which has also been observed by Andoni et al. [2]).

The results for reachability-preserving minors are obtained by exploiting the technique of Coppersmith and Elkin [13] on counting "branching" events between shortest paths in the directed setting (this technique has also been recently leveraged by Bodwin [4]). We then combine our construction with the compact reachability oracle for planar graphs by Thorup [39], to show our upper-bound for planar graphs. The lower-bound follows by adapting the ideas of Krauthgamer et al. [26] from their lower-bound proof on distance-preserving minors for undirected graphs.

Our lower bound of the space complexity of any compression function approximately preserving terminal pairwise distance is derived by combining extremal combinatorics construction of Steiner Triple System that was used to prove lower bounds on the size of distance approximating minors (see [10]) and the incompressibility technique from [33].

Related Work. There has been a long line of work on investigating the tradeoff between the quality of the vertex sparsifier and its size (see e.g., [17, 27, 2]). (Throughout, cut, flow and distance sparsifiers will refer to their vertex versions.) Quality-1 *cut sparsifiers* (or equivalently, mimicking networks) were first introduced by Hagerup et al. [22], who proved that for any graph G, there always exists a mimicking network of size $O(2^{2^k})$. Krauthgamer and Rika [27] showed how to build a mimicking network of size $O(k^2 2^{2k})$ for any planar graph G that is minor of the input graph. They also proved a lower bound of $\Omega(k^2)$ on the number of edges of the mimicking network of planar graphs, and a lower bound of $2^{\Omega(k)}$ on the number of vertices of the mimicking network for general graphs.

Quality-1 vertex flow sparsifiers have been studied in [2, 20], albeit only for restricted families of graphs like quasi-bipartite, series-parallel, etc. It is not known if any general undirected graph G admits a constant quality flow sparsifier with size independent of $|V(G)|$ and the edge capacities. For the quality 1 distance sparsifiers, Krauthgamer, Nguyen and Zondiner [26] introduced the notion of *distance-preserving minors*, and showed an upper-bound of size $O(k^4)$ for general undirected graphs. They also gave a lower bound of $\Omega(k^2)$ on the size of such a minor for planar graphs. Over the last two decades, there has been a considerable amount of work on understanding the tradeoff between the sparsifier's quality q and its size for $q > 1$, i.e., when the sparsifiers only *approximately* preserve the corresponding properties [11, 2, 34, 31, 6, 17, 32, 21, 7, 5, 17, 23, 10, 16].

2 Preliminaries

Let $G = (V, E, c)$ be an undirected graph with terminal set $K \subset V$ of cardinality k, where $c : E \to \mathbb{R}_{\geq 0}$ assigns a non-negative capacity to each edge. We will refer to such a graph as a *k-terminal graph*. Throughout the paper we will be dealing with two special types of graphs.

A *grid* graph is a graph with $n \times n$ vertices $\{(u, v) : u, v = 1, \ldots, n\}$, where (u, v) and (u', v') are adjacent if $|u' - u| + |v' - v| = 1$. For $k < n$, a *half-grid* graph with k terminals is a graph $T_k^n = (V, E)$ with $K \subset V$ and $n(n+1)/2$ vertices $\{(i, j) : i \leq j \text{ and } i, j = 1, \ldots, n\}$, where (i, j) and (i', j') are connected by an edge if $|i' - i| + |j' - j| = 1$, and additional diagonal edges between (i, i) and $(i + 1, i + 1)$ for $i = 1, \ldots, n - 1$. Moreover, each terminal vertex in T_k^n must be one of its diagonal vertices, i.e., every $x \in K$ is of the form (m, m) for some $m \in \{1, \ldots, n\}$. Let $\hat{T}_k^n$ be the same graph as T_k^n but excluding the diagonal edges.

Let $U \subset V$ and $S \subset K$. We say that a cut $(U, V \setminus U)$ is S-separating if it separates the terminal subset S from its complement $K \setminus S$, i.e., $U \cap K$ is either S or $K \setminus S$. We will refer to such cut as a *terminal cut*. The cutset $\delta(U)$ of a cut $(U, V \setminus U)$ represents the edges that have one endpoint in U and the other one in $V \setminus U$. The cost $\text{cap}_G(\delta(U))$ of a cut $(U, V \setminus U)$ is the sum over all capacities of the edges belonging to the cutset. We let $\text{mincut}_G(S, K \setminus S)$ denote the minimum cost of any S-separating cut of G. A graph $H = (V_H, E_H, c_H)$, $K \subset V_H$ is a *vertex cut sparsifier* of G with *quality* $q \geq 1$ if for any $S \subset K$, $\text{mincut}_G(S, K \setminus S) \leq \text{mincut}_H(S, K \setminus S) \leq q \cdot \text{mincut}_G(S, K \setminus S)$.

Let $G = (V, E)$ be a directed graph with terminal set $K \subset V$, $|K| = k$, which we will refer to as a *k-terminal digraph*. We say G is a *k-terminal* DAG if G has no directed cycles. The *in-degree* of a vertex v, denoted by $\deg_G^-(v)$, is the number of edges directed towards v in G. A digraph $H = (V_H, E_H)$, $K \subset V_H$ is a *vertex reachability sparsifier* of G if for any $x, x' \in K$, there is a directed path from x to x' in H iff there is a directed path from x to x' in G. If H is obtained by performing minor operations in G, then we say that H is a *reachability-preserving minor* of G. We define the *size* of H to be the number of non-terminals in H, i.e. $|V_H \setminus K|$.

Wye-Delta Transformations. In this section we investigate the applicability of some graph reduction techniques that aim at reducing the number of non-terminals in a k-terminal graph. We start by reviewing the so-called *Wye-Delta* operations in graph reductions. These operations consist of five basic rules, which we describe below.

1. *Degree-one reduction:* Delete a degree-one non-terminal and its incident edge.
2. *Series reduction:* Delete a degree-two non-terminal y and its incident edges (x, y) and (y, z), and add a new edge (x, z) of capacity $\min\{c(x, y), c(y, z)\}$.
3. *Parallel reduction:* Replace all parallel edges by a single edge whose capacity is the sum over all capacities of parallel edges.
4. *Wye-Delta transformation:* Let x be a degree-three non-terminal with neighbours $\delta(x) = \{u, v, w\}$. Assume w.l.o.g.[1] that for any pair $(u, v) \in \delta(x)$, $c(u, x) + c(v, x) \geq c(w, x)$, where $w \in \delta(v) \setminus \{u, v\}$. Then we can delete x (along with all its incident edges) and add edges $(u, v), (v, w)$ and (w, u) with capacities $(c(u, x) + c(v, x) - c(w, x))/2$, $(c(v, x) + c(w, x) - c(u, x))/2$ and $(c(u, x) + c(w, x) - c(v, x))/2$, respectively.

[1] Suppose there exist a pair $(u, v) \in \delta(x)$ with $c(u, x) + c(v, x) < c(w, x)$, where $w \in \delta(v) \setminus \{u, v\}$. Then we can simply set $c(w, x) = c(u, x) + c(v, x)$, since any terminal minimum cut would cut the edges (u, x) and (v, x) instead of the edge (w, x).

5. *Delta-Wye transformation:* Delete the edges of a triangle connecting x, y and z, introduce a new non-terminal vertex w and add new edges (w, x), (w, y) and (w, z) with edge capacities $c(x, y) + c(x, z)$, $c(x, y) + c(y, z)$ and $c(x, z) + c(y, z)$ respectively.

The following lemma (which follows from the above definitions) shows that the above rules preserve exactly all terminal minimum cuts.

▶ **Lemma 5.** *Let G be a k-terminal graph and G' be a k-terminal graph obtained from G by applying one of the rules $1 - 5$. Then G' is a quality 1-vertex cut sparsifier of G.*

For our application, it will be useful to enrich the set of rules by introducing two new operations. These operations can be realized as series of the operations 1-5.

6. *Edge deletion (with vertex x):* For a degree-three non-terminal with neighbours u, v, the edge (u, v) can be deleted, if it exists. To achieve this, we use a Delta-Wye transformation followed by a series reduction.
7. *Edge replacement:* For a degree-four non-terminal vertex with neighbours x, u, v, w, if the edge (x, u) exists, then it can be replaced by the edge (v, w). To achieve this, we use a Delta-Wye transformation followed by a Wye-Delta transformation.

A k-terminal graph G is *Wye-Delta* reducible to another k-terminal graph H, if G is reduced to H by repeatedly applying one of the operations 1-7. We obtain the following lemma, whose proof we defer to the full version.

▶ **Lemma 6.** *Let G and H be k-terminal graphs. Moreover, let G be Wye-Delta reducible to H. Then H is a quality 1-vertex cut sparsifier of G.*

Graph Embeddings. Throughout this paper, we will be dealing with the embedding of a planar graph into a square *grid* graph. One way of drawing graphs in the plane are *orthogonal grid-embeddings* [41]. In such a setting, the vertices correspond to distinct points and edges consist of alternating sequences of vertical and horizontal segments. Equivalently, one can view this as drawing our input graph as a subgraph of some grid. Formally, a *node-embedding* ρ of $G_1 = (V_1, E_1)$ into $G_2 = (V_2, E_2)$ is an injective mapping that maps V_1 into V_2, and E_1 into paths in G_2, i.e., (u, v) maps to a path from $\rho(u)$ to $\rho(v)$, such that every pair of paths that correspond to two different edges in G_1 is vertex-disjoint (except possibly at the endpoints). If G_2 is a planar graph, then $\rho(G_1)$ and G_1 are also planar. Thus, if G_1 and G_2 are planar we then refer to ρ as an *orthogonal embedding*. Moreover, given a planar graph G_1 drawn in the plane, the embedding ρ is called *region-preserving* if $\rho(G_1)$ and G_1 have the same planar topological embedding.

Let G_1 be a k-terminal graph. Since the embedding does not affect the vertices of G_1, the terminals of G_1 are also terminals in $\rho(G_1)$. Although the embedding does not consider capacity of the edges in G_1, we can still guarantee that such an embedding preserves all terminal minimum cuts, for which we make use of the following operation:

1. *Edge subdivision:* Let (u, v) be an edge of capacity $c(u, v)$. Delete (u, v), introduce a new vertex w and add edges (u, w) and (w, v), each of capacity $c(u, v)$.

The proof of the following Lemma is deferred to the full version.

▶ **Lemma 7.** *Let ρ be a node-embedding and let G_1 and $\rho(G_1)$ be k-terminal graphs defined as above. Then $\rho(G_1)$ preserves exactly all terminal minimum cuts of G.*

3 An Exact Vertex Cut Sparsifier of Size $O(k^2)$

In this section we show that given a k-terminal planar graph, where all terminals lie on the same face, one can construct a quality-1 vertex cut sparsifier of size $O(k^2)$. Note that it suffices to consider the case when all terminals lie on the *outer* face.

Embedding into Grids. It is well-known that one can obtain an orthogonal embedding of a planar graph with maximum-degree at most three into a grid (see Valiant [41]). However, our input planar graph can have arbitrarily large maximum degree. In order to be able to make use of such an embedding, we need to first reduce our input graph to a bounded-degree graph while preserving planarity and all terminal minimum cuts. We achieve this by making use of a *vertex splitting* technique, which we describe below.

Given a k-terminal planar graph $G' = (V', E', c')$ with $K \subset V'$ lying on the outer face, vertex splitting produces a k-terminal planar graph $G = (V, E, c)$ with $K \subset V$ such that the maximum degree of G is at most three. Specifically, for each vertex v of degree $d > 3$ with neighboring vertices $u_1, \ldots, u_d$, we delete v and introduce new vertices $v_1, \ldots, v_d$ along with edges $\{(v_i, v_{i+1}) : i = 1, \ldots, d-1\}$, each of capacity $C + 1$, where $C = \sum_{e \in E'} c'(e)$. Further, we replace the edges $\{(u_i, v) : i = 1, \ldots, d\}$ with $\{(u_i, v_i) : i = 1, \ldots, d\}$, each of corresponding capacity. If v is a terminal vertex, we set one of the v_i's to be a terminal vertex. It follows that the resulting graph G is planar and terminals can be still embedded on the outer face. Note that while the degree of every vertex v_i is at most 3, the degree of any other vertex is not affected. The proof of the claim below is deferred to the full version.

▶ **Claim 8.** *Let G' and G be k-terminal graphs defined as above. Then G preserves exactly all minimum terminal cuts of G', i.e., G is a quality-1 cut sparsifier of G'.*

Let $G = (V, E)$ be a k-terminal graph obtained by vertex splitting of all vertices of degree larger than 3 of $G' = (V', E')$. Further, let $n' = |V'|$, $m' = |E'|$, $n = |V|$ and $m = |E|$. Then it is easy to show that $n \leq 2m'$ and $m \leq m' + n \leq 3m'$. Since G' is planar, we have that $n = O(n')$ and $m = O(n')$. Thus, by just a linear blow-up on the size of vertex and edge sets, we may assume w.l.o.g. that our input graph is a planar graph of degree at most three.

Valiant [41] and Tamassia et al. [38] showed that a k-terminal planar graph G with n vertices and degree at most three admits an orthogonal region-preserving embedding into some square grid of size $O(n) \times O(n)$. By Lemma 7, we know that the resulting graph exactly preserves all terminal minimum cuts of G. We remark that since the embedding is region-preserving, the outer face of the input graph is embedded to the outer face of the grid. Therefore, all terminals in the embedded graph lie on the outer face of the grid. Performing appropriate edge subdivisions, we can make all the terminals lie on the boundary of some possibly larger grid. Further, we can add dummy non-terminals and zero edge capacities to transform our graph into a full-grid H. We observe that the latter does not affect any terminal min-cut. The above leads to the following:

▶ **Lemma 9.** *Given a k-terminal planar graph G, where all terminals lie on the outer face, there exists a k-terminal grid graph H, where all terminals lie on the boundary such that H preserves exactly all terminal minimum cuts of G. The resulting graph has $O(n^2)$ vertices and edges.*

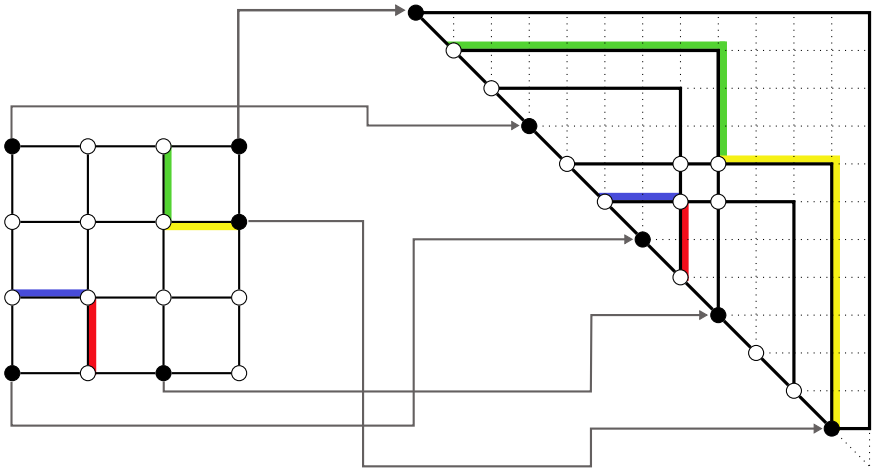


Figure 1 Embedding grid into half-grid. Black vertices represent terminals while white vertices represent non-terminals. The counter-clockwise ordering starts at the top right terminal. Coloured edges and paths correspond to the mapping of the respective edges: blue for edges $((i,1),(i,2))$, red for edges $((n-1,j),(n,j))$, green for edges $((1,j),(2,j))$ and yellow for edges $((i,n-1),(i,n))$, where $i,j=2,\ldots,n-1$.

Embedding Grids into Half-Grids. Next, we show how to embed square grids into half-grid graphs (see Section 2), which will facilitate the application of Wye-Delta transformations. The existence of such an embedding was claimed in the thesis of Gitler [19], but no details on its construction were given.

Let G be a k-terminal square grid on $n \times n$ vertices where terminals lie on the boundary of the grid. We obtain the following:

▶ **Lemma 10.** *There exists a node embedding of the grid G into T_k^ℓ, where $\ell = 4n-3$.*

Proof. Our construction works as follows (See Fig. 1 for an example). We first fix an ordering on the vertices lying on the boundary of the grid in the order induced by the grid. Then we embed each vertex according to that order into the diagonal vertices of the half-grid, along with the edges that form the boundary of the grid. The sub-grid obtained by removing all boundary vertices is embedded appropriately into the upper-part of the half-grid. Finally, we show how to embed edges between the boundary and the sub-grid vertices and argue that such an embedding is indeed vertex-disjoint for any pair of paths.

We start with the embedding of the vertices of G. Let us first consider the boundary vertices. The ordering imposed on these vertices can be viewed as starting with the upper-right vertex $(1,n)$ and visiting the rest of vertices in a counter-clockwise direction until reaching the vertex $(2,n)$. We map the vertices on the boundary as follows.

1. The vertex $(1,j)$ is mapped to the vertex $(n-j+1, n-j+1)$ for $j=2,\ldots,n$,
2. The vertex $(i,1)$ is mapped to the vertex $(n+i-1, n+i-1)$ for $i=1,\ldots,n-1$,
3. The vertex (n,j) is mapped to the vertex $(2n+j-2, 2n+j-2)$ for $j=1,\ldots,n-1$,
4. The vertex (i,n) is mapped to the vertex $(4n-i-2, 4n-i-2)$ for $i=2,\ldots,n$.

Now we consider the vertices that belong to the induced sub-grid S of G of size $(n-2)^2$ when removing the boundary vertices of our input grid. We map the vertex (i,j) to the vertex $(n+i-1, 2n+j-2)$ for $i,j=2,\ldots,n-1$. In other words, for every vertex of S we make a vertical shift by $n-1$ units and an horizontal shift by $2n-2$ units. By construction, it is not hard to check that every vertex of G is mapped to a different vertex of T_k^ℓ and all terminal vertices lie on the diagonal of T_k^ℓ.

We continue with the embedding of the edges of G. First, every edge between two boundary vertices in G is embedded to the edge between the corresponding mapped diagonal vertices of T_k^ℓ, except the edge between $(1,n)$ and $(2,n)$. For this edge, we define an edge embedding between the corresponding vertices $(1,1)$ and $(4n-4,4n-4)$ of T_k^ℓ by using the path:

$$(1,1) \to (1,2) \to \ldots \to (1,4n-3) \to (2,4n-3) \to \ldots \to (4n-4,4n-3) \to (4n-4,4n-4).$$

Next, every edge of the sub-grid S is embedded in to the edge connecting the mapped endpoints of that edge in T_k^ℓ. In other words, if (i,j) and (i',j') were connected by an edge e in S, then $(n+i-1,2n+j-2)$ and $(n+i'-1,2n+j'-2)$ are connected by an edge e' in T_k^ℓ and e is mapped to e'. Finally, the only edges that remain are those connecting a boundary vertex of G with a boundary vertex of S. We distinguish four cases depending on the edge position.

1. The edge $((i,2),(i,1))$ is mapped to the horizontal path given by:
$$(n+i-1,2n) \to (n+i-1,2n-1) \to \ldots \to (n+i-1,n+i-1) \text{ for } i=2,\ldots,n-1.$$
2. The edge $((n-1,j),(n,j))$ is mapped to the vertical path given by:
$$(2n-2,2n+j-2) \to (2n-1,2n+j-2) \to \ldots \to (2n+j-2,2n+j-2) \text{ for } j=2,\ldots,n-1.$$
3. The edge $((2,j),(1,j))$ is mapped to the L-shaped path:
$$\begin{aligned}(n+1,\ 2n+j-2) &\to (n,2n+j-2) \to \ldots \to (n-j+1,2n+j-2)\\ &\to (n-j+1,2n+j-3) \to \ldots \to (n-j+1,n-j+1) \text{ for } j=2,\ldots,n-1.\end{aligned}$$
4. The edge $((i,n-1),(i,n))$ is mapped to the L-shaped path:
$$\begin{aligned}(n+i-1,\ 3n-3) &\to (n+i-1,3n-2) \to \ldots \to (n+i-1,4n-i-2)\\ &\to (n+i,4n-i-2) \to \ldots \to (4n-i-2,4n-i-2) \text{ for } i=2,\ldots,n-1.\end{aligned}$$

By construction, it follows that the paths in our edge embedding are vertex disjoint. ◀

Reducing Half-Grids and Bringing the Piece Together. We now review the construction of Gitler [19], which shows how to reduce half-grids to much smaller half-grids (excluding diagonal edges) whose size depends only on k. Recall that $\hat{T}_k^n$ is the graph T_k^n without the diagonal edges. The proof of the lemma below is deferred to the full version.

▶ **Lemma 11** ([19]). *For any positive k,n with $k<n$, T_k^n is Wye-Delta reducible to $\hat{T}_k^k$.*

Combining the above reductions leads to the following theorem:

▶ **Theorem 12.** *Let G be a k-terminal planar graph where all terminals lie on the outer face. Then G admits a quality 1-vertex cut sparsifier of size $O(k^2)$, which is also a planar graph.*

Proof. Let n denote the number of vertices in G. First, we apply Lemma 9 on G to obtain a grid graph H with $O(n^2)$ vertices, which preserves exactly all terminal minimum cuts of G. We then apply Lemma 10 on H to obtain a node embedding ρ into the half-grid T_k^ℓ, where $\ell = 4n-3$. By Lemma 7, $\rho(H)$ preserves exactly all terminal minimum cuts of H. We can further extend $\rho(H)$ to the full half-grid T_k^ℓ, if dummy non-terminals and zero edge capacities are added. Finally, we apply Lemma 11 on T_k^ℓ to obtain a Wye-Delta reduction to the reduced half-grid graph $\hat{T}_k^k$. It follows by Lemma 6 that $\hat{T}_k^k$ is a quality 1-vertex cut sparsifier of T_k^ℓ, where the size guarantee is immediate from the definition of $\hat{T}_k^k$. ◀

The results about flow and distance sparsifiers are deferred to the full version.

4 Rechability-Preserving Minors for General Digraphs

In this section we show that any k-terminal digraph admits a reachability-preserving minor of size $O(k^4)$. We accomplish this by first restricting our attention to DAGs, and then showing how to generalize the result to any digraph. Details are deferred to the full version.

We first introduce the following definition. Given a k-terminal digraph G with a terminal pair-set P, we say that H is a *reachability-preserving minor with respect to* P, if H is a minor of G that preserves the reachability information only among the pairs in P. Note that the previous definition of reachability-preserving minor of G corresponds to the special case when the pair-set P is *trivial*, i.e., for any pair $x, x' \in K$, both (x, x') and (x', x) belong to P. Observe that the trivial pair-set contains $k(k-1)$ terminal-pairs.

We next review a useful scheme for breaking ties between shortest paths connecting some vertex pair from P. This tie-breaking is usually achieved by slightly perturbing the edge lengths of the original graph such that no two paths have the same length (note that in our case, edge lengths are initially one). The perturbation gives a *consistent* scheme in the sense that whenever π is chosen as a shortest path, every sub-path of π is also chosen as a shortest path. Below we formalize these ideas using two definitions and a lemma from [4].

▶ **Definition 13** (Tie-breaking Scheme)**.** Given a k-terminal G, a *shortest path tie breaking scheme* is a function π that maps every pair of vertices (s, t) to some shortest path between s and t in G. For any pair-set P, we let $\pi(P)$ denote the union over all shortest paths between pairs in P with respect to the scheme π.

▶ **Definition 14** (Consistency)**.** A tie-breaking scheme is consistent if, for all vertices y, x, x', y', if $x, x' \in \pi(y, y')$ with $d(y, x) < d(y, x')$, then $\pi(x, x')$ is a sub-path of $\pi(y, y')$.

▶ **Lemma 15** ([4])**.** *For any k-terminal G, there is a consistent tie-breaking scheme in G.*

Let G be a k-terminal DAG. Given a tie-breaking scheme π, the first step to construct a reachability-preserving minor is to start with an empty graph H and then for every pair $p \in P$, repeatedly add the shortest-path $\pi(p)$ to H. We can alternatively think of this as deleting vertices and edges that do not participate in any shortest path among terminal-pairs in P with respect to the scheme π. Clearly, the DAG $H = (V_H, E_H)$, $E_H := \pi(P)$, is a minor of G and preservers all reachability information among pairs in P. We next review the notion of a branching event, which will be useful to bound the size of H.

▶ **Definition 16** (Branching Event)**.** A *branching event* is a set of two distinct directed edges $\{e_1 = (u_1, v), e_2 = (u_2, v)\}$ that enter the same node v.

▶ **Lemma 17.** *The* DAG *H has at most $|P|(|P|-1|)/2$ branching events.*

The proof of the above lemma is deferred to the full version. We now have all the tools to present our algorithm for constructing reachability-preserving minors for DAGs.

The proofs of the lemma and the theorem below are deferred to the full version.

▶ **Lemma 18.** *Given a k-terminal* DAG *G with a pair-set P, the above algorithm outputs a reachability-preserving minor H of size $O(|P|^2)$ for G with respect to P.*

▶ **Theorem 19.** *Given a k-terminal digraph G with a pair-set P, there is an algorithm that constructs a reachability-preserving minor H of size $O(|P|^2)$ with respect to P.*

Taking P to be the trivial pair-set we obtain the following corollary.

▶ **Corollary 20.** *Any k-terminal digraph admits reachability-preserving minor of size $O(k^4)$.*

Algorithm 1 MinorSparsifyDag (k-terminal DAG G, pair-set P)

1: Set $H = \emptyset$ and compute a consistent tie-breaking scheme π for shortest paths in G.
2: For each $p \in P$, add the shortest path $\pi(p)$ to H.
3: **while** there is an edge (u, v) directed towards a non-terminal v with $\deg^-_H(v) = 1$ **do**
4: Contract the edge (u, v).
5: **end while**
6: **return** H

5 Reachability-Preserving Minors for Planar Digraphs

In this section we show that any k-terminal planar digraph G admits a reachability-preserving minor of size $O(k^2 \log^2 k)$. This matches the lower-bound of Theorem 3 up to a $O(\log^2 k)$ factor. The main idea is as follows. Given a k-terminal planar digraph G with the trivial pair-set P, $|P| = k(k-1)$, our goal will be to slightly increase the number of terminals while considerably reducing the size of the pair-set P, under the condition that no reachability information is lost among the terminal-pairs in P.

We apply the following Preprocessing Step. Given a k-terminal digraph G, we apply Theorem 19 to get a reachability-preserving minor G'. To simplify the notation, we will use G instead of G', i.e., throughout we assume that G has at most $O(k^4)$ vertices.

Decomposition into Path-Separable Digraphs and the Algorithm. We say that a graph $G = (V, E)$ admits an *α-separator* if there exists a set $S \subset V$ whose removal partitions G into connected components, each of size at most $\alpha \cdot |V|$, where $1/2 \leq \alpha < 1$. If the vertices of S consist of the union over r paths of G, for some $r \geq 1$, we say that G is *(α, r)-path separable*. We now review the following reduction due to Thorup [39].

▶ **Theorem 21** ([39]). *Given a digraph G, we can construct a series of digraphs $G_0, \ldots, G_n$ such that the number of vertices and edges over all G_i's is linear in the number of vertices and edges in G, and*

1. *Each vertex and edge of G appears in at most two G_i's.*
2. *For all $u, v \in V$, if there is a dipath R from u to v in G, there is a G_i that contains R.*
3. *Each $G_i = (V_i, E_i)$ is $(1/2, 6)$-path separable.*
4. *Each G_i is a minor of G. In particular, if G is planar, so is G_i.*

Now we review how directed reachability can be efficiently represented by separator dipaths. Let G be a k-terminal directed graph that contains some directed path Q. Assume that the vertices of Q are ordered in increasing order in the direction of Q. For each terminal $x \in K$, let $\texttt{to}_x[Q]$ be the first vertex in Q that can be reached by x, and let $\texttt{from}_x[Q]$ be the last vertex in Q that reaches x. Let (s, t) be a terminal pair and let R be the directed path from s to t in G. We say that R intersects Q iff s can reach $\texttt{to}_s[Q]$ and t can be reached from $\texttt{from}_t[Q]$ in Q, and $\texttt{to}_s[Q]$ precedes $\texttt{from}_t[Q]$ in Q.

We now are going to combine the above tools to give our labelling algorithm aimed at reducing the size of the trivial pair-set P. By Theorem 21, we restrict our attention only to the digraphs G_i. Let $K_i := V(G_i) \cap K$ be the set of terminals restricted to the graph G_i.

▶ **Lemma 22.** *Let G be a k-terminal planar digraph. Let $P' := \cup_{i=1}^{t-1} P'_i$ be the union over all pair-sets output by running Algorithm 2 below on each digraph G_i. Then the size of $|P'|$ is at most $O(k \log k)$. Moreover, if H is a reachability-preserving minor of G with respect to P', then H is a reachability-preserving minor of G with respect to all terminal pairs.*

Algorithm 2 ReducePairSet (planar digraph G_i, terminals K_i)

1: **if** $|V(G_i)| \leq 1$ or $K_i = \emptyset$ **then return** $\emptyset$.
2: Let $P_i' = \emptyset$ be the new pair-set.
3: Compute a 1/2-separator S of G_i consisting of 6 dipaths by Item 4 of Theorem [39].
4: **for** each dipath $Q \in S$ **do**
5: `// Addition of terminal connections with` Q
6: Let Q' be the set of existing terminals of Q.
7: **for** each terminal $x \in K_i$ **do**
8: Compute $\texttt{to}_x[Q]$ and $\texttt{from}_x[Q]$, declare them terminals and add them to Q'.
9: Add $(x, \texttt{to}_x[Q])$ and $(\texttt{from}_x[Q], x)$ to P_i'.
10: **end for**
11: `// Sparsification of` Q `using` Q'
12: Define directed pairs (s, t), where s and t are consecutive terminals of Q', according to the ordering of Q and add all these pairs to P_i'.
13: **end for**
14: Let $(G_i^{(1)}, K_i^{(1)})$ and $(G_i^{(2)}, K_i^{(2)})$ be the resulting graphs from $G \setminus S$, where $K_i^{(1)}$ and $K_i^{(2)}$ are disjoint subsets of the terminals K separated by S.
15: `// Note that reachability info. about terminals in` S `are taken care of.`
16: **return** $P_i' \cup \bigcup_{j=1}^{2}$ ReducePairSet$(G_i^{(j)}, K_i^{(j)})$.

Proof. By preprocessing, G has at most $O(k^4)$ vertices. Throughout, it will be useful to think of the above algorithm as simultaneously running it on each digraph G_i. By Item 2 of Theorem 21, each terminal appears in at most two G_i's. Thus at each recursive level, there will be at most $O(k)$ active G_i's. Also, note that the separator properties imply that there are $O(\log k)$ recursive calls overall.

We next bound the size of the pair-set P'. Let q denote the total number of newly added terminals in Line 8 per recursive level. Since there are $O(k)$ terminals, each adding at most $O(1)$ new terminals, it follows that $q = O(k)$. First, we argue about the number of pairs added in Line 9. Since this is bounded by $O(q)$, we get that there are $O(k \log k)$ pairs overall. Second, we bound the number of pairs added when sparsifying the separator paths, i.e., pair additions in Line 13. For all the separators in the same recursive level, we can write $q := \sum_i |Q_j'|$, where Q_j' denotes the set newly added terminals for some separator dipath (Line 7). By Line 12, it follows that we need only $(|Q_j'| - 1)$ pairs to represent each such dipath. Thus, per recursive call, the total number of pairs added in Line 13 is $O(q) = O(k)$. Summing these overall $O(\log k)$ levels, and combining this with the previous bound, gives the claimed bound on $|P'|$.

Finally, we argue that P' is a pair-set that can recover reachability information among terminals. Fix any terminal pair (s, t) and let R be a directed path from s to t in G. By Item 3 of Theorem 21, there is some digraph G_i that contains R. Then, R must intersect with some separator dipath Q, at some level of the recursion of the above algorithm on G_i. The above discussion gives that P' contains all the necessary information to give a (possibly) another directed path from s to t in G. ◀

Applying Theorem 19 on the digraph G with pair-set P', as defined by the above lemma, we get Theorem 2.

References

1 Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

2 Alexandr Andoni, Anupam Gupta, and Robert Krauthgamer. Towards $(1+\varepsilon)$-approximate flow sparsifiers. In *Proc. of the 25th SODA*, pages 279–293, 2014.

3 András A. Benczúr and David R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In *Proc. of the 28th STOC*, pages 47–55, 1996.

4 Greg Bodwin. Linear size distance preservers. In *Proc. of the 28th SODA*, pages 600–615, 2017.

5 T-H Hubert Chan, Donglin Xia, Goran Konjevod, and Andrea Richa. A tight lower bound for the steiner point removal problem on trees. In *Proc. of the 9th APPROX/RANDOM*, pages 70–81, 2006.

6 Moses Charikar, Tom Leighton, Shi Li, and Ankur Moitra. Vertex sparsifiers and abstract rounding algorithms. In *Proc. of the 51th FOCS*, pages 265–274, 2010.

7 Chandra Chekuri, Anupam Gupta, Ilan Newman, Yuri Rabinovich, and Alistair Sinclair. Embedding k-outerplanar graphs into l1. *SIAM J. Discrete Math.*, 20(1):119–136, 2006.

8 Chandra Chekuri, Sanjeev Khanna, and F Bruce Shepherd. Edge-disjoint paths in planar graphs with constant congestion. *SIAM J. Comput.*, 39(1):281–301, 2009.

9 Chandra Chekuri, F Bruce Shepherd, and Christophe Weibel. Flow-cut gaps for integer and fractional multiflows. In *Proc. of the 21st SODA*, pages 1198–1208, 2010.

10 Yun Kuen Cheung, Gramoz Goranci, and Monika Henzinger. Graph minors for preserving terminal distances approximately - Lower and Upper Bounds. In *Proc. of the 43rd ICALP*, pages 131:1–131:14, 2016.

11 Julia Chuzhoy. On vertex sparsifiers with steiner nodes. In *Proc. of the 44th STOC*, pages 673–688, 2012.

12 Julia Chuzhoy. Routing in undirected graphs with constant congestion. In *Proc. of the 44th STOC*, pages 855–874, 2012.

13 Don Coppersmith and Michael Elkin. Sparse sourcewise and pairwise distance preservers. *SIAM J. Discrete Math.*, 20(2):463–501, 2006.

14 Edward B Curtis, David Ingerman, and James A Morrow. Circular planar graphs and resistor networks. *Linear algebra and its applications*, 283(1):115–150, 1998.

15 Krzysztof Diks and Piotr Sankowski. Dynamic plane transitive closure. In *Proc. of the 15th ESA*, pages 594–604, 2007.

16 Michael Elkin, Arnold Filtser, and Ofer Neiman. Terminal embeddings. In *Proc. of the 18th APPROX/RANDOM*, pages 242–264, 2015.

17 Matthias Englert, Anupam Gupta, Robert Krauthgamer, Harald Räcke, Inbal Talgam-Cohen, and Kunal Talwar. Vertex sparsifiers: New results from old techniques. *SIAM J. Comput.*, 43(4):1239–1262, 2014.

18 Thomas A Feo and J Scott Provan. Delta-wye transformations and the efficient reduction of two-terminal planar graphs. *Operations Research*, 41(3):572–582, 1993.

19 Isidoro Gitler. *Delta-Wye-Delta Transformations: Algorithms and Applications*. PhD thesis, Department of Combinatorics and Optimization, University of Waterloo, 1991.

20 Gramoz Goranci and Harald Räcke. Vertex sparsification in trees. In *Proc. of the 14th WAOA*, pages 103–115, 2016.

21 Anupam Gupta. Steiner points in tree metrics don't (really) help. In *Proc. of the 12th SODA*, pages 220–227, 2001.

22 Torben Hagerup, Jyrki Katajainen, Naomi Nishimura, and Prabhakar Ragde. Characterizing multiterminal flow networks and computing flows in networks of small treewidth. *J. Comput. Syst. Sci.*, 57(3):366–375, 1998.

23 Lior Kamma, Robert Krauthgamer, and Huy L Nguyen. Cutting corners cheaply, or how to remove steiner points. *SIAM J. Comput.*, 44(4):975–995, 2015.

24 Irit Katriel, Martin Kutz, and Martin Skutella. Reachability substitutes for planar digraphs. In *Technical Report MPI-I-2005-1-002*. Max-Planck-Institut für Informatik, 2005.

25 Arindam Khan and Prasad Raghavendra. On mimicking networks representing minimum terminal cuts. *Inf. Process. Lett.*, 114(7):365–371, 2014.

26 Robert Krauthgamer, Huy L Nguyen, and Tamar Zondiner. Preserving terminal distances using minors. *SIAM J. Discrete Math.*, 28(1):127–141, 2014.

27 Robert Krauthgamer and Inbal Rika. Mimicking networks and succinct representations of terminal cuts. In *Proc. of the 24th SODA*, pages 1789–1799, 2013.

28 Robert Krauthgamer and Inbal Rika. Refined vertex sparsifiers of planar graphs. *CoRR*, abs/1702.05951, 2017.

29 Robert Krauthgamer and Tamar Zondiner. Preserving terminal distances using minors. In *Proc. of the 39th ICALP*, pages 594–605, 2012.

30 James R Lee, Manor Mendel, and Mohammad Moharrami. A node-capacitated okamura-seymour theorem. In *Proc. of the 45th STOC*, pages 495–504, 2013.

31 Frank Thomson Leighton and Ankur Moitra. Extensions and limits to vertex sparsification. In *Proc. of the 42nd STOC*, pages 47–56, 2010.

32 Konstantin Makarychev and Yury Makarychev. Metric extension operators, vertex sparsifiers and lipschitz extendability. In *Proc. of the 51th FOCS*, pages 255–264, 2010.

33 Jiří Matoušek. On the distortion required for embedding finite metric spaces into normed spaces. *Israel Journal of Mathematics*, 93(1):333–344, 1996.

34 Ankur Moitra. Approximation algorithms for multicommodity-type problems with guarantees independent of the graph size. In *Proc. of the 50th FOCS*, 2009.

35 Haruko Okamura and Paul D. Seymour. Multicommodity flows in planar graphs. *J. Comb. Theory, Ser. B*, 31(1):75–81, 1981.

36 Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM J. Comput.*, 40(4):981–1025, 2011.

37 Sairam Subramanian. A fully dynamic data structure for reachability in planar digraphs. In *Proc. of the 1st ESA*, pages 372–383, 1993.

38 Roberto Tamassia and Ioannis G Tollis. Planar grid embedding in linear time. *IEEE Trans. Circuits Syst.*, 36(9):1230–1234, 1989.

39 Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, 2004.

40 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.

41 Leslie G. Valiant. Universality considerations in VLSI circuits. *IEEE Trans. Computers*, 30(2):135–140, 1981.

The Power of Vertex Sparsifiers in Dynamic Graph Algorithms*

Gramoz Goranci[1], Monika Henzinger[2], and Pan Peng[3]

1 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
`gramoz.goranci@univie.ac.at`
2 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
`monika.henzinger@univie.ac.at`
3 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
`pan.peng@univie.ac.at`

Abstract

We introduce a new algorithmic framework for designing dynamic graph algorithms in minor-free graphs, by exploiting the structure of such graphs and a tool called *vertex sparsification*, which is a way to compress large graphs into small ones that well preserve relevant properties among a subset of vertices and has previously mainly been used in the design of approximation algorithms.

Using this framework, we obtain a Monte Carlo randomized fully dynamic algorithm for $(1+\varepsilon)$-approximating the energy of electrical flows in n-vertex planar graphs with $\tilde{O}(r\varepsilon^{-2})$ worst-case update time and $\tilde{O}((r+\frac{n}{\sqrt{r}})\varepsilon^{-2})$ worst-case query time, for any r larger than some constant. For $r = n^{2/3}$, this gives $\tilde{O}(n^{2/3}\varepsilon^{-2})$ update time and $\tilde{O}(n^{2/3}\varepsilon^{-2})$ query time. We also extend this algorithm to work for minor-free graphs with similar approximation and running time guarantees. Furthermore, we illustrate our framework on the all-pairs max flow and shortest path problems by giving corresponding dynamic algorithms in minor-free graphs with both sublinear update and query times. To the best of our knowledge, our results are the first to systematically establish such a connection between dynamic graph algorithms and vertex sparsification.

We also present both upper bound and lower bound for maintaining the energy of electrical flows in the incremental subgraph model, where updates consist of only vertex activations, which might be of independent interest.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Dynamic graph algorithms, electrical flow, minor-free graphs, max flow

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.45

1 Introduction

A *dynamic graph* is a graph that undergoes constant changes over time. Such changes or updates may correspond to inserting/deleting an edge or activating/deactivating a vertex from the graph. The goal of a *dynamic graph algorithm* is to maintain some property of a graph and support an intermixed sequence of update and query operations that can be processed quickly. In particular, the algorithm should at least beat the trivial one that recomputes the solution from scratch after each update. The last three decades have witnessed a large body of research on dynamic graph algorithms for a number of fundamental

* The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

properties, including connectivity, minimum spanning tree, shortest path, matching and so on. Most of these problems have been considered in both general graphs as well as planar graphs, with quite different techniques and trade-offs between update and query times. In particular, many dynamic algorithms for planar graphs heavily depend on the duality of planar graphs [41, 26, 19] and do not seem easily generalizable to a larger class of graphs, e.g., the family of minor-free graphs.

In this paper, we provide a new algorithmic framework for designing dynamic graph algorithms in *minor-free* graphs that are free of a K_t-minor for any fixed integer $t \geq 1$, by utilizing a tool called *vertex sparsification* as well as the structure of minor-free graphs. Vertex sparsification is a way of compressing large graphs into smaller ones that well preserve the relevant properties (e.g., cut, flow and distance information) among a subset of vertices (called *terminals*) (e.g., [36, 7, 29]). Besides the natural motivation of achieving more space-efficient storage and obtaining faster algorithms on the reduced graphs, it has also found applications in the design of approximation algorithms [36], network design and routing [11]. We show that good quality and efficiently constructible vertex sparsifiers can be used to give efficient dynamic graph algorithms. To the best of our knowledge, our results are the first to systematically establish such a connection between dynamic graph algorithms and vertex sparsification.

We illustrate our algorithmic framework on the *all-pairs electrical flow, all-pairs max flow* and *all-pairs shortest path* problems in minor-free graphs. Previously, there is no known dynamic algorithms for the first problem (even for special class of graphs), and for the second problem, we only know dynamic algorithms for planar graphs. Due to space constraints, we focus on electrical flow in the conference version of the paper and give the results for max flow and shortest path in the full version.

The *electrical flow* problem is one of the most fundamental problems in electrical engineering and physics [12], and recently received increasing interest in computer science due to its close relation to linear equation solvers [40, 28], graph sparsification [39, 38], maximum flows (and minimum cuts) [10, 31, 33, 22, 34]. Slightly more formally, the $s-t$ electrical flow problem asks to find the flow (current) that minimizes the energy dissipation of a weighted graph when one unit of flow is injected at the source s and extracted at the sink t.

In the dynamic *all-pairs electrical flow* problem, our objective is to minimize the update time and the query time for outputting the exact (or approximate) energy of the $s-t$ electrical flow (see Section 2 for formal definitions) in the current graph, for any two vertices s, t. In the following, we will focus on *fully dynamic* graphs, in which updates consist of both edge insertions and deletions. Specifically, we allow the following operations:

- $\textsc{Insert}(u, v, r)$: Insert the edge (u, v) with resistance r in G, provided that the new edge preserves the planarity (or minor-freeness) of G.
- $\textsc{Delete}(u, v)$: Delete the edge (u, v) from G.
- $\textsc{ElectricalFlow}(s, t)$: Return the exact (or approximate) energy of the $s-t$ electrical flow in the current graph G.

For a graph G and two vertices s, t, we let $\mathcal{E}_G(s, t)$ denote the energy of the $s-t$ electrical flow. For any $\alpha \geq 1$, we say that an algorithm is an α-approximation to $\mathcal{E}_G(s, t)$ if $\textsc{ElectricalFlow}(s, t)$ returns a positive number k such that $\mathcal{E}_G(s, t) \leq k \leq \alpha \cdot \mathcal{E}_G(s, t)$.

1.1 Our Results

We present the first non-trivial fully dynamic algorithm for maintaining a $(1+\varepsilon)$-approximation to the energy of the $s-t$ electrical flow in planar and minor-free graphs. Our algorithm achieves both sublinear worst-case query and update times. (Throughout the paper, we

use $\tilde{O}(\cdot)$ to hide polylogarithmic factors, i.e., $\tilde{O}(f(n)) = O(f(n) \cdot \operatorname{poly}\log f(n))$; "with high probability" refers to "with probability at least $1 - \frac{1}{n^c}$, for some $c > 0$".)

▶ **Theorem 1.** *Fix $\varepsilon \in (0,1)$ and integer $t > 0$. Let $r \geq c$ for some large constant $c > 0$. Given a K_t-minor-free graph $G = (V, E, \mathbf{w})$ with positive edge weights, we can maintain a $(1+\varepsilon)$-approximation to the all-pairs electrical flow problem with high probability. The worst-case update time per operation is $\tilde{O}(\frac{n^\xi \cdot r}{\varepsilon^2})$ and the worst-case query time is $\tilde{O}((r + n/\sqrt{r})\varepsilon^{-2})$, for any constant $\xi > 0$. Furthermore, if G is planar, then ξ can be chosen to be 0.*

Note that by setting $r = n^{2/3}$, we obtain a dynamic algorithm for planar graphs with worst-case $\tilde{O}(n^{2/3}\varepsilon^{-2})$ update time and $\tilde{O}(n^{2/3}\varepsilon^{-2})$ query time. One may be tempted to reduce our problem to dynamically maintaining *spectral sparsifiers*. Despite the fact that such sparsifiers approximately preserve electrical flows and that a $(1 \pm \varepsilon)$-spectral sparsifier can be maintained with amortized update time $\operatorname{poly}(\log n, \varepsilon^{-1})$ [5], performing query operations on the sparsifier about the energy of $s - t$ electrical flow still requires $\Omega(n)$ time.

We also give a dynamic algorithm for all-pairs electrical flow for minor-free graphs in the *incremental subgraph* model, where the updates in the dynamic graph are a sequence of *vertex activation* operations. Our algorithm maintains a $(1+\varepsilon)$-approximation of the energy of electrical flows in minor-free graphs with $\tilde{O}(r\varepsilon^{-2})$ *amortized* update time and $\tilde{O}((r + n/\sqrt{r})\varepsilon^{-2})$ *worst-case* query time, for any $r \geq c$. For $r = n^{2/3}$, this gives $\tilde{O}(n^{2/3}\varepsilon^{-2})$ amortized update and worst-case query time. We complement this result by showing the following conditional lower bound: there is no incremental algorithm in the subgraph model that C-approximates the energy of the electrical flows in *general* graphs with both $O(n^{1-\varepsilon})$ worst-case update time and $O(n^{2-\varepsilon})$ worst-case query time, for any $C > 0$ and constant $\varepsilon > 0$, unless the *online matrix vector multiplication (oMv)* conjecture if false. Our results show a polynomial gap of dynamic algorithms for subgraph electrical flows between minor-free graphs and general graphs, conditioned on the oMv conjecture. These results might be of independent interest and the details are deferred to the full version, due to space constraints.

Our second result from our algorithmic framework is a dynamic algorithm for all-pairs max flows in minor free graphs. In this problem, the query $\textsc{MaxFlow}(s,t)$ asks the exact (or approximate) $s - t$ max flow value in the current graph.

▶ **Theorem 2.** *Let $t > 0$ be a fixed integer. Let $r \geq c$ for some large constant $c > 0$. Given a K_t-minor-free graph $G = (V, E, \mathbf{w})$ with positive edge weights, we can maintain a $O(1)$-approximation to the all-pairs max-flows problem. The worst-case update time is $\tilde{O}(n^\xi r + r^3)$ for any constant $\xi > 0$, and the worst-case query time is $\tilde{O}(r + n/\sqrt{r})$. One can also maintain a $O(\log^4 n)$-approximation to the all-pairs max-flows problem, with worst-case update time $\tilde{O}(n^\xi r + r)$, and worst-case query time $\tilde{O}(r + n/\sqrt{r})$.*

Note that by setting $r \in \left(\operatorname{poly}\log n, o(n^{1/3})\right)$, we can maintain a $O(1)$-approximation to the all-pairs max flows problem in minor free graphs with both sublinear worst-case update and query times. By setting $r = n^{2/3}$, we can obtain $O(\log^4 n)$-approximation with worst-case $\tilde{O}(n^{\xi+2/3})$ update time and $\tilde{O}(n^{2/3})$ query time.

We remark that Italiano et al. [19] have given a fully dynamic algorithm for *exact* all-pairs max-flow in planar graphs with worst-case $\tilde{O}(n^{2/3})$ update and $\tilde{O}(n^{2/3})$ query time. Their algorithm is based on maintaining an edge decomposition (called *r-division*) of the planar graph, which is similar to ours, while there are some substantial differences. First of all, their algorithm does not seem easily generalizable to minor-free graphs since it depends on the duality of planar graphs. Second, it is required that the embedding of the graph does not change throughout the sequence of updates [19], which is not necessary in our algorithm.

Third, though their algorithm can answer the exact max flow value with the aforementioned running time guarantee, it does not provide an update/query trade-off as ours.

Our third result is a fully dynamic algorithm for all-pairs shortest paths in minor-free graphs. In this problem, the query ShortestPath(s,t) asks the exact (or approximate) shortest path length between s and t in the current graph.

▶ **Theorem 3.** *Let $t, q \geq 1$. Given a K_t-minor-free graph $G = (V, E, \mathbf{w})$ with positive edge weights, we can maintain a $(2q-1)$-approximation to the all-pair shortest path problem. The worst-case expected update time is $\tilde{O}(n^{6/7})$ and the worst-case query time is $\tilde{O}(n^{\frac{6}{7}+\frac{3}{7q}})$.*

Note that for $q \geq 4$, both update time and query time of the above algorithm will be sublinear. We remark that for the special case of planar graphs, the above running time and approximation guarantee are worse than the result of Abraham et al. [3], who gave a fully dynamic $(1+\varepsilon)$-approximation for planar shortest path problem with worst-case $\tilde{O}(\sqrt{n})$ update and query time. However, it is unclear how to generalize their algorithm to minor-free graphs. There are also works on fully dynamic all-pairs shortest path in general graphs (e.g., [9, 4]), for which there is no known algorithm with non-trivial worst-case update time that breaks $O(n)$ barrier.

We want to point it out that all the above results might be generalized to a larger class of graphs that admit efficiently constructible good separators, while our main focus is to bring up this new algorithmic framework. Due to space constraints, the proofs of Theorem 2 and 3 are deferred to the full version.

1.2 Our Techniques

Our fully dynamic algorithms in planar and minor-free graphs combine the ideas of maintaining an edge decomposition of the current graph G and approximately preserving the relevant properties or quantities by smaller "substitutes", which allow us to operate on a small piece of the graph during each update (in the amortized sense) and significantly reduce the size of the query graph that well preserves the property of G. These "substitutes" refer to the vertex sparsifiers for the corresponding properties.

Such an edge decomposition is called r-division [15]. Given some graph G and a parameter r, we partition G into a collection of $O(n/r)$ edge disjoint subgraphs (called *regions*), each contains at most $O(r)$ vertices. This induces a partitioning of the vertex set into *interior* vertices (those that are incident only to vertices within the same region) and *boundary* vertices (those that are incident to vertices in different regions). In addition, we ensure that the total number of boundary vertices is $O(n/\sqrt{r})$. Maintaining an r-division has also been used in some previous dynamic algorithms for planar graphs [41, 26, 16, 19].

Now a key observation is that for any s, t, by removing all the interior vertices from other regions that do not contain s, t and adding some edges with appropriate weights among boundary vertices, one can guarantee that the resulting graph exactly preserves the quantities between s, t (e.g., the energy of $s-t$ electrical flow, the value of $s-t$ max flow). Now let us elaborate on the electrical flow problem, for which the aforementioned reduction is called *Schur Complement*. The problem of performing such a Schur Complement on a region is that it is very time-consuming as it adds too many edges among boundary vertices. Instead, we resort to a recent tool called approximate Schur Complement ([13]; see Section 3.1), which well approximates the pairwise effective resistances among boundary vertices and also gives a sparse graph (or a substitute) induced by all boundary vertices. Now for an update, we only recompute a constant number of such substitutes (and we need to periodically rebuild the data structure); for a query, we take the small graph defined by choosing appropriate regions

and substitutes, and answer the query according to the $s-t$ effective resistance on this small graph. Since such a substitute can be computed very fast and is sparse, we are ensured to obtain sublinear amortized update time and worst-case query time. Using a global rebuilding technique, we show that one can also achieve worst-case update time.

Our approach differs from the previous dynamic planar graph algorithms in that the r-division we use does not require that the boundary of each region contains a constant number of faces or the duality of planar graphs, since we only need to maintain the r-division and fast compute the approximate Schur Complement.

Such an approximate Schur Complement can be viewed as a vertex *spectral/resistance sparsifier* by treating boundary vertices as terminals. To obtain dynamic algorithms for the all-pairs max flows (resp., shortest paths) problems, we can use vertex cut sparsifiers (resp., distance sparsifiers), which well preserve the values of minimum cut separating any subset of terminals (resp., the distances among all terminal pairs).

1.3 Related Work

In the static setting, the electrical flow problem amounts to solving a system of linear equations, where the underlying matrix is a Laplacian (see the monograph of Doyle and Snell [12]). Christiano et al. [10] used the electrical flow computation as a subroutine within the multiplicative-weights update framework [8], to obtain the breakthrough result of $(1-\varepsilon)$-approximating the undirected maximum $s-t$ flow (and the minimum $s-t$ cut) in $\tilde{O}(mn^{1/3}\varepsilon^{-11/3})$ time. This has inspired and led to further development of fast algorithms for approximating $s-t$ maximum flow, which culminated in an $\tilde{O}(m)$ time algorithm for this problem in undirected graphs [37].

Lipton, Rose and Tarjan [32] consider the problem of designing fast algorithms for *exactly* solving linear systems where the matrix is positive definite and the associated graph is planar. Their result implies an $O(n^{3/2})$ time algorithm for electrical flow in planar graphs. This was latter improved to $O(n^{\omega/2})$ by Alon and Yuster [6], where $\omega = 2.37..$ is the exponent in the running time of the fastest algorithm for matrix multiplication [42]. Miller and Koutis [27] consider parallel algorithms for approximately solving planar Laplacian systems. Their algorithm runs in $\tilde{O}(n^{1/6+c})$ parallel time and $O(n)$ work, where c is any positive constant. We refer the reader to [24] for other useful properties of Laplacians on planar graphs.

Related data structure concepts dealing with spectral properties of graphs include semi-streaming and dynamic algorithms for maintaining spectral sparsifiers. Kelner and Levin [23] give single-pass incremental streaming algorithm using near-linear space and total update time. This was extended by Kapralov et al. [20] to the dynamic semi-streaming model which allows both edge insertions and deletions. Recently, Abraham et al. [5] give a fully-dynamic algorithm for maintaining spectral sparsifiers in poly-logarithmic amortized update time.

There is a line of work on dynamic algorithms for planar graphs that maintains information about important measures like reachability, connectivity, shortest path, max-flow etc. Subramanian [41] shows a fully-dynamic algorithm for maintaining reachability in directed planar graphs in $O(n^{2/3}\log n)$ time per operation. For the connectivity measure, Eppstein et al. [14] give an algorithm with $O(\log^2 n)$ amortized update time and $O(\log n)$ query time. Dynamic all-pairs shortest path problem in planar graphs was initiated by Klein and Subramanian [26], who showed how to maintain a $(1+\varepsilon)$-approximation to shortest paths in $O(n^{2/3}\log^2 n \log D)$ amortized update time and $O(n^{2/3}\log^2 n \log D)$ worst-case query time, where D denotes the sum of edge lengths. The best known algorithm is due to Abraham, Chechik and Gavoille [3] and maintains a $(1+\varepsilon)$-approximation in $O(\sqrt{n}\log^2 n/\varepsilon)$ worst-case time per operation. Italiano et al. [19] obtain a fully-dynamic algorithm for *exact* $s-t$ max-flow in planar graphs with $O(n^{2/3}\log^{8/3})$ worst-case time per operation.

Motivated by the recent developments on proving conditional lower-bounds for dynamic problems [2, 18], Abboud and Dahlgaard [1] give conditional lower-bounds for a class of dynamic graph problems restricted to planar graphs. Specifically, under the conjecture that all-pair-shortest path problem cannot be solved in truly subcubic time, they show that no algorithm for dynamic shortest path in planar graphs can support both updates and queries in $O(n^{1/2-\varepsilon})$ amortized time, for $\varepsilon > 0$.

2 Preliminaries

We consider a weighted undirected graph G undergoing edge insertions/deletions or vertex activations/deactivations. Our dynamic algorithms are characterized by two time measures: *query time*, which denotes the time needed to answer a query, and *update time*, which denotes the time needed to perform an update operation. We say that an algorithm has $O(t(n))$ *worst-case* update time, if it takes $O(t(n))$ time to process *each* update. We say that an algorithm has $O(t(n))$ *amortized* update time if it takes $O(f \cdot t(n))$ total update time for processing f updates (edge insertions/deletions or vertex activations/deactivations).

Basic Definitions. Let $G = (V, E, \mathbf{w})$ be any undirected weighted graph with n vertices and m edges, where for any edge e, its weight $\mathbf{w}(e) > 0$. Let $\mathbf{A}$ denote the weighted adjacency matrix, let $\mathbf{D}$ denote the weighted degree diagonal matrix, and let $\mathbf{L} = \mathbf{D} - \mathbf{A}$ denote the *Laplacian* matrix of G. We fix an arbitrary orientation of edges, that is, for any two vertices u, v connected by an edge, exactly one of $(u, v) \in E$ or $(v, u) \in E$ holds. Let $\mathbf{B} \in \mathbb{R}^{m \times n}$ denote the incidence matrix of G such that for any edge $e = (u, v)$ and vertex $w \in V$, $\mathbf{B}((u, v), w) = 1$ if $u = w$, -1 if $v = w$, and 0 otherwise. We will also think of the weight $\mathbf{w}(e)$ of any edge e as the *conductance* of e, and its reciprocal $\frac{1}{\mathbf{w}(e)}$, denoted as $\mathbf{r}(e)$, as the *resistance* of e. Let $\mathbf{R} \in \mathbb{R}^{m \times m}$ denote a diagonal matrix with $\mathbf{R}(e, e) = \mathbf{r}(e)$, for any edge e. Note that $\mathbf{L} = \mathbf{B}^T \mathbf{R}^{-1} \mathbf{B}$.

For any $\mathbf{x} \in \mathbb{R}^n$, the quadratic form associated with $\mathbf{L}$ is given by $\mathbf{x}^T \mathbf{L} \mathbf{x}$. For any two different vertices u, v, let $\chi_{u,v} \in \mathbb{R}^n$ denote the vector such that $\chi_{u,v}(w) = 1$ if $w = u$, -1 if $w = v$ and 0 otherwise. For any two vertices $s, t \in V$, an $s - t$ *flow* is a mapping $\mathbf{f} : E \to \mathbb{R}^+$ satisfying the following conservation constraint: for any $v \neq s, t$, it holds that $\sum_{e=(v,u)} \mathbf{f}(e) = \sum_{e=(u,v)} \mathbf{f}(e)$, where for any edge $e = (v, u)$, $\mathbf{f}(e) := \mathbf{f}(v, u)$ and $\mathbf{f}(u, v) := -\mathbf{f}(v, u)$.

We will let $\mathrm{val}(\mathbf{f}) = \sum_{v:(s,v)\in E} \mathbf{f}(s, v)$ denote the *value of an* $s - t$ *flow*. Note that for an $s - t$ flow with value 1, it holds that $\mathbf{B}^T \mathbf{f} = \chi_{s,t}$. Given an $s - t$ flow $\mathbf{f}$, its *energy* (with respect to the resistance vector $\mathbf{r}$) is defined as $\mathcal{E}_{\mathbf{r}}(\mathbf{f}, s, t) = \sum_e \mathbf{r}(e)\mathbf{f}(e)^2 = \mathbf{f}^T \mathbf{R} \mathbf{f}$.

We define the $s - t$ *electrical flow* in G to be the $s - t$ flow that minimizes the energy $\mathcal{E}_{\mathbf{r}}(\mathbf{f}, s, t)$ among all $s - t$ flows with *unit* flow value. It is known that such a flow is unique [12].

Any $s - t$ flow $\mathbf{f}$ in G is an $s - t$ electrical flow with respect to $\mathbf{r}$, iff there exists a vertex potential function $\phi : V \to \mathbb{R}^+$ such that for any $e = (u, v)$ that is oriented from u to v, $\mathbf{f}(e) = \frac{\phi(v) - \phi(u)}{\mathbf{r}(e)}$. It is known that such a vector ϕ satisfies that $\phi = \mathbf{L}^\dagger \chi_{s,t}$, where $\mathbf{L}^\dagger$ denotes the (Moore-Penrose) pseudo-inverse of $\mathbf{L}$. In addition, $\mathbf{f} = \mathbf{R}^{-1} \mathbf{B}^T \phi = \mathbf{R}^{-1} \mathbf{B}^T \mathbf{L}^\dagger \chi_{s,t}$ [12].

The *effective* $s - t$ *resistance* $R_G(\mathbf{r}, s, t)$ of G with respect to the resistances $\mathbf{r}$ is the potential difference between s, t when we send one unit of electrical flow from s to t. That is, $R_G(\mathbf{r}, s, t) = \phi(s) - \phi(t) = \chi_{s,t}^T \mathbf{L}^\dagger \chi_{s,t}$, where ϕ is the vector of vertex potentials induced by the $s - t$ electrical flow of value 1. We will often denote $R_G(\mathbf{r}, s, t)$ by $R_G(s, t)$ when $\mathbf{r}$ is clear from the context. It is known that the effective $s - t$ resistance is equal to the energy of the $s - t$ electrical flow of value 1, that is $R_G(\mathbf{r}, s, t) = \mathcal{E}_{\mathbf{r}}(\mathbf{f}, s, t)$.

Graph r-Divisions. Let $G = (V, E)$ be a graph. Let $F \subset E$ be a subset of edges. We call the subgraph G_F induced by all edges in F a *region*. For a subgraph P of G, any vertex that is incident to vertices not in P is called a *boundary* vertex. The *vertex boundary* of P, denoted by $\partial_G(P)$ is the set of boundary vertices belonging to P. All other vertices in P will be called *interior vertices* of P.

▶ **Definition 4.** Let $c_1, c_2 > 0$ be some constant. For any $r \in (1, n)$, a *weak r-division* (with respect to c_1, c_2) of an n-vertex graph G is an edge partition of it into regions $\mathcal{P} = \{P_1, \ldots, P_\ell\}$, where $\ell \leq c_1 \cdot \frac{n}{r}$ such that

- Each edge belongs to exactly one region.
- Each region P_i contains r vertices.
- The total number of all boundary vertices, i.e., $\cup_i \partial_G(P_i)$, is at most $c_2 n/\sqrt{r}$.

It is known that such an r-division (even with the stronger guarantee that each region has $O(\sqrt{r})$ boundary vertices) for planar graphs can be constructed in linear time [17, 25, 15].

▶ **Lemma 5** ([25]). *Let $c > 0$ be some constant. There is an algorithm that takes as input an n-vertex planar graph G and for any $r \geq c$, outputs an r-division of G in $O(n)$ time.*

We will need the following property on the boundary vertices of the r-division output by the above algorithm (see Section 3.3 in [25]).

▶ **Lemma 6** ([25]). *For an n-vertex planar graph G, let $\mathcal{P} = \{P_1, \ldots, P_\ell\}$, $\ell = O(n/r)$ be the r-division by the algorithm in Lemma 5. Then it holds that $\sum_{i=1}^{\ell} |\partial_G(P_i)| = O(n/\sqrt{r})$.*

Graph Sparsification. Graph Sparsification aims at compressing large graphs into smaller ones while (approximately) preserving some characteristics of the original graph. We present two notions of sparsification. The first requires that the quadratic form of the large and sparsified graph are close. The second requires that all-pairs effective resistances of the corresponding graphs are close.

▶ **Definition 7** (Spectral Sparsifier). Let $G = (V, E, \mathbf{w})$ be a weighted graph and $\varepsilon \in (0, 1)$. A $(1 \pm \varepsilon)$-*spectral sparsifier* for G is a subgraph $H = (V, E_H, \mathbf{w}_H)$ such that for all $\mathbf{x} \in \mathbb{R}^n$, $(1 - \varepsilon)\mathbf{x}^T\mathbf{L}\mathbf{x} \leq \mathbf{x}^T\widetilde{\mathbf{L}}\mathbf{x} \leq (1 + \varepsilon)\mathbf{x}^T\mathbf{L}\mathbf{x}$, where $\mathbf{L}$ and $\widetilde{\mathbf{L}}$ are the Laplacians of G and H, respectively.

▶ **Definition 8** (Resistance Sparsifier). Let $G = (V, E, \mathbf{w})$ be a weighted graph and $\varepsilon \in (0, 1)$. A $(1 \pm \varepsilon)$-*resistance sparsifier* for G is a subgraph $H = (V, E_H, \mathbf{w}_H)$ such that for all $u, v \in V$, $(1-\varepsilon)R_H(u, v) \leq R_G(u, v) \leq (1+\varepsilon)R_H(u, v)$, where $R_G(u, v)$ and $R_H(u, v)$ denote the effective $u - v$ resistance in G and H, respectively.

We remark that Definition 7 implies approximations for the pseudoinverse Laplacians, that is

$$\forall \mathbf{x} \in \mathbb{R}^n \quad \frac{1}{(1+\varepsilon)}\mathbf{x}^T\mathbf{L}^\dagger\mathbf{x} \leq \mathbf{x}^T\widetilde{\mathbf{L}}^\dagger\mathbf{x} \leq \frac{1}{(1-\varepsilon)}\mathbf{x}^T\mathbf{L}^\dagger\mathbf{x},$$

Since by definition, the effective resistance between any two nodes u and v is the quadratic form defined by the pseudo-inverse of the Laplacian computed at the vector $\chi_{u,v}$, it follows that the effective resistances between any two nodes in G and H are the same up to a $(1 \pm \varepsilon)$ factor. By our definitions for resistance and spectral sparsifiers, we have the following fact.

▶ **Fact 9.** *Let $\varepsilon \in (0, 1)$ and let G be a graph. Then every $(1 \pm \varepsilon)$-spectral sparsifier of G is a $(1 \pm \varepsilon)$-resistance sparsifier of G.*

The following lemma says that given a graph, by decomposing the graph into several pieces, and computing a good sparsifier for each piece, then one can obtain a good sparsifier for the original graph which is the union of the sparsifiers for all pieces. The proof is deferred to the full version of the paper.

▶ **Lemma 10** (Decomposability). *Let $G = (V, E, \mathbf{w})$ be a weighted graph whose set of edges is partitioned into $E_1, \ldots, E_\ell$. Let H_i be a $(1 \pm \varepsilon)$-spectral sparsifier of $G_i = (V, E_i)$, where $i = 1, \ldots, \ell$. Then $H = \bigcup_{i=1}^{\ell} H_i$ is a $(1 \pm \varepsilon)$-spectral sparsifier of G.*

3 A Dynamic Algorithm for Electrical Flow in Minor-Free Graphs

In order to present our dynamic algorithm for electrical flows, we first introduce the notion of *approximate Schur Complement*.

3.1 Schur Complement as Vertex Resistance Sparsifier

In the previous section we introduced graph sparsification for reducing the number of edges. For our application, it will be useful to define sparsifiers that apart from reducing the number of edges, they also reduce the number of vertices. More precisely, given a weighted graph $G = (V, E, \mathbf{w})$ with terminal set $K \subset V$, we are looking for a graph $H = (V_H, E_H, \mathbf{w}_H)$ with $K \subseteq V_H$ and as few vertices and edges as possible while preserving some important feature among terminals vertices. Graph H is usually referred to as a *vertex sparsifier* of G.

Exact Schur Complement. We first review a folklore result [35] on constructing vertex sparsifiers that preserve effective resistances among terminal pairs. For sake of simplicity, we first work with Laplcians of graphs. For a given connected graph G as above, let $N = V \setminus K$ be the set of non-terminal vertices in G. The partition of V into N and K naturally induces the following partition of the Laplacian $\mathbf{L}$ of G into blocks:

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_N & \mathbf{L}_M \\ \mathbf{L}_M^T & \mathbf{L}_K \end{bmatrix}$$

We remark that since G is connected and N and K are non-empty, $\mathbf{L}_N$ is invertible. We next define the Schur complement of $\mathbf{L}$, which can be viewed as an equivalent to $\mathbf{L}$ only on the terminal vertices.

▶ **Definition 11** (Schur Complement). The *Schur complement* of a graph Laplacian $\mathbf{L}$ with respect to a terminal set K is $\mathbf{L}_S^K := \mathbf{L}_K - \mathbf{L}_M^T \mathbf{L}_N^{-1} \mathbf{L}_M$.

It is known that the matrix $\mathbf{L}_S^K$ is a Laplacian matrix for some graph G' [30]. We can think of Schur Complement as performing Gaussian elimination on the non-terminals $V \setminus K$. This process recursively eliminates a vertex $v \in V \setminus K$ by deleting v and adding a clique with appropriate edge weights on the neighbors of v in the current graph (see, e.g. [30]). The following lemma shows that the quadratic form of the pseudo-inverse of the Laplacian $\mathbf{L}$ will be preserved by taking the quadratic form of the pseudo-inverse of its Schur Complement, for vectors supported on the terminals. See the full version for the proof.

▶ **Lemma 12.** *Let $\mathbf{d}$ be a vector of a graph G whose vertices are partitioned into terminals K, and non-terminals N and only terminals have non-zero entries in $\mathbf{d}$. Let $\mathbf{d}_K$ be the restriction of $\mathbf{d}$ on the terminals and let $\mathbf{L}_S^K$ be the Schur complement of the Laplacian $\mathbf{L}$ of G with respect to K. Then $\mathbf{d}^T \mathbf{L}^\dagger \mathbf{d} = \mathbf{d}_K^T (\mathbf{L}_S^K)^\dagger \mathbf{d}_K$.*

Using interchangeability between graphs and their Laplacians, we can interpret the above result in terms of graphs as well. We first present the following notion of sparsification.

▶ **Definition 13** (Vertex Resistance Sparsifier)**.** Let $G = (V, E, \mathbf{w})$ be a weighted graph with $K \subset V$ and $\alpha \geq 1$. An α-vertex resistance sparsifier of G with respect to K is a graph $H = (K, E_H, \mathbf{w}_H)$ such that for all $s, t \in K$, $R_H(s,t) \leq R_G(s,t) \leq \alpha \cdot R_H(s,t)$.

The lemma below relates the Schur Complement and resistance sparsifiers.

▶ **Lemma 14.** *Let $G = (V, E, \mathbf{w})$ be a weighted graph with $K \subset V$, Laplacian matrix $\mathbf{L}$ and Schur Complement $\mathbf{L}_S^K$ (with respect to the terminal set K). Then the graph $H = (K, E_H, \mathbf{w}_H)$ associated with the Laplacian $\mathbf{L}_S^K$ is a 1-vertex resistance sparsifier of G with respect to K.*

Proof. Fix some terminal pair (s,t) and consider the vectors $\chi_{s,t}$ and $\chi'_{s,t}$ of dimension n and k, respectively. Lemma 12, the definition of effective resistance and the fact that $\chi_{s,t}$ and $\chi'_{s,t}$ are valid vectors for $\mathbf{L}$ and $\mathbf{L}_S^K$ give: $R_G(s,t) = \chi_{s,t}^T \mathbf{L}^\dagger \chi_{s,t} = \chi'^T_{s,t} \mathbf{L}_S^{K\dagger} \chi'_{s,t} = R_H(s,t)$. ◀

Approximate Schur Complement. We need the following lemma due to Durfee et al. [13].

▶ **Lemma 15** ([13])**.** *Fix $\varepsilon \in (0, 1/2)$ and $\delta \in (0,1)$. Let $G = (V, E, \mathbf{w})$ be a weighted graph with n vertices, m edges. Let $K \subset V$ with $|K| = k$. Let $\mathbf{L}$ be the Laplacian of G and $\mathbf{L}_S^K$ be the corresponding Schur complement with respect to K. Then there is an algorithm* ApproxSchur$(G, K, \varepsilon, \delta)$ *that returns a Laplacian matrix $\widetilde{\mathbf{L}}_S^K$ with associated graph $\widetilde{H}$ on the terminals K such that the following statements hold with probability at least $1 - \delta$:*

1. *The graph $\widetilde{H}$ has $O(k\varepsilon^{-2}\log(n/\delta))$ edges.*
2. *$\mathbf{L}_S^K$ and $\widetilde{\mathbf{L}}_S^K$ are spectrally close, that is*

$$\forall \mathbf{x} \in \mathbb{R}^k \quad (1-\varepsilon)\mathbf{x}^T \mathbf{L}_S^K \mathbf{x} \leq \mathbf{x}^T \widetilde{\mathbf{L}}_S^K \mathbf{x} \leq (1+\varepsilon)\mathbf{x}^T \mathbf{L}_S^K \mathbf{x}.$$

The total running time for producing $\widetilde{H}$ is $\tilde{O}((n+m)\varepsilon^{-2}\log^4(n/\delta))$.

In the following, we call the Laplacian $\widetilde{\mathbf{L}}_S^K$ (or equivalently, the graph $\widetilde{H}$) satisfying the above two conditions an *approximate Schur Complement* of G with respect to K. Note that by definition, the graph $\widetilde{H}$ is a $(1 \pm \varepsilon)$-spectral sparsifier of the graph H that is associated with graph $\mathbf{L}_S^K$, which in turn is a 1-vertex resistance sparsifier of G with respect to K. Therefore, $\widetilde{H}$ is a $(1 \pm O(\varepsilon))$-vertex resistance sparsifier of G with respect to K (see Section 2).

3.2 Proof of Theorem 1

We now present a fully dynamic algorithm for maintaining the energy of electrical flows up to a $(1+\varepsilon)$ factor in minor-free graphs and prove Theorem 1. We start with the special case of planar graphs.

Data Structure. In our dynamic algorithm, we will maintain an r-division $\mathcal{P} = \{P_1, \cdots, P_\ell\}$ of G with $\ell = O(n/r)$ and for each region P_i, we compute a graph $\widetilde{H}_i$ by invoking the algorithm ApproxSchur in Lemma 15 with parameters P_i, $K = \partial_G(P_i)$, $\varepsilon = \frac{\varepsilon}{6}$ and $\delta = 1/n^3$.

Let $\mathcal{D}(G)$ denote such a data structure for G, and let $T_{\mathcal{D}(G)}$ denote the time to compute $\mathcal{D}(G)$. Note that by Lemma 5 and 15, $T_{\mathcal{D}(G)} = \tilde{O}(n + \frac{n}{r} \cdot r\varepsilon^{-2}) = \tilde{O}(n\varepsilon^{-2})$. Furthermore, note that there are at most $O(n/r)$ regions, and for each such a region P_i, the corresponding graph $\widetilde{H}_i$ is *not* an approximate Schur Complement of P_i with respect to its boundary $\partial_G(P_i)$ with probability at most $1/n^3$. Therefore, by the union bound, with probability at least $1 - n \cdot \frac{1}{n^3} = 1 - \frac{1}{n^2}$, for any $i \leq \ell$, the graph $\widetilde{H}_i$ is an approximate Schur Complement of P_i

with respect to $\partial_G(P_i)$, and thus a $(1 \pm \frac{\varepsilon}{6})$-spectral sparsifier of H_i, where H_i denotes the exact Schur complement of P_i with respect to $\partial_G(P_i)$. In the following, we will condition on this event. This data structure $\mathcal{D}(G)$ will be recomputed every $T_{\text{div}} := \Theta(n/r)$ operations.

Handling Edge Insertions/Deletions. We now describe the INSERT operation. Whenever we compute an approximate Schur Complement, we assume that the procedure APPROXSCHUR from Lemma 15 is invoked on the corresponding region and its boundary vertex set, with $\varepsilon = \frac{\varepsilon}{6}$ and error probability $\delta = 1/n^3$. Let us consider inserting an edge $e = (x, y)$.

- If both x, y belong to the same region, say P_i, then we add the edge e to P_i, and recompute an approximate Schur Complement $\widetilde{H}_i$ of the region P_i (with respect to its boundary vertex set) from scratch.
- If x and y do not belong to the same region, we do the following.
  - If x is an interior vertex of some region P_x, then adding an edge (x, y) will make x a boundary vertex. We then recompute an approximate Schur Complement $\widetilde{H}_x$ of P_x.
  - If y is an interior vertex of some region, then we handle it in the same way as we did for the interior vertex x.
  - We treat the edge (x, y) as a new region containing only this edge.

Observe that for each insertion, the number of vertices in any region is always at most r, and we perform only a constant number of calls to APPROXSCHUR, Lemma 15 implies that the time to handle an edge insertion is $\tilde{O}(r\varepsilon^{-2})$. Furthermore, since each edge insertion may increase by a constant the number of boundary nodes and the total number of regions.

We now describe the DELETE operation. If we delete some edge $e = (x, y)$, let P_i be the region such that both $x, y \in P_i$. We remove the edge from P_i, and then recompute an approximate Schur Complement $\widetilde{H}_i$ of P_i with respect to its boundary. By Lemma 15, the cost of this resparsification step is bounded by $\tilde{O}(r\varepsilon^{-2})$.

Since we recompute the data structure every $\Theta(n/r)$ operations, the amortized update time is $\tilde{O}\left(\frac{n\varepsilon^{-2}}{n/r} + r\varepsilon^{-2}\right) = \tilde{O}(r\varepsilon^{-2})$.

Handing Queries. In order to return a $(1+\varepsilon)$-approximation of the energy of $s-t$ electrical flow for an ELECTRICALFLOW(s, t) query, it suffices to return a $(1 - \frac{\varepsilon}{2})$-approximation of the effective $s - t$ resistance, for which we first need to review the static algorithm for computing effective resistance. The following result is due to Durfee et al. [13] (which builds and/or improves upon [10, 28, 39]).

▶ **Theorem 16** ([13]). *Fix $\varepsilon \in (0, 1/2)$ and let $G = (V, E, \mathbf{w})$ be a weighted graph with n vertices and m edges. There is an algorithm* EFFECTIVERESISTANCE *that computes a value ψ such that $(1-\varepsilon)R_G(s,t) \le \psi \le (1+\varepsilon)R_G(s,t)$, in time $\tilde{O}(m + \frac{n}{\varepsilon^2})$ with high probability.*

To answer the query ELECTRICALFLOW(s, t), we will form a smaller auxiliary graph that is the union of the regions containing s, t and the approximate Schur Complements of the remaining regions with respect to their boundaries, and output the approximate effective $s - t$ resistance of the smaller graph. More precisely, let P_s and P_t be two regions that contain s and t, respectively. Let $\mathcal{J}$ denote the index set of all the remaining regions, i.e., $\mathcal{J} = \{i : P_i \in \mathcal{P} \setminus \{P_s, P_t\}\}$. For each region P_i such that $i \in \mathcal{J}$, as before, let $\widetilde{H}_i$ be the approximate Schur Complement of P_i that we have maintained. Now we form an auxiliary graph H by taking the union over the regions P_s and P_t and all the approximate Schur Complements of the remaining regions, i.e., $H = P_s \cup P_t \cup \bigcup_{i \in \mathcal{J}} \widetilde{H}_i$. We then run the algorithm EFFECTIVERESISTANCE on H with $\varepsilon = \frac{\varepsilon}{6}$ to obtain an estimator ψ and return $c_H(s, t) := (1 - \frac{\varepsilon}{6})\psi$. Next we show that the returned value is a good approximation to the actual effective resistance.

▶ **Lemma 17.** *Fix $\varepsilon \in (0,1)$. Let $G = (V, E, \mathbf{w})$ be some current graph and $s, t \in V$. Further, let $H = P_s \cup P_t \cup \bigcup_{i \in \mathcal{J}} \widetilde{H}_i$ be defined as above and let $c_H(s,t)$ be the value returned as above by invoking* EffectiveResistance *on H. Then, with high probability, we get*

$$(1 - \frac{\varepsilon}{2})R_G(s,t) \leq c_H(s,t) \leq (1 - \frac{\varepsilon}{2})R_G(s,t).$$

Proof. For the sake of analysis, we divide the sequence of updates into intervals each consisting of $T_{\text{div}} = \Theta(n/r)$ operations. Let I be the interval in which the query is made. Let $G^{(0)}$ denote the graph at the beginning of I. We compute the data structure $\mathcal{D}(G^{(0)})$ of $G^{(0)}$, which contains an r-division $\mathcal{P}^{(0)}$ and the corresponding approximate Schur Complements $\widetilde{H}_i^{(0)}$. As mentioned before, with probability at least $1 - \frac{1}{n^2}$, each of the graphs $\widetilde{H}_i^{(0)}$ will be a $(1 \pm \frac{\varepsilon}{6})$-spectral sparsifier of the exact Schur Complement $H_i^{(0)}$of the corresponding region with respect to its boundary vertex set.

Let G be the current graph when the query is made, which is formed from $G^{(0)}$ after some updates in I. Let $\mathcal{P} = \{P_i\}_i, \widetilde{H}_i, 1 \leq i \leq O(n/r)$ be the r-division and the approximate Schur Complements in the current data structure, respectively. Let H_i denote the exact Schur Complement of the region P_i with respect to its boundary vertex set. Since the total number of updates in I is $\Theta(n/r)$, and each update only involves a constant number of invocations of ApproxSchur with error probability $1/n^3$ that recomputes the approximate Schur Complements of some regions, we have that with probability at least $1 - O(n/r) \cdot \frac{1}{n^3} \geq 1 - \frac{1}{n^2}$, these recomputed approximate Schur Complements are $(1 \pm \frac{\varepsilon}{6})$-spectral sparsifiers of the corresponding exact Schur Complements. Therefore, for the current graph G and its data structure, with probability $1 - 2 \cdot \frac{1}{n^2} = 1 - \frac{2}{n^2}$, for all i, the graph $\widetilde{H}_i$ is a $(1 \pm \frac{\varepsilon}{6})$-spectral sparsifier of H_i. In the following, we will condition on this event.

Recall that P_s and P_t are two regions that contain s and t, respectively. Consider the graph $G' = P_s \cup P_t \cup \bigcup_{i \in \mathcal{J}} H_i$. We have the following lemma whose proof is deferred to the full version of the paper.

▶ **Lemma 18.** *For any two vertices $u, v \in V(G')$, it holds that $R_G(u,v) = R_{G'}(u,v)$.*

It follows from the above lemma that $R_G(s,t) = R_{G'}(s,t)$. We next argue that H is a $(1 \pm \frac{\varepsilon}{6})$-resistance sparsifier to G' with high probability. First, note that each of the subgraphs P_s, P_t, H_i and $\widetilde{H}_i$, $i \in \mathcal{J}$ can be treated as graphs defined on the same vertex set $V(G')$ with appropriate isolated vertices. Second, since for each $i \in \mathcal{J}$, $\widetilde{H}_i$ is $(1 \pm \frac{\varepsilon}{6})$-spectral sparsifier of H_i, and P_s, P_t are sparsifiers of itself, we know that by Lemma 10 about the decomposability of sparsifiers, H is a $(1 \pm \frac{\varepsilon}{6})$-spectral sparsifier of G'. Since every $(1 \pm \frac{\varepsilon}{6})$-spectral sparsifier is a $(1 \pm \frac{\varepsilon}{6})$-resistance sparsifier, it holds that

$$(1 - \frac{\varepsilon}{6})R_H(s,t) \leq R_{G'}(s,t) \leq (1 + \frac{\varepsilon}{6})R_H(s,t). \tag{1}$$

Since by definition we have $c_H(s,t) := (1 - \frac{\varepsilon}{6})\psi$, Theorem 16 implies that

$$(1 - \frac{\varepsilon}{6})^2 R_H(s,t) \leq (1 - \frac{\varepsilon}{6})\psi \leq (1 - \frac{\varepsilon}{6})(1 + \frac{\varepsilon}{6})R_H(s,t), \tag{2}$$

with high probability. Combining (1) and (2) we get

$$\frac{(1 - \frac{\varepsilon}{6})^2}{(1 + \frac{\varepsilon}{6})} R_{G'}(s,t) \leq (1 - \frac{\varepsilon}{6})\psi \leq (1 + \frac{\varepsilon}{6})R_{G'}(s,t),$$

which in turn along with $R_G(s,t) = R_{G'}(s,t)$ imply that,

$$(1 - \frac{\varepsilon}{2})R_G(s,t) \leq (1 - \frac{\varepsilon}{6})\psi \leq (1 + \frac{\varepsilon}{2})R_G(s,t).$$

Therefore, with high probability, the algorithm outputs a $(1-\frac{\varepsilon}{2})$-approximation to the effective $s-t$ resistance. ◀

To bound the query time, we need to bound the size of the $H = P_s \cup P_t \cup \bigcup_{i\in\mathcal{J}} \widetilde{H}_i$. As in the proof of Lemma 17, we let $G^{(0)}$ denote the graph right after the last rebuilding of the data structure. Let $\mathcal{P}^{(0)}$ denote the corresponding r-division. By definition, for each $P \in \mathcal{P}^{(0)}$, $|P| \leq r$ and the size of all the boundary vertices is $c_2 n/\sqrt{r}$. By Lemma 6, we have that $\sum_{P\in\mathcal{P}^{(0)}} |\partial_{G^{(0)}}(P)| \leq O(n/\sqrt{r})$, i.e., the sum of the numbers of boundary vertices over all regions of $G^{(0)}$ is at most $O(n/\sqrt{r})$.

Note that there will be at most $T_{\text{div}} = \Theta(n/r)$ updates between $G^{(0)}$ and G, the graph to which the query is performed, and each update can only increase the number of boundary vertices and the total number of regions by a constant. These facts imply that the size of all boundary nodes is $O(n/\sqrt{r})$. Therefore, we have that $|V(H)| \leq O(r + n/\sqrt{r})$, and that the sum of the numbers of boundary vertices of the regions of G is at most $O(n/\sqrt{r})$, i.e., $\sum_i |V(\widetilde{H}_i)| \leq O(n/\sqrt{r})$.

On the other hand, by Lemma 15, for each i, $|E(\widetilde{H}_i)| = O(|V(\widetilde{H}_i)| \cdot \varepsilon^{-2}\log n)$. Thus,

$$\begin{aligned} |E(H)| &\leq |E(P_s)| + |E(P_t)| + \sum_i |E(\widetilde{H}_i)| \leq O(r) + \sum_i |V(\widetilde{H}_i)| \cdot O(\varepsilon^{-2}\log n) \\ &= O((r + n/\sqrt{r})\varepsilon^{-2}\log n). \end{aligned}$$

By Theorem 16, it follows that the worst-case query time is $\tilde{O}((r + n/\sqrt{r})\varepsilon^{-2})$.

To achieve asymptotically the same worst-case update time, we use a standard global rebuilding technique (see the full version for details), which then finishes the proof of Theorem 1 when the input graph is planar.

Extension to Minor-Free Graphs. In the following, we briefly discuss how one can adapt the previous dynamic algorithms for planar graphs to minor-free graphs.

The key observation is that since the approximate Schur Complement can be constructed in nearly-linear time for any graph, it suffices for us to efficiently maintain an r-division of any minor-free graph, i.e., we need fast algorithms for computing a *separator* of order $\sqrt{n}$ in such graphs. (A separator is a subset S of vertices whose deletion will partition the graph into connected components, each of size at most $\frac{2n}{3}$). Kawarabayashi and Reed [21] showed that for any K_t-minor-free graph G, one can construct in $O(n^{1+\xi})$ time a separator of size $O(\sqrt{n})$ for G, for any constant $\xi > 0$ and constant t. (The $O(\cdot)$ notation for the running time hides huge dependency on t.) Applying this separator construction recursively as in Frederickson's algorithm [15], we can maintain an r-division of any K_t-minor-free G in $\tilde{O}(n^{1+\xi})$ time. Furthermore, by analysis in [15], it holds that the total sum of the sizes of all boundary vertex sets is also bounded by $O(\frac{n}{\sqrt{r}})$ as guaranteed by Lemma 6 for planar graphs.

Now we can dynamically maintain the data structure for electrical flows in minor-free graphs almost the same as we did for planar graphs, except that we use the above $\tilde{O}(n^{1+\xi})$ time algorithm to compute the r-divisions. Thus, the time to compute the data structure for any minor-free graph is then $\tilde{O}(n^{1+\xi} + n\varepsilon^{-2})$, for arbitrarily small constant $\xi > 0$. Then from previous analysis, the worst-case update time is $\tilde{O}(n^{\xi} r\varepsilon^{-2})$ update time, and the worst-case query time is $\tilde{O}((r + n/\sqrt{r})\varepsilon^{-2})$. This completes the proof of Theorem 1.

References

1 Amir Abboud and Søren Dahlgaard. Popular conjectures as a barrier for dynamic planar graph algorithms. In *Proc. of the 57th FOCS*, pages 477–486, 2016.

2 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. of the 55th FOCS*, pages 434–443, 2014.

3 Ittai Abraham, Shiri Chechik, and Cyril Gavoille. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In *Proc. of the 44th STOC*, pages 1199–1218, 2012.

4 Ittai Abraham, Shiri Chechik, and Kunal Talwar. Fully dynamic all-pairs shortest paths: Breaking the $o(n)$ barrier. In *Proc. of the 17th APPROX*, 2014.

5 Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On fully dynamic graph sparsifiers. In *Proc. of the 57th FOCS*, pages 335–344, 2016.

6 Noga Alon and Raphael Yuster. Solving linear systems through nested dissection. In *Proc. of the 51st FOCS*, pages 225–234, 2010.

7 Alexandr Andoni, Anupam Gupta, and Robert Krauthgamer. Towards $(1+\varepsilon)$-approximate flow sparsifiers. In *Proc. of the 25th SODA*, pages 279–293, 2014.

8 Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

9 Aaron Bernstein. Fully dynamic $(2+\epsilon)$ approximate all-pairs shortest paths with fast query and close to linear update time. In *Proc. of the 50th FOCS*, pages 693–702, 2009.

10 Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proc. of the 43rd STOC*, pages 273–282, 2011.

11 Julia Chuzhoy. Routing in undirected graphs with constant congestion. In *Proc. of the 44th STOC*, pages 855–874. ACM, 2012.

12 Peter G Doyle and J Laurie Snell. *Random Walks and Electric Networks*. Carus Mathematical Monographs. Mathematical Association of America, 1984.

13 David Durfee, Rasmus Kyng, John Peebles, Anup B. Rao, and Sushant Sachdeva. Sampling random spanning trees faster than matrix multiplication. In *Proc. of the 49th STOC*, pages 730–742, 2017.

14 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification. i. planary testing and minimum spanning trees. *J. Comput. Syst. Sci.*, 52(1):3–27, 1996.

15 Greg N Federickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

16 Zvi Galil, Giuseppe F. Italiano, and Neil Sarnak. Fully dynamic planarity testing with applications. *J. ACM*, 46(1):28–91, 1999.

17 MT Goodrich. Planar separators and parallel polygon triangulation. *Journal of Computer and System Sciences*, 3(51):374–389, 1995.

18 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proc. of the 47th STOC*, pages 21–30, 2015.

19 Giuseppe F. Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *Proc. of the 43rd STOC*, pages 313–322, 2011.

20 Michael Kapralov, Yin Tat Lee, Cameron Musco, Christopher Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. In *Proc. of the 55th FOCS*, pages 561–570, 2014.

21 Ken-ichi Kawarabayashi and Bruce Reed. A separator theorem in minor-closed classes. In *Proc. of the 51st FOCS*, pages 153–162. IEEE, 2010.

22 Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proc. of the 25th SODA*, pages 217–226, 2014.

23 Jonathan A. Kelner and Alex Levin. Spectral sparsification in the semi-streaming setting. *Theory Comput. Syst.*, 53(2):243–262, 2013.

24 Richard Kenyon. The laplacian on planar graphs and graphs on surfaces. *Current Developments in Mathematics*, 2011.

25 Philip N Klein, Shay Mozes, and Christian Sommer. Structured recursive separator decompositions for planar graphs in linear time. In *Proc. of the 45th STOC*, pages 505–514, 2013.

26 Philip N. Klein and Sairam Subramanian. A fully dynamic approximation scheme for shortest paths in planar graphs. *Algorithmica*, 22(3):235–249, 1998.

27 Ioannis Koutis and Gary L. Miller. A linear work, $o(n^{1/6})$ time, parallel algorithm for solving planar laplacians. In *Proc. of the 18th SODA*, pages 1002–1011, 2007.

28 Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching optimality for solving SDD linear systems. *SIAM J. Comput.*, 43(1):337–354, 2014.

29 Robert Krauthgamer, Huy L Nguyen, and Tamar Zondiner. Preserving terminal distances using minors. *SIAM Journal on Discrete Mathematics*, 28(1):127–141, 2014.

30 Rasmus Kyng and Sushant Sachdeva. Approximate gaussian elimination for laplacians-fast, sparse, and simple. In *Proc. of the 57th FOCS*, pages 573–582, 2016.

31 Yin Tat Lee, Satish Rao, and Nikhil Srivastava. A new approach to computing maximum flows using electrical flows. In *Proc. of the 45th STOC*, pages 755–764, 2013.

32 Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, 1979.

33 Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *Proc. of the 54th FOCS*, pages 253–262, 2013.

34 Aleksander Madry. Computing maximum flow with augmenting electrical flows. In *Proc. of the 57th FOCS*, pages 593–602, 2016.

35 Gary L. Miller and Richard Peng. Approximate maximum flow on separable undirected graphs. In *Proc. of the 24th SODA*, pages 1151–1170, 2013.

36 Ankur Moitra. Approximation algorithms for multicommodity-type problems with guarantees independent of the graph size. In *Proc. of the 50th FOCS*, pages 3–12, 2009.

37 Richard Peng. Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time. In *Proc. of the 27th SODA*, pages 1862–1867, 2016.

38 Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM J. Comput.*, 40(6):1913–1926, 2011.

39 Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM J. Comput.*, 40(4):981–1025, 2011.

40 Daniel A. Spielman and Shang-Hua Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM J. Matrix Analysis Applications*, 35(3):835–885, 2014.

41 Sairam Subramanian. A fully dynamic data structure for reachability in planar digraphs. In *Proc. of the 1st ESA*, pages 372–383, 1993.

42 Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proc. of the 44th STOC*, pages 887–898, 2012.

Single-Sink Fractionally Subadditive Network Design*

Guru Guruganesh[†1], **Jennifer Iglesias**[‡2], **R. Ravi**[3], **and Laura Sanità**[4]

1 **Carnegie Mellon University, Pittsburgh, PA, USA**
ggurugan@andrew.cmu.edu
2 **Carnegie Mellon University, Pittsburgh, PA, USA**
jiglesia@andrew.cmu.edu
3 **Carnegie Mellon University, Pittsburgh, PA, USA**
ravi@andrew.cmu.edu
4 **University of Waterloo, Waterloo, Canada**
laura.sanita@uwaterloo.ca

Abstract

We study a generalization of the Steiner tree problem, where we are given a weighted network G together with a collection of k subsets of its vertices and a root r. We wish to construct a minimum cost network such that the network supports one unit of flow to the root from every node in a subset simultaneously. The network constructed does not need to support flows from all the subsets simultaneously.

We settle an open question regarding the complexity of this problem for $k = 2$, and give a $\frac{3}{2}$-approximation algorithm that improves over a (trivial) known 2-approximation. Furthermore, we prove some structural results that prevent many well-known techniques from doing better than the known $O(\log n)$-approximation. Despite these obstacles, we conjecture that this problem should have an $O(1)$-approximation. We also give an approximation result for a variant of the problem where the solution is required to be a path.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases Network design, single-commodity flow, approximation algorithms, Steiner tree

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.46

1 Introduction

We study a robust version of a single-sink network design problem that we call the *Single-sink fractionally-subadditive network design* (f-SAND) problem. In an instance of f-SAND, we are given an undirected graph $G = (V, E)$ with edge costs $w_e \geq 0$ for all $e \in E$, a root node $r \in V$, and k *colors* represented as vertex subsets $C_i \subseteq V \setminus \{r\}$ for all $i \in [k]$, that wish to send flow to r. A feasible solution is an integer capacity installation on the edges of G, such that for every $i \in [k]$, each node in C_i can *simultaneously* send one unit of flow to r. Thus,

* Full version available at: https://arxiv.org/abs/1707.01487.

† This material is based upon work supported in part by National Science Foundation awards CCF-1319811,CCF-1536002, and CCF-1617790.

‡ This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 2013170941.

the total flow sent by color i nodes is is $|C_i|$ while the flows sent from nodes of different colors are instead *non-simultaneous* and can share capacity. An optimal solution is a feasible one that minimizes the total cost of the installation.

The single-sink nature of the problem suggests a natural *cut*-covering formulation, namely:

$$\begin{aligned} \min \sum_{e\in E} w_e x_e \quad & \text{s.t.} \\ \sum_{e\in\delta(S)} x_e \geq f(S) \qquad & \forall S \subset V \setminus \{r\} \\ x \geq 0\,, \quad x \in \mathbb{Z}\,, & \end{aligned} \tag{IP}$$

where $\delta(S)$ denotes the set of edges with exactly one endpoint in S, and

$$f(S) := \max_{i\in[k]}\{|C_i \cap S|\} \tag{1}$$

for all $S \subseteq V \setminus \{r\}$. Despite having exponentially many constraints, the LP-relaxation of (IP) can be solved in polynomial-time because the separation problem reduces to performing k max-flow computations. The main challenge is to round the resulting solution into an integer solution.

Rounding algorithms for the LP relaxation of (IP) have been investigated by many authors, under certain assumptions of the function $f(S)$. Prominent examples are some classes of 0/1-functions (such as *uncrossable* functions), or integer-valued functions such as *proper* functions, or *weakly supermodular* functions [12, 19]; however, these papers consider arbitrary cut requirements rather than the single-sink connectivity requirements we study.

Our single-sink problem is a special case of a broader class of subadditive network design problems where the function f is allowed to be a general subadditive function. Despite their generality, the single-sink network design problem for general subadditive functions can be approximated within an $O(\log |V|)$ factor by using a tree drawn from the probabilistic tree decomposition of the metric induced by G using the results of Fakcharoenphol, Rao, and Talwar [8], and installing the required capacity on the tree edges. Hence, a natural direction is to consider special cases of such subadditive cut requirement functions.

Our function $f(S)$ defined in (1) is an interesting and important special case of subadditive functions. It was introduced as XOS-functions (max-of-sum functions) in the context of combinatorial auctions by Lehman et al. [20]. Feige [9] proved that this function is equivalent to fractionally-subadditive functions which are a strict generalization of submodular functions (hence the title). These functions have been extensively studied in the context of learning theory and algorithmic game theory [1, 3, 20]. Our work is an attempt to understand their behavior as single-sink network design requirement functions.

f-SAND was first studied by Oriolo et al. [21] in the context of *robust* network design, where the goal is to install minimum cost capacity on a network in order to satisfy a given set of (non-simultaneous) traffic demands among terminal nodes. Each subset C_i can in fact be seen as a way to specify a distinct traffic demand that the network would like to support. They observed that f-SAND generalizes the Steiner tree problem: an instance of the Steiner tree problem with $k+1$ terminals $t_1, \ldots, t_{k+1}$ is equivalent to the f-SAND instance with $r := t_{k+1}$ and $C_i := \{t_i\}$ for all $i \in [k]$. This immediately shows that f-SAND is NP-hard (in fact, APX-hard [6]) when k is part of the input. The authors in [21] strengthened the hardness result by proving that f-SAND is NP-hard even if k is not part of the input, and in particular for $k = 3$ (if $k = 1$ the problem is trivially solvable in polynomial-time by computing a shortest path tree rooted at r). From the positive side, they observed that there is a trivial

k-approximation algorithm, that relies on routing via shortest paths, and an $O(\log |\cup_i C_i|)$-approximation algorithm using metric embeddings [8, 17]. The authors conclude their paper mentioning two open questions, namely whether the problem is polynomial-time solvable for $k = 2$, and whether there exists an $O(1)$-approximation algorithm.

1.1 Our results

1. In this paper, we answer the first open question in [21] by showing that f-SAND is NP-hard for $k = 2$ via a reduction from SAT.
2. We give a $\frac{3}{2}$-approximation algorithm for this case ($k = 2$). This is the first improvement over the (trivial) k-approximation obtained using shortest paths for any k. Our approximation algorithm is based on pairing terminals of different groups together, and therefore reducing to a suitable minimum cost matching problem. While the idea behind the algorithm is natural, its analysis requires a deeper understanding of the structure of the optimal solution.
3. We also introduce an interesting variant of f-SAND, which we call the Latency-f-SAND problem, where the network built is restricted to being a *path* with the root r being one of the endpoints (f-SAND-path). We show a $O(\log^2 k \log n)$-approximation using a new reformulation of the problem that allows us to exploit techniques recently developed for *latency* problems [4].
4. While being a generalization of well-studied problems, f-SAND does not seem to admit an easy $O(1)$-approximation via standard LP-rounding techniques for arbitrary values of k. We prove some structural results that highlight the difficulty of the general problem (see full version [18]). In particular, we show a family of a instances providing a super-constant gap between an optimal f-SAND solution and an optimal *tree*-solution, i.e., a solution whose support is a tree – this rules out many methods that output a solution with a tree structure. The bulk of the construction was shown in [13] and we amend it to our problem using a simple observation. Furthermore, we give some evidence that an iterative rounding approach (as in Jain's fundamental work [19]) is unlikely to work. This follows by considering a special class of Kneser Graphs, where the LP seems to put low fractional weight on each edge in an extreme point.
5. **Open Questions.** We offer the following conjecture as our main open question.[1]
   ▶ **Conjecture 1.** *There exists an $O(1)$-approximation algorithm for the* f-SAND *problem.*
   Although standard LP-based approaches seem to fail in providing a constant factor approximation, the worst known integrality gap example we are aware of yields a (trivial) lower bound of 2 on the integrality gap of (IP) for f-SAND. A related open question is if there is an instance of f-SAND for which the integrality gap of (IP) is greater than 2.

1.2 Related work

Network design problems where the goal is to build a minimum cost network in order to support a given set of flow demands, have been extensively studied in the literature (we refer to the survey [5]). There has been a huge amount of research focusing on the case the set of demands is described via a polyhedron (see e.g. [2]). In this context a very popular model is the *Virtual private network* [7, 11], for which many approximation results have been developed (see e.g. [14, 15, 16] and the references therein). For the case where the set of

[1] Although the problem is known in some circles, it has not been explicitly stated as a conjecture. We do so here, in the hopes that it will encourage others to work on this problem.

demands is instead given as a (finite) discrete list, the authors in [21] developed a constant factor approximation algorithm on ring networks, and proved that f-SAND is polynomial-time solvable on ring networks.
Regarding the formulation (IP), Goemans and Williamson [12] gave a $O(\log(f_{max}))$-approximation algorithm for solving (IP) whenever $f(S)$ is an integer-valued proper function that can take values up to f_{max}, based on a primal-dual approach. Subsequently, Jain [19] improved this result by giving a 2-approximation algorithm using iterative rounding of the LP-relaxation. Recently, a strongly-polynomial time FPTAS to solve the LP-relaxation of (IP) with proper functions has been given in [10].

2 3/2-approximation for the two color case

The goal of this section is to give a $\frac{3}{2}$-approximation algorithm for SAND with two colors. We remark that our algorithm bypasses the difficulties mentioned in the previous section. In particular, the final output is not a tree.

2.1 Simplifying Assumptions

We will refer to the two colors as *green* and *blue*, and let $C_G \subset V$ denote the set of green terminals, and $C_B \subset V$ denote the set of blue terminals. Without loss of generality, we will assume that $|C_G| = |C_B|$, i.e., the cardinality of green terminals is equal to the cardinality of blue terminals (if not, we could easily add dummy nodes at distance 0 from the root). Furthermore, by replacing each edge in the original graph with $|C_G|$ parallel edges of the same cost, we can assume that in a feasible solution the capacity installed on each edge must be either 0 or 1. This means that each edge is used by *at most* one terminal of C_G (resp. C_B) to carry flow to the root. Lastly, we assume that every terminal in C_G shares at least one edge with some terminal in C_B in the optimal solution.[2]

Let OPT denote an optimal solution to a given instance of SAND with two colors. We start by developing some results on the structure of OPT, that will be crucial to analyze our approximation algorithm later.

2.2 Understanding the structure of OPT

A feasible solution of a SAND instance consists of a (integer valued) capacity installation on the edges that allows for a flow from the terminals to the root. Given a feasible solution, each terminal will send its unit of flow to t on a single path. Let us call the collection of such paths a *routing* associated with the feasible solution. The first important concept we need is the concept of splits.

2.2.1 Shared Edges and Splits

Given a routing, for each terminal $g \in C_G$ (and $b \in C_B$ respectively) let P_g (P_b) denote the path along which g (b) sends flow to the root; i.e. $P_g := \{g = x_0, x_1, \ldots, x_{|P_g|} = r\}$. We say that an edge e is **shared** if the paths of two terminals of different color contain the edge. We say that $g \in C_G$ and $b \in C_B$ are **partners** with respect to a shared edge $e = uv$, if their respective paths use the edge e; i.e. $e \in P_g \cap P_b$.

[2] We can easily ensure this e.g. by modifying our instance as follows: we add a dummy node r' which is only connected to r with $|C_G|$ parallel edges of 0 cost, and we make r' be the new root. In this way, all terminals will use one copy of the edge (r, r').

▶ **Definition 2.** A **split** in the path P_g is a maximal set of consecutive edges of the path such that g is partnered with some b on all the edges of this set.

If $\{(x_i, x_{i+1}), (x_{i+1}, x_{i+2}), \ldots, (x_{i+j-1}, x_{i+j})\}$ is a split in the path $P_g = \{g = x_0, x_1, \ldots, x_{|P_g|} = r\}$ for $g \in C_G$, then there exists a unique terminal $b \in C_B$ such that P_b contains the edges $\{(x_i, x_{i+1}), (x_{i+1}, x_{i+2}), \ldots, (x_{i+j-1}, x_{i+j})\}$, P_b does not contain the edge (x_{i-1}, x_i), and if $x_{i+j} \neq r$ then P_b does not contain the edge (x_{i+j}, x_{i+j+1}). By our assumptions, the terminal b is unique as each edge is used by at most one terminal of each color.

Since the flow is going from a terminal g to r, the path P_g naturally induces an orientation on its edges given by the direction of the flow, even though the edges are undirected. Of course, the paths of different terminals could potentially induce opposite orientations on (some of) the shared edges (see Figure 1).

▶ **Definition 3.** A split is **wide**, if the paths of the two terminals that are partners on the edges of the split induce opposite orientations on the edges. A split is **thin**, if the paths of the two terminals that are partners on the edges of the split induce the same orientation on the edges.

The above notions are well defined for any routing with respect to a feasible solution. Now, we focus on the structure of an optimal routing, i.e., a routing with respect to an optimal solution. For the rest of this section, we let $\{P_g\}_{g \in C_G}$ and $\{P_b\}_{b \in C_B}$ be an optimal routing. The following lemma is immediate.

▶ **Lemma 4.** *Let $\{(x_i, x_{i+1}), (x_{i+1}, x_{i+2}), \ldots, (x_{i+j-1}, x_{i+j})\}$ be a split in the path P_g (for some $g \in C_G$). The edges of the split form a shortest path from x_i to x_{i+j}.*

Proof. If not, we could replace this set of edges with the set of edges of a shortest path from x_i to x_{i+j}, in both P_g and P_b, where b is the partner of g on the split. Therefore, we can install one unit of capacity on these edges, and remove the unit of capacity from the edges of the split. We get another feasible solution with smaller cost, a contradiction to the optimality of our initial solution. ◀

2.2.2 Split Graph

A consequence of Lemma 4 is each split is entirely characterized by the endpoints of the split and the terminals that share them. We denote each split by a tuple (u, v, g, b) which states that there is a shortest path between u and v whose edges are shared by g and b.

Let $\mathbb{S}$ denote the set of all splits in the optimal routing. We construct a directed graph $G^{\mathbb{S}}$ whose vertex set corresponds to $V = \mathbb{S} \cup C_G \cup C_B$ (i.e. the vertex set contains one vertex for each split and one vertex for each terminal). For each $g \in C_G$, we place a directed *green* edge going between two consecutive splits in P_g. Specifically, if $\{(x_i, x_{i+1}), \ldots, (x_{i+j-1}, x_{i+j})\}$ and $\{(x_{i'}, x_{i'+1}), \ldots, (x_{i'+j'-1}, x_{i'+j'})\}$ are two splits in P_g with $i < i'$, we say that they are consecutive if the subpath from x_{i+j} to $x_{i'}$ does not contain any split. In this case, we place a directed edge in $G^{\mathbb{S}}$ whose tail is the vertex corresponding to the first split, and whose head is the vertex corresponding to the second one. Similarly, for each $b \in C_B$ we place a directed *blue* edge between vertices of consecutive splits that appear in P_b. Furthermore, for each $g \in C_G$ (resp. $b \in C_B$) we place a directed green (resp. blue) edge from g (resp. b) to the vertex corresponding to the first split on the path P_g (resp. P_b), if any. This graph is denoted as the **Split Graph** (see Figure 1).

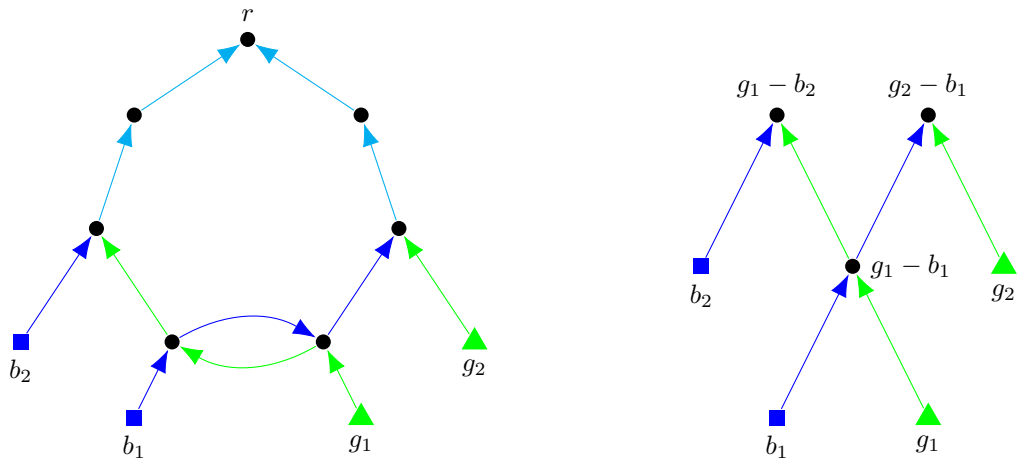


Figure 1 The above left graph (where each undirected edge is supposed to have unit capacity) shows an optimal routing for some f-SAND instance. Note that b_1 and g_2 (resp. b_2 and g_1) send flow to r going counterclockwise (resp. clockwise) on the edges of the cycle. The path P_{b_1} contains two splits: the first is wide (b_1 is partnered with g_1), the second is thin (b_1 is partnered with g_2). The graph on the right is the Split Graph for the optimal solution on the left. The pair of vertices g_1, b_1 and the pair of vertices g_2, b_2 constitute the fresh pairs.

Each split indicates that two terminals of different colors are sharing the capacity on a set of edges in an optimal routing. Hence, each split-vertex in $G^{\mathbb{S}}$ has indegree 2 (in particular, one edge of each color). Furthermore, each split-vertex in $G^{\mathbb{S}}$ has outdegree either 0 or 2; if it has two outgoing edges, one is green and one is blue. Similarly, each terminal has indegree 0, and outdegree 1 (as we assume that each terminal shares at least one edge).

2.2.3 Fresh Pairs

We need one additional definition before proceeding to the algorithm.

▶ **Definition 5.** An **$\mathbb{S}$-alternating sequence** is a sequence of vertices of the Split Graph $\{v, s_1, s_2, \ldots, s_h, w\}$ with $h \geq 1$, that satisfies the following:

(i) (v, s_1) and (w, s_h) are directed edges in $G^{\mathbb{S}}$ and v, w are terminals of different color.

(ii) For all even $i \geq 2$, (s_i, s_{i-1}) and (s_i, s_{i+1}) are both directed edges in $G^{\mathbb{S}}$ with opposite colors.

We call the path obtained by taking the edges in (i) and (ii) an **$\mathbb{S}$-alternating path**. We call (v, w) a **fresh pair** if they are the endpoints of an $\mathbb{S}$-alternating path.

By definition, in an $\mathbb{S}$-alternating sequence the vertices $s_1, \ldots, s_h$ are all split-vertices, and h is odd. We remark here that an $\mathbb{S}$-alternating path is *not* a directed path. (See again Figure 1).

▶ **Lemma 6.** *We can find a set of edge-disjoint $\mathbb{S}$-alternating paths in the Split Graph such that each terminal is the endpoint of exactly one path in this set.*

Proof. We construct the desired set as follows. For each vertex $g \in C_G$, there is a unique outgoing edge to a split vertex $s \in \mathbb{S}$ (as we assume every terminal participates in a split). Since each split-vertex has indegree 2, s has another ingoing edge coming from a different vertex w. If $w \in C_B$, then (v, w) is a fresh pair and we have found an $\mathbb{S}$-alternating sequence $\{v, s, w\}$. If w is a split-vertex, then it has another outgoing edge to a different split-vertex

s', which in its turn has another incoming edge from a different vertex w'. We continue to build an alternating sequence (and a corresponding alternating path) in this way until it terminates in a terminal. Since the path is of even length and the colors alternate, we can conclude that this will terminate in a terminal of opposite color. We remove the edges of this path from the Split Graph, and iterate the process. Each terminal will belong to exactly one $\mathbb{S}$-alternating path, as it has outdegree exactly 1, and all the paths are edge-disjoint, proving the lemma. ◀

2.3 The Algorithm

We are now ready to present our *matching algorithm*. The algorithm has two steps. First, construct a complete bipartite graph $\mathcal{H}$ with the bipartitions C_G and C_B, where the weight on the edge $(g, b) \in C_G \times C_B$ is equal to the cost of the Steiner tree in G connecting g, b and the root. Note that the graph $\mathcal{H}$ can be computed in polynomial time, since a Steiner tree on 3 vertices can be easily computed in polynomial time.

Second, find a minimum-weight perfect matching $\mathcal{M}$ in $\mathcal{H}$, and for each edge $(g, b) \in \mathcal{M}$ install (cumulatively) one unit of capacity on each edge of G that is in the Steiner tree associated to the edge $(g, b) \in \mathcal{M}$. The capacity installation output by this procedure is a feasible solution to f-SAND, and has total cost equal to the weight of $\mathcal{M}$.

▶ **Lemma 7.** *The matching algorithm is a $\frac{3}{2}$-approximation algorithm.*

Proof. First, we partition OPT into four parts; let w_b (and w_g respectively) be the cost of the edges which are used only by blue (green respectively) terminals in OPT, and let w_t (w_d) be the cost of edges in thin (wide) splits in OPT. Thus, $w(OPT) = w_b + w_g + w_t + w_d$. By Lemma 6, we can extract from the Split Graph associated to OPT a set of $\mathbb{S}$-alternating paths such that each terminal is contained in exactly one fresh pair. Consider the matching $\mathcal{M}_1$ determined by the set of fresh pairs found by the aforementioned procedure. We will now bound the weight of $\mathcal{M}_1$.

▶ **Claim 8.** *The weight of the matching formed by connecting the fresh pairs is at most*

$$\frac{3}{2} \cdot w_b + \frac{3}{2} \cdot w_g + 1 \cdot w_t + 3 \cdot w_d.$$

Proof. Let (g, b) be a fresh pair and $(g, s_1, \ldots, s_h, b)$ be the corresponding $\mathbb{S}$-alternating sequence. The edges of the associated $\mathbb{S}$-alternating path naturally correspond to paths in G composed by non-shared edges (that connect either the endpoints of two different splits, or one terminal and one endpoint of a split). These paths together with the edges of the wide splits in the sequence, naturally yield a path $P(b, g)$ in G connecting g and b.

If we do this for all fresh pairs, we obtain that the total cost of the paths $P(b, g)$ is upper bounded by $1 \cdot w_b + 1 \cdot w_g + 2 \cdot w_d$. The reason for having a coefficient of 2 in front of w_d is because the $\mathbb{S}$-alternating paths of Lemma 6 are edge-disjoint, but not necessarily vertex-disjoint: however, since each split-vertex has at most 4 edges incident into it, it can be part of at most 2 $\mathbb{S}$-alternating paths.

Using the aforementioned connection, we can move all terminals in C_G to their partners in C_B. Subsequently, we connect them to the root using the P_b for all $b \in C_B$. This connection to the root will incur a cost of $1 \cdot w_b + 1 \cdot w_d + 1 \cdot w_t$. Combining this together, we get a total cost of $2 \cdot w_b + 1 \cdot w_g + 1 \cdot w_t + 3 \cdot w_d$. Analogously, if we connect the partners in C_G to the root using the the path P_g for all $g \in C_G$, we will incur a total cost of $1 \cdot w_b + 2 \cdot w_g + 1 \cdot w_t + 3 \cdot w_d$.

Since the sum of the cost of the Steiner trees connecting the fresh pairs to the root is no more than either of these two values, we can bound the weight of $\mathcal{M}_1$ by their average:

$$\frac{3}{2} \cdot w_b + \frac{3}{2} \cdot w_g + 1 \cdot w_t + 3 \cdot w_d. \qquad \blacktriangleleft$$

▶ **Claim 9.** *There exists a matching in $\mathcal{H}$ of weight at most $1 \cdot w_b + 1 \cdot w_g + 2 \cdot w_t$.*

Proof. Consider the flow routed on the optimal paths by the set of all terminals $C_G \cup C_B$. We modify the flow (and the corresponding routing) as follows. Whenever two terminals traverse a wide-split, re-route the flows so as to not use the wide-split. This is always possible as the two terminals traverse these edges in opposite directions (by definition of wide splits). This re-routing ensures that all the edges of wide-splits are not used anymore in the resulting paths. However, thin-splits which contained terminals of different colors passing in the same direction, might now contain two terminals of the same color passing through the edges. This means that these edges will be used twice (or must have twice the capacity installed). All other edges do not need to have their capacity changed. Thus, the resulting flow can be associated with a feasible solution of cost at most $1 \cdot w_b + 1 \cdot w_g + 2 \cdot w_t + 0 \cdot w_d$. This flow corresponds to all vertices directly connecting to the root as any shared edge is counted twice. Hence, this is a bound on any matching in $\mathcal{H}$. $\blacktriangleleft$

The average weight of the above matchings is an upper-bound on the minimum weight of a matching in $\mathcal{H}$. Hence, the weight of $\mathcal{M}$ is at most

$$\begin{aligned}&\frac{1}{2} \cdot \left(\frac{3}{2} \cdot w_b + \frac{3}{2} \cdot w_g + 1 \cdot w_t + 3 \cdot w_d\right) + \frac{1}{2} \cdot \left(1 \cdot w_b + 1 \cdot w_g + 2 \cdot w_t + 0 \cdot w_d\right) \\ &\leq \frac{3}{2} \cdot \left(w_b + w_g + w_t + w_d\right)\end{aligned}$$

Therefore, the matching algorithm is $\frac{3}{2}$-approximation algorithm. $\blacktriangleleft$

3 Hardness for two colors

We prove that the SAND problem is NP-hard even with just two colors.

▶ **Theorem 10.** *The SAND problem with 2 colors is NP-hard.*

Proof. We use a reduction from a variant of the Satisfiability (SAT) problem, where each variable can appear in at most 3 clauses, that is known to be NP-hard [22]. Formally, in a SAT instance we are given m clauses $K_1, \ldots, K_m$, and p variables $x_1, \ldots, x_p$. Each clause K_j is a disjunction of some *literals*, where a literal is either a variable x_i or its negation $\bar{x}_i$, for some i in $1, \ldots, p$. The goal is to find a truth assignment for the variables that satisfies all clauses, where a clause is satisfied if at least one of its literals takes value *true*. In the instances under consideration, each variable x_i appears in at most 3 clauses, either as a literal x_i, or as a literal $\bar{x}_i$. It is not difficult to see that, without loss of generality, we can assume that every variable appears in exactly 3 clauses. Furthermore, by possibly replacing all occurrences of x_j with $\bar{x}_j$ and vice versa, we can assume that each variable x_i appears in exactly one clause in its negated form ($\bar{x}_i$).

Given such a SAT instance, we define an instance of SAND as follows (see Fig. 2). We construct a graph $G = (V, E)$ by introducing one sink node r, one node k_j for each clause K_j, and 7 distinct nodes y_i^ℓ, ($\ell = 1, \ldots, 7$), for each variable x_i. That is,

$$V := \{r\} \cup \{k_1, \ldots, k_m\} \cup \left\{ \bigcup_{i=1}^{p} \{y_i^1, y_i^2, y_i^3, y_i^4, y_i^5, y_i^6, y_i^7\} \right\}$$

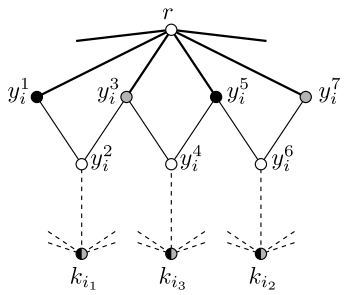


Figure 2 The picture shows the subgraph introduced for every variable x_i. Bold edges have cost 2, solid edges have cost 1, and dashed edges have cost M. Black circles indicate nodes in C_1, and grey circles indicate nodes in C_2. Nodes in $C_1 \cap C_2$ are colored half-black and half-grey.

The set of edges E is the disjoint union of three different sets, $E := E_1 \cup E_2 \cup E_3$, where:

$$E_1 := \bigcup_{i=1}^{p} \left\{ \bigcup_{\ell=1}^{4} \{r, y_i^{2\ell-1}\} \right\}; \quad E_2 := \bigcup_{i=1}^{p} \left\{ \bigcup_{\ell=1}^{6} \{y_i^{\ell}, y_i^{\ell+1}\} \right\}.$$

To define the set E_3, we need to introduce some more notation. For a variable x_i, we let i_1 and i_2 be the two indices of the clauses containing the literal x_i, and we let i_3 be the index of the clause containing the literal $\bar{x}_i$. We then have

$$E_3 := \bigcup_{i=1}^{p} \left\{ \{y_i^2, k_{i_1}\}, \{y_i^4, k_{i_3}\}, \{y_i^6, k_{i_2}\} \right\}.$$

We assign cost 2 to the edges in E_1, unit cost to the edges in E_2, and a big cost $M >> 0$ to the edges of E_3 (in particular, $M > 2m + 8p$). Finally, we let the color classes[3] be defined as:

$$C_1 := \{k_1, \ldots, k_m\} \cup \left\{ \bigcup_{i=1}^{p} \{y_i^1, y_i^5\} \right\}; \quad C_2 := \{k_1, \ldots, k_m\} \cup \left\{ \bigcup_{i=1}^{p} \{y_i^3, y_i^7\} \right\}.$$

We claim that there exists an optimal solution to the SAND instance of cost at most $(M+2)m + 8p$ if and only if there is a truth assignment satisfying all clauses for the SAT instance.

3.1 Completeness

First, let us assume that the SAT instance is satisfiable. For each clause K_j, we select one literal that is set to *true* in the truth assignment. We define the paths for our terminal nodes in C_1 as follows. For each node $y \in \bigcup_{i=1}^{p} \{y_i^1, y_i^5\}$, we let the flow travel from y to r along the edge $\{y, r\}$. For each k_j, we let the flow travel to r on a path P_1^j, that we define based on the literal selected for K_j. Specifically, let x_i be the variable corresponding to the literal selected for the clause K_j. Then:

- if $K_j = K_{i_1}$, we let P_1^j be the path with nodes $\{k_j, y_i^2, y_i^3, r\}$,
- if $K_j = K_{i_2}$, we let P_1^j be the path with nodes $\{k_j, y_i^6, y_i^7, r\}$,
- if $K_j = K_{i_3}$, we let P_1^j be the path with nodes $\{k_j, y_i^4, y_i^3, r\}$.

[3] We here have $C_1 \cap C_2 \neq \emptyset$. However, the reduction can be easily modified to prove hardness of instances where $C_1 \cap C_2 = \emptyset$, by simply adding for all j two nodes k_j^1, k_j^2 adjacent to k_j with an edge of zero cost, and by letting k_j^1 (resp. k_j^2) be in C_1 (resp. C_2) instead of k_j.

We define the paths for our terminal nodes in C_2 similarly. For each node $y \in \bigcup_{i=1}^{p}\{y_i^3, y_i^7\}$, we let the flow travel from y to r along the edge $\{y, r\}$. For each k_j, we let the flow travel to r on a path P_2^j defined as follows. Let x_i be the variable corresponding to the literal selected for the clause K_j. Then:

- if $K_j = K_{i_1}$, we let P_2^j be the path with nodes $\{k_j, y_i^2, y_i^1, r\}$,
- if $K_j = K_{i_2}$, we let P_2^j be the path with nodes $\{k_j, y_i^6, y_i^5, r\}$,
- if $K_j = K_{i_3}$, we let P_2^j be the path with nodes $\{k_j, y_i^4, y_i^5, r\}$.

Note that the paths of terminals belonging to the same color set do not share edges. In fact, by construction, the paths of two terminals in C_1 could possibly share an edge only if for two distinct clauses $K_j \neq K_{j'}$ we selected a literal corresponding to the same variable x_i, and we have $K_j = K_{i_1}$ and $K_{j'} = K_{i_3}$, since in this case the paths P_1^j and $P_1^{j'}$ would share the edge $\{y_i^3, r\}$. However, selecting x_i for K_{i_1} means x_i takes value *true* in the truth assignment, while selecting x_i for K_{i_3} means x_i takes value *false* in the truth assignment, which is clearly a contradiction. A similar observation applies to paths of terminals in C_2. It follows that installing one unit of capacity on every edge that appears in (at least) one selected path is enough to support the flow of both color sets. The total installation cost is exactly $8p + (M + 2)m$.

3.2 Soundness

Suppose there is an optimal solution to the SAND instance of cost at most $(M + 2)m + 8p$. Let $\mathcal{S}$ denote such solution. Since the support of any feasible solution has to include at least one distinct edge of cost M for each node k_j, and $M > 2m + 8p$, it follows that $\mathcal{S}$ has exactly m edges of cost M in its support, each with one unit of capacity installed. Hence, if we denote by P_1^j (resp. P_2^j) the path used by k_j to send flow to r with terminals in C_1 (resp. C_2), we have the following fact.

Fact 1. For each $j = 1, \ldots, m$, the paths P_1^j and P_2^j from k_j to r share the first edge.

We use this insight to construct a truth assignment for the SAT variables. Specifically, let y_i^ℓ be the endpoint of the first edge of P_1^j and P_2^j. We set x_i to *true* if $y_i^\ell = y_i^2$ or if $y_i^\ell = y_i^6$, and we set x_i to *false* if $y_i^\ell = y_i^4$. We repeat this for all clauses $j = 1, \ldots, m$, and we assign an arbitrary truth value to all remaining variables, if any. In order to finish the proof, we have to show that this assignment is consistent for all $i = 1, \ldots, p$. To this aim, let us say that a variable x_i is *in conflict* if there is a node k_j sending flow to r on a path whose first edge has endpoint y_i^4, and there is node $k_{j'} \neq k_j$ sending flow to r on a path whose first edge has endpoint y_i^2 or y_i^6. Note that our assignment procedure is consistent and yields indeed a valid truth assignment if and only if there is no variable in conflict.

We now make a few claims on the structure of $\mathcal{S}$, that will be useful to show that no variable can be in conflict. Next fact follows from basic flow theory.

Fact 2. Without loss of generality, we can assume that the flow sent from terminals in C_1 (resp. C_2) to r, does not induce directed cycles.

▶ **Claim 11.** *Without loss of generality, we can assume that every terminal sends flow to r on a path that contains exactly one node $y \in \bigcup_{i=1}^{p}\{y_i^1, y_i^3, y_i^5, y_i^7\}$.*

We defer the proof of this claim which is central to the remaining proof to the end. Let G_i be the subgraph of G induced by the nodes $\{r, y_i^1, \ldots, y_i^7\}$, and let χ_i be the total cost of the capacity that $\mathcal{S}$ installs on the subgraph G_i. Note that, by Fact 1, the cost of $\mathcal{S}$ is

$m \cdot M + \sum_{i=1}^{m} \chi_i$. We will use Claim 1 to give a bound on the value χ_i. To this aim, let n_i be the number of nodes k_j whose path P_1^j contains edges of G_i. Note that $0 \leq n_i \leq 3$, and each k_j contributes to exactly one n_i, for some $i = 1, \ldots, p$.

▶ **Claim 12.** *We have $\chi_i \geq 8 + 2n_i$, with the inequality being strict if the variable x_i is in conflict.*

Claim 12 finishes our proof, since it implies that the cost of $\mathcal{S}$ is at least

$$m \cdot M + \sum_{i=1}^{p} \chi_i \geq m \cdot M + \sum_{i=1}^{p} (8 + 2n_i) = m \cdot M + 8p + 2m,$$

with the inequality being tight if and only if there is no variable in conflict. ◀

4 Latency SAND

By adapting a construction from [13], there is a $\Omega(\log n)$ gap between the tree and graph version of f-SAND. This naturally raises the question of approximating f-SAND when the solution must be restricted to different topologies. In this section, we consider the f-SAND when the output topology must be a path. Since this variant of f-SANDis not easy to solve on a tree, it is not clear how to solve it using tree metrics.

▶ **Definition 13.** In the *latency*-f-SAND problem, we are given an instance of f-SAND, but require the output to be a path with the root r as one of its endpoints. Our goal is output a minimum cost path, where the cost of an edge is $w_e \cdot$ (load on e). The load on an edge is the maximum number of nodes of one color it separates from the root.

We assume that the lengths are integers and polynomially bounded in the input and give a time-indexed length formulation for this problem. This linear programming formulation was introduced by Chakrabarty and Swamy [4] for orienteering problems.

The Linear Programming Formulation for Latency-f-SAND.

$$\min \quad \sum_{j,t} t \cdot x_{j,t} \qquad (\mathrm{LP}_{\mathcal{P}}^{b})$$

$$\text{s.t.} \quad \sum_{t} x_{j,t} \geq 1 \qquad \forall j \in [m] \qquad (2)$$

$$\sum_{P \in \mathcal{P}_{b \cdot t}} z_{P,t} \leq 1 \qquad \forall t \in [T] \qquad (3)$$

$$\sum_{P \in \mathcal{P}_{b \cdot t} : j \in P} z_{P,t} \geq \sum_{t' \leq t} x_{j,t'} \qquad \forall j \in [m], t \in [T] \qquad (4)$$

$$x, z \geq 0$$

We assume without loss of generality, that $|C_i| = m$ for all $i \in [k]$. $\mathcal{P}_t$ denotes the set of paths of weight at most t starting from the root. Since the lengths are polynomially bounded, we can contain a variable for each possible length (we denote T to be the maximum possible length). We use $j \in P_t$ to indicate that the path P_t contains j terminals of each color. The variable $x_{j,t}$ indicates that we have seen j terminals of each color by time t and $z_{P,t}$ indicates that we use path P to visit the terminals at time t.

▶ **Lemma 14.** *The linear program $LP_{\mathcal{P}}^{b}$ is a relaxation of Latency-*f-SAND*for $b \geq 1$.*

Proof. We show that the contraints and objective are valid for any feasible solution to Latency-f-SAND.

- Constraint 2 ensures that j terminals of each color are covered at some given time period, for every $j \in [m]$.
- Constraint 3 ensures that only one path is (fractionally) picked for each time period t.
- Constraint 4 indicates that we must have picked a path P that covers j terminals by time t if $\sum_{t' \leq t} x_{j,t'} = 1$.
- The objective function correctly captures the cost of the path. For an integer solution, $x_{j,t} = 1$ indicates that time t is the first time j terminals of each color are present in the path. Thus the objective counts the prefix length t^1 corresponding to where $x_{1,t^1} = 1$ in all m of the terms, the next prefix of length $t^2 - t^1$ in $m - 1$ of them and so on. This accurately accounts for the loads in these segments of the path according to the objective function in f-SAND. Finally, $b \geq 1$ only allows the paths to be of lengths longer by a factor of b so keeps the optimal solution feasible. ◀

First, we can relax the above LP by replacing $\mathcal{P}_t$ with $\mathcal{T}_t$ which is the set of all trees of size at most t. This is a relaxation as $\mathcal{P}_t \subseteq \mathcal{T}_t$. Lemma 15, shows that we can round $\text{LP}^b_{\mathcal{T}}$ to get a $O(b)$ approximation to latency-f-SAND.

▶ **Lemma 15.** *Given a fractional solution (x, z) to $LP^b_{\mathcal{T}}$, we can round it to a solution to latency-*f-SAND *with cost at most $O(b)$ times the cost of $LP^b_{\mathcal{T}}$.*

We defer the proof to the full version [18] due to space constraints but briefly sketch the argument. Roughly, we sample the trees at geometric intervals and "eulerify" them to produce a solution whose cost is not too much larger than the LP-objective.

Despite, being able to round the LP, we cannot hope to solve it effeciently due to the exponential number of variables in the primal. We will use the dual to obtain a solution to a relaxed version of the primal.

$$\max \quad \sum_j \alpha_j - \sum_t \beta_t \tag{Dual$^b_{\mathcal{P}}$}$$

$$\text{s.t.} \quad \alpha_j \leq t + \sum_{t' \geq t} \theta_{j,t'} \qquad \forall j, t \tag{5}$$

$$\sum_{j \in P} \theta_{j,t} \leq \beta_t \qquad \forall t, P \in \mathcal{P}_{bt} \tag{6}$$

$$\alpha, \beta, \theta \geq 0. \tag{7}$$

Following [4] it is suffient that an "approximate separation oracle" in the sense of Lemma 16 is sufficient to compute an optimal solution to $\text{LP}^b_{\mathcal{T}}$.

▶ **Lemma 16.** *Given a solution (α, β, θ), we can show that either (α, β, θ) is a solution to $Dual^1_{\mathcal{T}}$ or find a violated inequality for (α, β, θ) for $Dual^b_{\mathcal{T}}$ for $b = O(\log^2 k \log n)$.*

Once again, we defer the proof to the full version [18], but sketch the argument. To efficiently separate, we observe that constraint 6 can be recast as a covering Steiner tree problem. Using approximation algorithms for this problem, we find a violated inequality for a (stronger) constraint. This results in the "approximate separation oracle".

▶ **Theorem 17.** *There exists a $O(\log^2 k \log n)$ approximation to the Latency-SAND problem.*

Proof. Combining Lemma 3.2 of [4] with Lemma 16, we can now compute an ϵ-additive optimal solution to $\text{LP}^b_{\mathcal{T}}$ for $b = O(\log^2 k \log n)$. Using Lemma 15, we then achieve an $O(b)$ approximation for our problem. ◀

References

1 M. Balcan, F. Constantin, S. Iwata, and L. Wang. Learning valuation functions. In *Conference on Learning Theory*, volume 23, pages 4–1, 2012.

2 W. Ben-Ameur and H. Kerivin. Routing of uncertain demands. *Optimization and Engineering*, 3:283–313, 2005.

3 K. Bhawalkar and T. Roughgarden. Welfare guarantees for combinatorial auctions with item bidding. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 700–709. Society for Industrial and Applied Mathematics, 2011.

4 D. Chakrabarty and C. Swamy. Facility location with client latencies: LP-based techniques for minimum-latency problems. *Mathematics of Operations Research*, 41(3):865–883, 2016.

5 C. Chekuri. Routing and network design with robustness to changing or uncertain traffic demands. *SIGACT News*, 38(3):106–128, 2007.

6 M. Chlebik and J. Chlebikova. Approximation hardness of the steiner tree problem on graphs. *Proc. of the Scandinavian Workshop on Algorithm Theory*, pages 170–179, 2002.

7 N. G. Duffield, P. Goyal, A. G. Greenberg, P. P. Mishra, K. K. Ramakrishnan, and J. E. van der Merwe. A flexible model for resource management in virtual private networks. *Proceedings of SIGCOMM*, 29:95–108, 1999.

8 J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *Journal of Computer and System Sciences*, 69:485–497, 2004.

9 U. Feige. On maximizing welfare when utility functions are subadditive. *SIAM Journal on Computing*, 39(1):122–142, 2009.

10 A. E. Feldmann, J. Könemann, K. Pashkovich, and L. Sanità. Fast approximation algorithms for the generalized survivable network design problem. *Proceedings of ISAAC (International symposium on algorithms and computation)*, pages 33:1– 33:12, 2016.

11 J. Fingerhut, S. Suri, and J. Turner. Designing least-cost nonblocking broadband networks. *Journal of Algorithms*, 24(2):287–309, 1997.

12 M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.

13 N. Goyal, N. Olver, and F. B. Shepherd. Dynamic vs. oblivious routing in network design. *Algorithmica*, 61(1):161–173, 2011.

14 N. Goyal, N. Olver, and F. B. Shepherd. The VPN conjecture is true. *Journal of the ACM*, 60(3):17:1–17:17, June 2013.

15 F. Grandoni, T. Rothvoß, and L. Sanità. From uncertainty to non-linearity: Solving virtual private network via single-sink buy-at-bulk. *Mathematics of Operations Research*, 36(2):185–204, 2011.

16 A. Gupta, J. Kleingerg, R. Kumar, B. Rastogi, and B. Yener. Provisioning a virtual private network: A network design problem for multicommodity flow. *Proceedings of Symposium on Theory of Computing (STOC)*, pages 389–398, 2001.

17 A. Gupta, V. Nagarajan, and R. Ravi. An improved approximation algorithm for requirement cut. *Operations Research Letters*, 38(4):322–325, 2010.

18 G. Guruganesh, J. Iglesias, R. Ravi, and L. Sanità. Plane gossip: Approximating rumor spread in planar graphs. *arXiv preprint arXiv:1707.01487*, 2017.

19 K. Jain. A factor 2 approximation algorithm for the generalized steiner network problem. *Combinatorica*, 21(1):39–60, 2001.

20 B. Lehmann, D. Lehmann, and N. Nisan. Combinatorial auctions with decreasing marginal utilities. In *Proc. of the 3rd ACM Conf. on Electronic Commerce*, pages 18–28. ACM, 2001.

21 G. Oriolo, L. Sanità, and R. Zenklusen. Network design with a discrete set of traffic matrices. *Operations Research Letters*, 41(4):390–396, 2013.

22 M. Yannakakis. Node-and edge-deletion np-complete problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 253–264. ACM, 1978.

Path-Contractions, Edge Deletions and Connectivity Preservation*†

Gregory Gutin[1], M. S. Ramanujan[2], Felix Reidl[3], and Magnus Wahlström[4]

1 **Royal Holloway University, University of London, Egham, UK**
G.Gutin@rhul.ac.uk
2 **Algorithms and Complexity Group, TU Wien, Vienna, Austria**
ramanujan@ac.tuwien.ac.at
3 **Royal Holloway University, University of London, Egham, UK**
felix.reidl@gmail.com
4 **Royal Holloway University, University of London, Egham, UK**
Magnus.Wahlstrom@rhul.ac.uk

Abstract

We study several problems related to graph modification problems under connectivity constraints from the perspective of parameterized complexity: (Weighted) Biconnectivity Deletion, where we are tasked with deleting k edges while preserving biconnectivity in an undirected graph, Vertex-deletion Preserving Strong Connectivity, where we want to maintain strong connectivity of a digraph while deleting exactly k vertices, and Path-contraction Preserving Strong Connectivity, in which the operation of path contraction on arcs is used instead. The parameterized tractability of this last problem was posed in [Bang-Jensen and Yeo, Discrete Applied Math 2008] as an open question and we answer it here in the negative: both variants of preserving strong connectivity are W[1]-hard. Preserving biconnectivity, on the other hand, turns out to be fixed parameter tractable (FPT) and we provide an FPT algorithm that solves Weighted Biconnectivity Deletion. Further, we show that the unweighted case even admits a randomized polynomial kernel. All our results provide further interesting data points for the systematic study of connectivity-preservation constraints in the parameterized setting.

1998 ACM Subject Classification G.2.2 Graph algorithms

Keywords and phrases connectivity, strong connectivity, vertex deletion, arc contraction

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.47

1 Introduction

Some of the most well studied classes of network design problems involve starting with a given network and making modifications to it so that the resulting network satisfies certain connectivity requirements, for instance a prescribed edge- or vertex-connectivity. This class of problems has a long and rich history (see *e.g.* [1, 8]) and has recently started to be examined through the lens of parameterized complexity. Under this paradigm, we ask whether a (hard) problem admits an algorithm with a running time $f(k)n^{O(1)}$, where n is the size of the input,

* The research of Gregory Gutin was partially supported by Royal Society Wolfson Research Merit Award. M. S. Ramanujan acknowledges support from Austrian Science Fund (FWF, project P26696). Felix Reidl and Magnus Wahlström were supported by EPSRC grant EP/P007228/1.
† A full version of the paper is available at https://arxiv.org/abs/1704.06622.

k the *parameter*, and f some computable function. A natural parameter to consider in this context is the number of editing operations allowed and we can reasonably assume that this number is small compared to the size of the graph.

To approach this line of research systematically, let us identify the 'moving parts' of the broader question of editing under connectivity constraints: first and foremost, the network in question might best be modelled as either a directed or undirected graph, potentially with edge- or vertex-weights. This, in turn, informs the type of connectivity we restrict, *e.g.* strong connectivity or fixed value of edge-/vertex-connectivity. Additionally, the connectivity requirement might be non-uniform, *i.e.* it might be specified for individual vertex-pairs. The constraint one operates under might either be to *preserve*, to *augment*, or to *decrease* said connectivity. Finally, we need to fix a suitable editing operation; besides the obvious vertex- and edge-removal, more intricate operations like edge contractions are possible.

While not all possible combinations of these factors might result in a problem that currently has an immediate real-world application, they are nonetheless important data points in the systematic study of algorithmic tractability. For example, if we fix the editing operation to be the addition of edges (often called 'links' in this context) and our goal is to increase connectivity, then the resulting class of *connectivity augmentation problems* has been thoroughly researched. We refer to the monograph by Frank [8] for further results on polynomial-time solvable cases and approximation algorithms. Under the parameterized complexity paradigm, Nagamochi [16] and Guo and Uhlmann [10] studied the problem of augmenting a 1-edge- connected graph with k links to a 2-edge-connected graph. Nagamochi obtained an FPT algorithm for this problem while Guo and Uhlmann showed that this problem, alongside its vertex-connectivity variant, admits a quadratic kernel. Marx and Végh [14] studied the more general problem of augmenting the edge-connectivity of an undirected graph from $\lambda - 1$ to λ, via a minimum set of links that has a total *cost* of at most k, and obtained an FPT algorithm as well as a polynomial kernel for this problem. Basavaraju *et al.* [3] improved the running time of their algorithm and further showed the fixed-parameter tractability of a dual parameterization of this problem.

A second large body of work can be found in the antithetical class of problems, where we ask to *delete* edges from a network while *preserving* connectivity. Probably the most studied member of these *connectivity preservation problems* is the Minimum Strong Spanning Spanning Subgraph (MSSS) problem: given a strongly connected digraph we are asked to find a strongly connected subgraph with a minimum number of arcs. The problem is NP-complete (an easy reduction from the Hamiltonian Cycle problem) and there exist a number of approximation algorithms for it (see the monograph by Bang-Jensen and Gutin for details and references [1]). Bang-Jensen and Yeo [2] were the first to study MSSS from the parameterized complexity perspective. They presented an algorithm that runs in time $2^{O(k \log k)} n^{O(1)}$ and decides whether a given strongly connected digraph D on n vertices and m arcs has a strongly connected subgraph with at most $m - k$ arcs provided $m \geqslant 2n - 2$. Basavaraju *et al.* [4] extended this result not only to arbitrary number m of arcs but also to λ-arc-strong connectivity for an arbitrary integer λ, and they further extended it to λ-edge-connected *undirected* graphs.

We consider the undirected variant of this problem, however, we aim to preserve the *vertex-connectivity* instead of edge-connectivity. As noted by Marx and Végh [14], vertex-connectivity variants of parameterized connectivity problems seem to be much harder to approach than their edge-connectivity counterparts.[1] Moreover, even the complexity of the

[1] Marx and Végh [14] compare [18] and [7] to [9] and [17] with respect to polynomial-time exact and

problem of augmenting the vertex-connectivity of an undirected graph from 2 to 3, via a minimum set of up to k new links remains open [14]. Our main result in this direction is the first FPT algorithm for the following problem[2]:

WEIGHTED BICONNECTIVITY DELETION parametrised by k

Input: A biconnected graph G, $k \in \mathbb{N}$, $w^* \in \mathbb{R}_{\geqslant 0}$ and a function $w : E(G) \to \mathbb{R}_{\geqslant 0}$.

Problem: Is there a set $S \subseteq E(G)$ of size at most k such that $G - S$ is biconnected and $w(S) \geqslant w^*$?

▶ **Theorem 1.** WEIGHTED BICONNECTIVITY DELETION *can be solved in time* $2^{O(k \log k)} n^{O(1)}$.

We further show that this problem has a randomized polynomial kernelization when the edges are required to have only unit weights. To be precise, all inputs for the unweighted variant UNWEIGHTED BICONNECTIVITY DELETION (UBD) are of the form (G, k, w^*, w), where $w^* = k$ and $w(e) = 1$ for every $e \in E(G)$.

▶ **Theorem 2.** UBD *has a randomized kernel with* $O(k^9)$ *vertices.*

Along with arc-additions and arc-deletions, a third interesting operation on digraphs is the *path-contraction* operation which has been used to obtain structural results on paths in digraphs [1]. To *path-contract* an arc (x, y) in a digraph D, we remove it from D, identify x and y and keep the in-arcs of x and the out-arcs of y for the combined vertex. The resulting digraph is denoted by $D /\!\!/ (x, y)$. It is useful to extend this notation to sequences of contractions: let $S = (a_1, a_2, \ldots, a_p)$ be a sequence of arcs of a digraph D. Then $D /\!\!/ S$ is defined as $(\ldots((D /\!\!/ a_1) /\!\!/ a_2) /\!\!/ \ldots) /\!\!/ a_p$. Since the resulting digraph does not depend on the order of the arcs [1], this notation can equivalently be used for arc-sets.

Bang-Jensen and Yeo [2] asked whether the problem of path-contracting at least k arcs to maintain strong connectivity of a given digraph D is fixed-parameter tractable. Formally, the problem is stated as follows:

PATH-CONTRACTION PRESERVING STRONG CONNECTIVITY parametrised by k

Input: A strongly connected digraph D and an integer k.

Problem: Is there a sequence $S = (a_1, \ldots, a_k)$ of arcs of D such that $D /\!\!/ S$ is also strongly connected?

Our first result is a negative answer to the question of Bang-Jensen and Yeo. That is, we show that this problem is unlikely to be FPT.

▶ **Theorem 3.** PATH-CONTRACTION PRESERVING STRONG CONNECTIVITY *is W[1]-hard.*

We follow up this result by considering a natural vertex-deletion variant of the problem and extending our W[1]-hardness result to this problem as well. In this variant, the objective is to check for the existence of a set of *exactly*[3] k vertices such that on deleting these vertices from the given digraph, the digraph stays strongly connected.

▶ **Theorem 4.** VERTEX-DELETION PRESERVING STRONG CONNECTIVITY *is W[1]-hard.*

approximation algorithms.

[2] Note that since 1-vertex-connectivity is trivially equivalent to 1-edge-connectivity, the 1-vertex-connectivity case was proved to be FPT by Basavaraju *et al.* [4].

[3] We require *exactly* k vertices rather than *at least* k vertices to be deleted since the one-vertex digraph is strongly connected [1].

Our Methodology. Our algorithm for Weighted Biconnectivity Deletion builds upon the recent approach introduced by Basavaraju *et al.* [4] to handle connectivity preservation problems, in particular the p-λ-Edge Connected Subgraph (p-λ-ECS) problem where the objective is to delete k edges while keeping the graph λ-edge connected. Call an edge *deletable* (we refer to it as *non-critical* in the case of vertex-connectivity) if deleting it keeps the given (di)graph λ-edge connected, *undeletable* (*critical*) otherwise, and call an edge *irrelevant* if there is a solution disjoint from the edge.

For an even value of λ and a λ-edge-connected undirected graph G, Basavaraju *et al.* [4] proved that unless the total number of deletable edges is bounded by $O(\lambda k^2)$, it is possible in polynomial time to obtain a set F of k edges such that $G - F$ is still λ-edge-connected. This result does not hold for odd values of λ as can be seen, *e.g.*, when $\lambda = 1$ and G is a cycle. In this much more involved case, unless the total number of deletable edges is bounded by $O(\lambda k^3)$, it is possible in polynomial time to obtain either a set F of k edges such that $G - F$ is still λ-edge-connected or to identify an irrelevant edge.

Weighted Biconnectivity Deletion is similar to the case of odd λ as we find either a solution or an irrelevant edge. The main difference between our FPT algorithm and the one presented by Basavaraju *et al.* is the deep structural analysis necessitated by the shift from edge-connectivity to vertex-connectivity: While in the former case the failure to find a solution means that G can be decomposed into a 'cycle-like' structure as shown in [4], in our case no such simple structure arises. Instead, we perform a careful examination of mixed cuts in the graph, each of which comprises precisely one critical edge e and a vertex w which we call the *partner* of e. We show that either a large number of critical edges share a common partner or there is a large number of critical edges with pairwise distinct partners. In the former case, we prove the existence of an irrelevant edge while in the latter case we are able to construct a solution. Our result is based on a non-trivial combination of several new structural properties of biconnected graphs and critical edges which we believe is of independent interest and useful in the study of other connectivity-constrained problems.

The kernel stated in Theorem 2 relies on the powerful *cut-covering lemma* of Kratsch and Wahlström [13] which has been central to the development of several recent kernelization algorithms [12]. While Basavaraju *et al.* obtained a randomized compression for the p-λ-ECS problem using sketching techniques from dynamic graph algorithms, we provide an alternative approach and show that when dealing with biconnectivity it is also possible to obtain a (randomized) polynomial *kernel*. We believe that this approach could be applicable for higher values of vertex- connectivity and for other connectivity deletion problems, as long as one is able to bound the number of critical or undeletable edges in the given instance by an appropriate function of the parameter.

Further related work. In the Minimum Equivalent Digraph problem, given a digraph D, the aim is to find a spanning subgraph H of D with minimum number of arcs such that if there is an x-y directed path in D then there is such a path in H for every pair x, y of vertices of D. Since it is not hard to solve Minimum Equivalent Digraph for acyclic digraphs, Minimum Equivalent Digraph for general digraphs can be reduced to MSSS in polynomial time. Chapter 12 of the monograph of Bang- Jensen and Gutin [1] surveys pre-2009 results on Minimum Equivalent Digraph. The first exact algorithm for the Mnimum Equivalent Digraph problem, running in time $2^{O(m)}$, was given by Moyles and Thompson [15] in 1969, where m is the number of arcs in the graph. More recently, Fomin, Lokshtanov, and Saurabh [6] gave the first vertex-exponential algorithm for this problem, *i.e.* an algorithm with a running time of $2^{O(n)}$.

Paper organization. This paper is a shortened version of the full paper [11]. Due to the space limit, we omitted several results, proofs, and other material.

2 Preliminaries

Graphs. For an undirected graph G and vertex set $S \subseteq V(G)$, we denote by $E(S)$ the set of edges of G with both endpoints in S. For a vertex set $X \subseteq V(G)$, we denote by $N_G(X)$ the set of vertices of $V(G) \setminus X$ which are adjacent to a vertex in X. A vertex in a connected undirected graph is a *cut-vertex* if deleting this vertex disconnects the graph. A *biconnected graph* is a connected graph on two or more vertices having no cut-vertices.

▶ **Definition 5.** Let G be a graph and $x, y \in V(G)$ two vertices. An *x-y separator* (an *x-y cut*) is a set $S \subseteq V(G) \setminus \{x, y\}$ (respectively $S \subseteq E(G)$) such that there is no x-y path in $G - S$. A *mixed x-y cut* is a set $S \subseteq V(G) \cup E(G)$ such that $|S \cap E(G)| = 1$ and there is no x-y path in $G - S$.

▶ **Definition 6.** Let G be a graph and $x, y \in V(G)$. Let $\mathcal{P}$ be a set of internally vertex-disjoint x-y paths in G. Then, we call $\mathcal{P}$ an *x-y flow*. The *value* of this flow is $|\mathcal{P}|$. We say that an edge e *participates* in the x-y flow $\mathcal{P}$ if $e \in \bigcup_{P \in \mathcal{P}} P$.

We denote by $\kappa_G(x, y)$ the value of the maximum x-y flow in G with the reference to G dropped when clear from the context.

Let $\mathcal{P}$ be a set of paths in G which have an endpoint in Y and intersect only in x. Then, we refer to $\mathcal{P}$ as an x-Y flow, with the value of this flow defined as $|\mathcal{P}|$.

Directed graphs. We will refer to edges in a digraph as *arcs*. For a vertex x in a digraph D we write $N_D^-(x)$ and $N_D^+(x)$ to denote its in- and out-neighbours, respectively. A *sink* is a vertex with no out-neighbours and a *source* is a vertex with no in-neighbours.

Parameterized Complexity. An instance of a parameterized problem Π is a pair (I, k) where I is the *main part* and k is the *parameter*; the latter is usually a non-negative integer. A parameterized problem is *fixed-parameter tractable* if there exists a computable function f such that instances (I, k) can be solved in time $O(f(k)|I|^c)$ where $|I|$ denotes the size of I. The class of all fixed-parameter tractable decision problems is called FPT and algorithms which run in the time specified above are called FPT algorithms.

To establish that a problem under a specific parameterization is not in FPT (under common complexity-theoretic assumptions) we provide *parameter-preserving reductions* from problems known to lie in intractable classes like W[1] or W[2]. In such a reduction, an instance (I_1, k_1) is reduced in FPT time to an instance (I_2, k_2) where $k_2 \leqslant f(k_1)$ for some function f. In the context of this paper we will use that Independent Set under its natural parameterization (the size of the independent set) is W[1]-hard [5].

A *reduction rule* for a parameterized problem Π is an algorithm that given an instance (I, k) of a problem Π returns an instance (I', k') of the *same* problem. The reduction rule is said to be *sound* if it holds that $(I, k) \in \Pi$ if and only if $(I', k') \in \Pi$. A *kernelization* is a polynomial-time algorithm that given any instance (I, k) returns an instance (I', k') such that $(I, k) \in \Pi$ if and only if $(I', k') \in \Pi$ *and* $|I'| + k' \leqslant f(k)$ for some computable function f. The function f is called the *size* of the kernelization, and we have a polynomial kernelization if $f(k)$ is polynomially bounded in k. A *randomized kernelization* is an algorithm which is allowed to err with certain probability. That is, the returned instance will be equivalent to the input instance only with a certain probability.

3 Preserving strong connectivity

In this section, we prove Theorem 3.

▶ **Theorem 3.** Path-contraction Preserving Strong Connectivity *is W[1]-hard.*

Proof. We reduce Independent Set to Path-contraction Preserving Strong Connectivity.

Construction. Let (G, k) be an instance of Independent Set. We now define a digraph D as follows. We begin with the vertex set of D. For every vertex $v \in V(G)$, D has two vertices v^-, v^+. For every edge $e = (u, v) \in E(G)$, the digraph D has $k + 2$ vertices $\hat{e}, \hat{e}_1, \ldots, \hat{e}_{k+1}$. Finally, there are $2k + 4$ special vertices $x, y, x^1, \ldots, x^{k+1}, y^1, \ldots, y^{k+1}$. This completes the definition of $V(D)$. We now define the arc set of D (see Figure 1).

- For every $v \in V(G)$, we add the arc (v^-, v^+) in D .
- For every $i \in [k + 1]$, we add the arcs $\{(x, x^i), (x^i, x), (y, y^i), (y^i, y), (y, x)\}$.
- For every edge $e = (u, v) \in E(G)$ and $i \in [k + 1]$, we add the arcs $\{(\hat{e}, \hat{e}_i), (\hat{e}_i, \hat{e}),\ (v^-, \hat{e}),\ (\hat{e}, v^+),\ (u^-, \hat{e}), (\hat{e}, u^+)\}$ in D .
- For every $v \in V(G)$, we add the arc (x, v^-) and the arc (v^+, y).

This completes the construction of the digraph D. Clearly, D is strongly-connected.

For an edge $e = (u, v) \in E(G)$, we denote by $\mathcal{B}_e$ the set of arcs $\{(v^-, \hat{e}), (\hat{e}, v^+),\ (u^-, \hat{e}), (\hat{e}, u^+)\}$ and by $\mathcal{F}_e$, the set of arcs

$$\mathcal{B}_e \cup \{(\hat{e}, \hat{e}_i), (\hat{e}_i, \hat{e}) | i \in [k + 1]\} \cup \{(u^-, u^+), (v^-, v^+), (x, v^-), (v^+, y), (x, u^-), (u^+, y), (y, x)\}.$$

We refer to the subgraph of D induced by $\mathcal{F}_e$ as the *edge-selection gadget* in D corresponding to e (see Figure 1). The intuition here is that, as we will prove formally, any solution in D will contain at most one of the two arcs $(u^-, u^+), (v^-, v^+)$.

Proof of correctness. We now argue that (G, k) is a yes-instance of Independent Set if and only if (D, k) is a yes-instance of Path-contraction Preserving Strong Connectivity. In the forward direction, suppose that (G, k) is a yes-instance of Independent Set and let $X \subseteq V(G)$ be a solution. Observe that $S = \{(v^-, v^+) \mid v \in X\}$ is a pairwise vertex-disjoint set of arcs. We claim that S is a solution for the instance (D, k). That is, $|S| \geqslant k$ and $D /\!\!/ S$ is strongly connected. The former is true by definition. We prove the latter, as the claim below, in the full version of the paper.

▶ **Claim 7.** *$D' = D /\!\!/ S$ is strongly connected.*

We now consider the converse direction. Suppose that (D, k) is a yes-instance of Path-contraction Preserving Strong Connectivity and let $S = \{a_1, \ldots, a_k\}$ be a solution for this instance. We require the following claim whose proof can be found in the full version of the paper.

▶ **Claim 8.** *For every edge $e = (u, v) \in E(G)$, $|S \cap \{(u^-, u^+), (v^-, v^+)\}| \leqslant 1$. Furthermore, $S \subseteq \{(v^-, v^+) \mid v \in V(G)\}$.*

The claim above implies that if X is a solution for the reduced instance of Path-contraction Preserving Strong Connectivity, then the set S of arcs corresponds independent set in G. In other words, (G, k) is a yes-instance of Independent Set. This proves the correctness of the reduction and completes the proof of the theorem. ◀

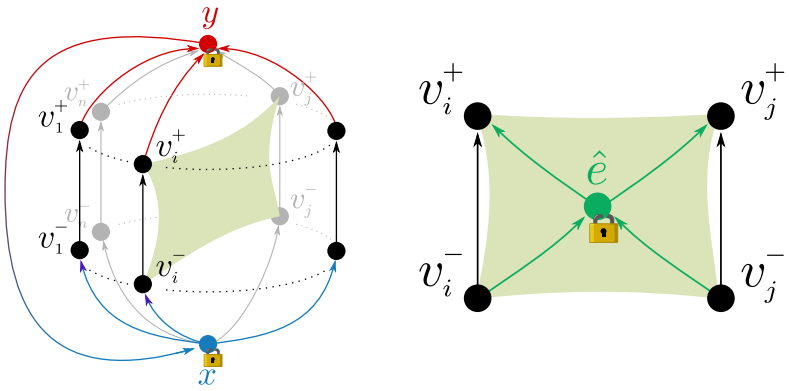


Figure 1 An illustration of the arcs in the reduced instance of Path-contraction Preserving Strong Connectivity. The second figure only contains the arcs of the edge-selection gadget corresponding to the edge $e = (v_i, v_j) \in E(G)$. Vertices with a padlock have additional $k+1$ pendant vertices with arcs in both directions.

4 Edge deletion to biconnected graphs

In this section, we consider the Weighted Biconnectivity Deletion problem on undirected graphs. Recall that the problem is defined as follows:

Weighted Biconnectivity Deletion parametrised by k

Input: A biconnected graph G, $k \in \mathbb{N}$, $w^* \in \mathbb{R}_{\geqslant 0}$ and a function $w : E(G) \to \mathbb{R}_{\geqslant 0}$.

Problem: Is there a set $S \subseteq E(G)$ of size at most k such that $G - S$ is biconnected and $w(S) \geqslant w^*$?

We refer to a set $S \subseteq E(G)$ such that $G - S$ is biconnected as a *biconnectivity deletion set* of G. For an instance (G, k, w^*, w) of Weighted Biconnectivity Deletion and a biconnectivity deletion set S of G, we say that S is a *solution* if $|S| \leqslant k$ and $w(S) \geqslant w^*$. The main result of this section is the following.

▶ **Theorem 1.** Weighted Biconnectivity Deletion *can be solved in time* $2^{O(k \log k)} n^{O(1)}$.

We denote by $\kappa(G)$ the vertex-connectivity of a graph G. Let G be a biconnected graph. An edge $e \in E(G)$ is called *critical* if $\kappa(G - e) < 2$. We denote by $\mathsf{Critical}_G(e)$ the subset of $E(G)$ comprising edges which are critical in $G - e$ but not in G. We denote by $\mathsf{Critical}_G(\emptyset)$ the set of edges which are already critical in G. In all notations, we ignore the explicit reference to G when it is clear from the context. We say that e is *critical for* a pair of vertices u, v in G if u and v are non-adjacent and e participates in every u-v flow of value two in G.

4.1 The FPT algorithm for Weighted Biconnectivity Deletion

To prove Theorem 1 we consider a more general version of the Weighted Biconnectivity Deletion problem where the input also includes a set $E^\infty \subseteq E(G)$ and the objective is to decide whether there is a solution disjoint from this set. Henceforth, instances of Weighted Biconnectivity Deletion will be of the form (G, k, w^*, w, E^∞) and any solution S is required to be disjoint from E^∞. We will refer to edges of $E(G) \setminus E^\infty$ as *potential solution edges*. We say that a potential solution edge e is *irrelevant* if either the instance has no

solution, or has a solution that does not contain e. For an instance $I = (G, k, w^*, w, E^\infty)$ and $r \in \mathbb{N}$, we denote by $\mathsf{Heavy}_I(r)$ the *heaviest* r potential solution edges of G with respect to the function w. If I is clear from the context, we simply write $\mathsf{Heavy}(r)$ when referring to $\mathsf{Heavy}_I(r)$.

Observe that no edge from the set $\mathsf{Critical}_G(\emptyset)$ can be part of a solution. As a result, we assume without loss of generality that for any instance (G, k, w^*, w, E^∞), the set $\mathsf{Critical}_G(\emptyset)$ is contained in E^∞. Furthermore, since the edges in E^∞ can never be part of a solution, we assume without loss of generality that for every edge $e \in E^\infty$, $w(e) = 0$. The proof of Theorem 1 is based on the following lemma which states that either a) the number of potential solution edges in the instance is already bounded polynomially in k, or b) a 'small' set of the heaviest edges in the instance must intersect a solution, or c) there is an irrelevant edge which can be found in polynomial time. For ease of presentation, let use define the polynomial $\mu(x) := 20x^3 + 46x^2 + x$ for the rest of this section.

▶ **Lemma 9.** *Let $I = (G, k, w^*, w, E^\infty)$ be an instance of* Weighted Biconnectivity Deletion. *If $|E(G) \setminus E^\infty| > \mu(k)$, then the set $\mathsf{Heavy}(\mu(k))$ contains either a solution edge or an irrelevant edge which can be computed in polynomial time.*

Proof. We give a brief proof sketch for the lemma. Note that the individual claims below do not map directly to claims in the full paper, but are present to illustrate the important ideas.

The general strategy of our result, following Basavaraju *et al.* [4], is a greedy algorithm that iteratively selects a non-critical edge $e \in \mathsf{Heavy}(\mu(k))$ for inclusion into a solution, computes which other edges become critical by the deletion of e, and either proceeds to select another edge for inclusion or halt, if either all edges of the remaining graph are critical or if the solution already contains k edges. Let us first cover the latter case.

▶ **Claim 10.** *If there exists a biconnectivity deletion set $S \subseteq \mathsf{Heavy}(\mu(k))$ with $|S| = k$, then any solution to the instance must intersect $\mathsf{Heavy}(\mu(k))$.*

Recall that this is one of the positive outcomes of Lemma 9. Therefore, we henceforth assume that there are more than $\mu(k)$ potential solution edges, but that the greedy algorithm terminates after fewer than k steps due to all edges of the remaining graph being critical. Let $f_1, \ldots, f_t$, $t < k$ be the sequence of edges selected by the greedy algorithm. By a pigeonhole argument, there must then be some index $r \in [t]$ such that $|\mathsf{Critical}_{G-\{f_1,\ldots,f_{r-1}\}}(f_r)| > 20k^2 + 46k$. Let $S = \{f_1, \ldots, f_{r-1}\}$ be the edges deleted until this point, let $e = f_r = (x, y)$, and let $G' = G - S$. We proceed to analyse the structure implied by the edges in $\mathsf{Critical}_{G'}(e)$.

▶ **Claim 11.** *The following hold.*

1. *The maximum value of an x-y flow in G' is 2.*
2. *For any x-y flow $\mathcal{P} = \{P_1, P_2\}$ in G', every edge of $\mathsf{Critical}_{G'}(e)$ participates in $\mathcal{P}$.*
3. *For every edge $e' \in \mathsf{Critical}_{G'}(e)$, say $e' \in E(P_1)$, there is a mixed x-y cut $\{e', w\}$ in $G' - e$, and for every such mixed cut we have $w \in V(P_2)$.*

In the latter case, we refer to w as a *partner vertex* of e', and to the set of all partner vertices of e' as the *partner set* of e'. The crux of the remainder of the proof lies in analysing the interaction between different mixed cuts and partner sets in G'. For convenience, we fix an order on P_1 and P_2 where we traverse both paths from x to y. We denote $\hat{E} = \mathsf{Critical}_{G'}(e) \cap E(P_1)$, and assume without loss of generality that $|\hat{E}| > 10k^2 + 23k$. The following is the most important structural observation on which our proof is based. For each edge $e_i \in \hat{E}$, let V_i denote the partner set of e_i.

▶ **Claim 12.** *For every pair of edges e_i, e_j in $\hat{E}$, where e_i lies before e_j on P_1, $|V_i \cap V_j| \leqslant 1$, and if $V_i \cap V_j$ contains such a common vertex w, then w is the last vertex of V_i and the first vertex of V_j on P_2.*

Our proof now splits into two fundamentally different cases. Either the flow $\mathcal{P}$ contains a long sequence of pairwise essentially non-interacting mixed cuts, or there are many edges of $\hat{E}$ with pairwise identical partner sets. We cover the first case now.

▶ **Claim 13.** *Let $e_1, \ldots, e_{3k+1}$ be a sequence of edges of $\hat{E}$, traversed in this order from x to y, such that for every $i \in [3k]$ the edges e_i and e_{i+1} have distinct partner sets. Then the set $F = \{e_1, e_4, \ldots, e_{3k-2}\}$ is a biconnectivity deletion set for G.*

By Claim 10, this would imply Lemma 9, so it remains to consider the case when Claim 13 fails to apply. In fact, Claim 13 applies whenever there is a sufficiently large number of distinct partner sets for edges of $\hat{E}$; hence we may assume that there is a large number of distinct edges of $\hat{E}$ with identical partner sets. Let $\hat{E}' \subseteq \hat{E}$ be a set of $\Omega(k)$ edges with pairwise identical partner sets. Then by Claim 12, there is a single vertex $w \in V(P_2)$ such that for any $e' \in \hat{E}'$ the partner set of e' is simply $\{w\}$. We show that this case implies an irrelevant edge rule. For simplicity, we illustrate the rule for the case that $S = \emptyset$.

▶ **Claim 14.** *Assume that $G = G'$ and $|\hat{E}'| \geqslant 2k + 4$. Let $e' = \arg\min_{e' \in \hat{E}'} w(e')$. Then for any biconnectivity deletion set S' of size k in G we have $|S' \cap \hat{E}'| \leqslant 1$, and if $e' \in S$ then there exists an edge $e'' \in \hat{E}' \setminus \{e'\}$ such that the set $S'' = (S' \setminus \{e'\}) \cup \{e''\}$ is a biconnectivity deletion set with $|S''| = k$ and $w(S'') \geqslant w(S')$. Hence e' is an irrelevant edge in G.*

By additional arguments omitted from this sketch, a similar result also holds for the general case of $S \neq \emptyset$, and under the assumption that $|\hat{E}| > 10k^2 + 23k$ we can show that either Claim 13 or Claim 14 applies. Hence in every case we find either an irrelevant edge or a large biconnectivity deletion set, and Lemma 9 follows. ◀

Given Lemma 9, Theorem 1 is proved as follows. Let $I = (G, k, w, w^*, E^\infty)$ be an instance of Weighted Biconnectivity Deletion. If the number of potential solution edges in this instance is already bounded by $\mu(k)$, then we simply enumerate all k-sized subsets of this set (there are $2^{O(k \log k)}$ choices) and check in polynomial time whether one of these subsets is a solution. Otherwise, we invoke Lemma 9 and either correctly conclude that the set $\mathsf{Heavy}(\mu(k))$ contains a solution edge, or we compute an irrelevant edge e in polynomial time. In the first case we branch on the set $\mathsf{Heavy}(\mu(k))$, reduce the budget k by 1 and the target weight w^* accordingly and recursively solve the resulting instance. In the second case, we add the edge e to the set E^∞ (thus decreasing the set of potential solution edges) and repeat.

4.2 A randomized kernel for Unweighted Biconnectivity Deletion

We now present our randomized kernel for the Weighted Biconnectivity Deletion problem where instances are of the form (G, k, w^*, w, E^∞) where $w(e) = 1$ for every $e \in E(G) \setminus E^\infty$, $w(e) = 0$ for every $e \in E^\infty$, and $w^* = k$. This version of the problem will be referred to as Unweighted Biconnectivity Deletion and instances of this problem will henceforth be of the form (G, k, E^∞) where a solution is a biconnectivity deletion set of size k contained in $E(G) \setminus E^\infty$. We continue to refer to the set $E(G) \setminus E^\infty$ as the set of potential solution edges and assume without loss of generality that at any point, any edge in the set $\mathsf{Critical}_G(\emptyset)$ is already part of E^∞. Finally, recall that a *linkage* from A to B in a digraph

D, where A and B are vertex sets, is a collection of $|A| = |B|$ pairwise vertex-disjoint paths originating in A and terminating in B.

Our kernelization relies on a result of Kratsch and Wahlström [13]. Before we are able to state it formally, we need the following definitions. Let us define a *potentially overlapping A-B vertex cut* in a digraph D to be a set of vertices $C \subseteq V(D)$ such that $D - C$ contains no directed path from $A \setminus C$ to $B \setminus C$. For any digraph D and set $X \subseteq V(D)$, a set $Z \subseteq V(D)$ is called a *cut-covering set* for (D, X) if for any $A, B, R \subseteq X$, there is a minimum-cardinality potentially overlapping A-B vertex cut C in $D - R$ such that $C \subseteq Z$.

▶ **Lemma 15** (Corollary 3, [13]). *Let D be a directed graph and let $X \subseteq V(D)$. We can identify a cut-covering set Z for (D, X) of size $O(|X|^3)$ in polynomial time with failure probability $O(2^{-|V(D)|})$.*

We first give a randomized kernelization that outputs an instance whose size is bounded polynomially in the number of the potential solution edges in the input instance.

▶ **Lemma 16.** Unweighted Biconnectivity Deletion *has a randomized kernel with number of vertices bounded by $O(|E(G) \setminus E^\infty|^3)$.*

Proof. Let $F = E(G) \setminus E^\infty$ be the set of potential solution edges. Now, the kernelization task essentially consists of retaining enough information from the input graph G to verify for any set $S \subseteq F$, whether S is a biconnectivity deletion set for G. Observe that this is equivalent to verifying whether there exists an edge $e = (u, v) \in S$, such that the maximum value of a u-v flow in $G - S$ is less than 2. We show an equivalent formulation of this as a question about the existence of linkages in an auxiliary digraph, followed by an application of Lemma 15.

For the formulation, we create a digraph $D_{G,F}$ from G and F. We refer to this digraph as D when G and F are clear from the context. In the first step, subdivide every edge $e \in F$ with a new vertex x_e. That is, for an edge $e = (u, v) \in F$, we create a new vertex x_e, remove the edge e and add edges (u, x_e) and (v, x_e). Let G_1 be the resulting undirected graph. In the second step, replace every edge (u, v) in $E(G_1)$ by a pair of arcs (u, v), (v, u). Finally, for every vertex v incident to any edge of F in G, add vertices v^+, v^- and add arcs from v^+ to all vertices in $N_{G_1}(v)$ and from all vertices in $N_{G_1}(v)$ to v^-. Let D be the resulting digraph. Note that $N_D^+(v^-) = \emptyset$ and $N_D^-(v^+) = \emptyset$. Let $X_E = \{x_e \mid e \in F\}$, $X_V = \{v^+, v^-, v \mid e \in F, e = (u, v)\}$ and $X = X_E \cup X_V$. We now relate solutions for the given instance and linkages in D. The following assertion is proved in the full version of the paper.

▶ **Claim 17.** *For any $S \subseteq F$, S is a biconnectivity deletion set for G if and only if for every edge $(u, v) \in S$ there is a linkage from $\{u^+, u\}$ to $\{v^-, v\}$ in $D - \{x_e \mid e \in S\}$.*

Let $Z \subseteq V(D)$ be the cut-covering set for (D, X), as computed by the algorithm of Lemma 15. Having in hand the set Z, we define the set $Y = (Z \cap V(G)) \cup V(F)$. Note that Z could contain vertices from X_V, but we want Y to be a subset of $V(G)$. Therefore, we first add to Y those vertices in Z which are also vertices in G and then add the vertices of $V(F)$. Our objective now is to reduce G down to what is commonly known as the *torso* graph of G defined by Y (see [13]). We now make this precise in the form of reduction rules. In the rest of the proof of the lemma, we fix Z to be a set computed using Lemma 15 and let Y be as defined above. We now state three reduction rules which will be applied on the given instance in the order in which they are presented.

▶ **Reduction Rule 18.** *If $k = 0$, then return an arbitrary yes-instance of constant size.*

▶ **Reduction Rule 19.** *Suppose that Reduction Rule 18 has been applied on the given instance. If there is an edge $(u, v) \in F$ such that G contains a u-v path avoiding all edges of F and all vertices of $Y \setminus \{u, v\}$, then delete (u, v) from G and reduce the budget k by 1. That is, return the instance $(G - \{(u, v)\}, k - 1, E^\infty)$.*

▶ **Reduction Rule 20.** *Suppose that Reduction Rule 18 and Reduction Rule 19 have been applied exhaustively on the given instance. For every pair $u, v \in Y$ such that $(u, v) \notin E(G)$ and there is a u-v-path in G that is internally vertex-disjoint from Y, we add the edge (u, v). Finally, return the instance (G', k, E'^∞), where $G' = G[Y]$ and $E'^\infty = (E^\infty \cap E(G')) \cup (E(G') \setminus E(G))$.*

The soundness of Rule 18 is trivial and we move on to prove the soundness of the remaining two rules.

▶ **Claim 21.** *Reduction Rules 19 and 20 are sound.*

Proof. Let $e = (p, q) \in F$ be an edge which is deleted in an application of Reduction Rule 19. Observe that in order to argue the soundness of this reduction rule, it suffices to argue that e is part of some solution for the given instance (if there exist any). Let S be an arbitrary subset of F containing e such that $S \setminus \{e\}$ is a solution. If S itself is a biconnectivity deletion set then we may correctly conclude that e is part of some solution for the given instance. Suppose that this is not the case.

Recall that by the previous claim, S is a biconnectivity deletion set for G if and only if there is a linkage from $\{u^+, u\}$ to $\{v^-, v\}$ in $D - \{x_e \mid e \in S\}$ for every $(u, v) \in S$. Since we are in the case that S is *not* a biconnectivity deletion set, there is a $(u, v) \in S$, with $A = \{u^+, u\}$, $B = \{v^-, v\}$, and $R = \{x_e \mid e \in S\}$ such that there is no linkage from A to B in $D - R$. Since $S \setminus \{e\}$ is a biconnectivity deletion set, we may assume without loss of generality that $u = p$ and $v = q$ and furthermore, $\kappa_{G-S}(p, q) = 1$. In addition, the fact that Z is a cut-covering set for (D, X) implies that Z contains a vertex w such that $C = \{w\}$ is a minimum-cardinality potentially overlapping A-B vertex cut in $D - R$. It is straightforward to see that $w \notin \{p, q, p^+, q^-\}$ since otherwise, there will be at least one path from A to B which is disjoint from w. Finally, since $\kappa_{G-S}(p, q) = 1$, it follows that every p-q path in $G - S$ intersects w. If $w \in X_E$ then we know that it corresponds to an edge in F. Otherwise, it corresponds to a vertex in Y. In either case, we obtain a contradiction to the applicability of Reduction Rule 19 on the edge (p, q), completing the proof of soundness for this rule.

We now argue the soundness of Reduction Rule 20. To do so, we prove that $S \subseteq F$ is a solution for (G, k, E^∞) if and only if it is a solution for (G', k, E'^∞). Let $D_1 = D_{G,F}$ and let $D_2 = D_{G',F}$.

In the forward direction, suppose that S is a solution for (G, k, E^∞). By Claim, 17, it follows that for every edge $(u, v) \in S$, there is a linkage from $\{u^+, u\}$ to $\{v^-, v\}$ in $D_1 - \{x_e \mid e \in S\}$. Fix such an edge (u, v) and let the paths in the linkage be P_1, P_2. If we demonstrate such a linkage in D_2, then we are done. This can be achieved as follows. Let $i \in \{1, 2\}$ and consider a pair of vertices $x_i, y_i \in V(P_i) \cap Y$ such that the subpath of P_i from x_i to y_i has all its internal vertices disjoint from Y. Then, we know that the graph G' contains the edge (x_i, y_i) and hence the digraph D_2 contains the arc (x_i, y_i). We replace the subpath from x_i to y_i with the arc (x_i, y_i) and we do this for every such subpath of P_i. It is straightforward to see that what results is indeed a linkage from $\{u^+, u\}$ to $\{v^-, v\}$ in $D_2 - \{x_e \mid e \in S\}$. Hence, we conclude that S is a solution for (G', k, E'^∞).

The same argument can be reversed for the converse direction in order to convert, for any $(u, v) \in S$, a linkage from $\{u^+, u\}$ to $\{v^-, v\}$ in $D_2 - \{x_e \mid e \in S\}$ to a a linkage from

$\{u^+, u\}$ to $\{v^-, v\}$ in $D_1 - \{x_e \mid e \in S\}$. This completes the proof of soundness of Reduction Rule 20. ◀

The above claim implies that if $(G', k', E(G') \setminus F')$ is the instance obtained by exhaustively applying the three reduction rules above, then $(G', k', E(G') \setminus F')$ is indeed equivalent to (G, k, E^∞). Furthermore, the size $|V(G')| = O(|F|^3)$ and the randomized polynomial running time follow from Lemma 15. This completes the proof of the lemma. ◀

▶ **Theorem 2.** UBD *has a randomized kernel with* $O(k^9)$ *vertices.*

Proof sketch. Let (G, k, E^∞) be the given instance and let $F = E(G) \setminus E^\infty$ be the set of potential solution edges in this instance. We present reduction rules which reduce F (while maintaining equivalence) to size $O(k^3)$; the result then follows from Lemma 16.

If $|F| = O(k^3)$, we are done. Otherwise, following the approach described in Subsection 4.1 in the full version [11], we greedily construct a biconnectivity deletion set in G, at each step keeping track of the edges that become critical. That is, we let $\hat{S} = \{f_1, \ldots, f_r\} \subseteq F$ be a set greedily constructed as follows. The edge f_1 is an arbitrary edge in F and for each $2 \leqslant i \leqslant r$, f_i is an arbitrary edge which is *not* critical in $G - \{f_1, \ldots, f_{i-1}\}$. As earlier, we terminate this procedure after k steps if we manage to find edges $\{f_1, \ldots, f_k\}$ or earlier if for some $r < k$, every remaining edge of F is critical in $G - \{f_1, \ldots, f_r\}$.

If $r = k$, then we identify the instance as a yes-instance and return an arbitrary yes-instance of constant size. Otherwise, if there is an $i \in [r]$ such that $G - \{f_1, \ldots, f_i\}$ is biconnected and $|\mathsf{Critical}_{G-\{f_1,\ldots,f_{i-1}\}}(f_i)| \geqslant 20k^2 + 46k$, then we execute the case analysis, as in Subsection 4.1 in the full version [11], and in polynomial time, either find $3k+1$ distinct partner sets or an irrelevant edge. In the latter case, we simply remove this irrelevant edge from F (add it to the set E^∞). Finally, if we reach a case with at least $3k+1$ distinct partner sets, we can find a biconnectivity deletion set $S \subseteq F$ with $|S| \geqslant k$ in polynomial time (see the full version), and since we are dealing with the unweighted case, we can simply identify the instance as a yes-instance and return an arbitrary yes-instance of constant size.

The only remaining case is that this greedy algorithm fails to produce a large enough solution yet never marks too many edges as critical at once. That is, it terminates in $r < k$ steps and never marks more than $20k^2 + 46k$ edges as critical in step i for any $i \in [r]$. This implies that $|F| \leqslant 20k^3 + 46k^2 + k = O(k^3)$, completing the proof of the theorem. ◀

5 Conclusions

Our results on Path-contraction Preserving Strong Connectivity and Weighted Biconnectivity Deletion provide additional data points for the algorithmic landscape of graph editing problems under connectivity constraints and its application in network design.

Since we established that Path-contraction Preserving Strong Connectivity is W[1]-hard for general digraphs, we ask whether the problem becomes FPT when restricted to planar digraphs or other structurally sparse classes.

Concerning the parameterized algorithm for Weighted Biconnectivity Deletion, we ask whether the dependence of $2^{O(k \log k)}$ can be improved to single-exponential or proven to be optimal. Naturally, we would further like to know whether we can reach beyond *bi*connectivity and extend our algorithm to higher values of vertex-connectivity. Is it possible to obtain a similar algorithm on digraphs?

Finally, regarding our polynomial kernel for Unweighted Biconnectivity Deletion, we ask whether it is possible to obtain a deterministic kernel. It is also left open whether the weighted case admits a polynomial kernel.

The results presented in this paper raise more questions than they answer, a clear indication that connectivity constraints are far from properly explored under the paradigm of parameterized complexity. As such, the topic offers exciting but challenging opportunities for further research.

References

1 Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.

2 Jørgen Bang-Jensen and Anders Yeo. The minimum spanning strong subdigraph problem is fixed parameter tractable. *Discrete Applied Mathematics*, 156(15):2924–2929, 2008.

3 Manu Basavaraju, Fedor V Fomin, Petr Golovach, Pranabendu Misra, MS Ramanujan, and Saket Saurabh. Parameterized algorithms to preserve connectivity. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 800–811. Springer, 2014.

4 Manu Basavaraju, Pranabendu Misra, M. S. Ramanujan, and Saket Saurabh. On finding highly connected spanning subgraphs. *CoRR*, abs/1701.02853, 2017.

5 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

6 Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, and Saket Saurabh. Efficient computation of representative families with applications in parameterized and exact algorithms. *Journal of the ACM (JACM)*, 63(4):29:1–29:60, September 2016.

7 András Frank. Augmenting graphs to meet edge-connectivity requirements. *SIAM J. Discrete Math.*, 5(1):25–53, 1992.

8 András Frank. *Connections in Combinatorial Optimization*. Oxford Univ. Press, 2011.

9 András Frank and Tibor Jordán. Minimal edge-coverings of pairs of sets. *J. Comb. Theory, Ser. B*, 65(1):73–110, 1995.

10 Jiong Guo and Johannes Uhlmann. Kernelization and complexity results for connectivity augmentation problems. *Networks*, 56(2):131–142, 2010.

11 Gregory Gutin, M. S. Ramanujan, Felix Reidl, and Magnus Wahlström. Path-contractions, edge deletions and connectivity preservation. *CoRR*, abs/1704.06622, 2017. URL: `https://arxiv.org/abs/1704.06622`.

12 Stefan Kratsch. Recent developments in kernelization: A survey. *Bulletin of the EATCS*, 113, 2014.

13 Stefan Kratsch and Magnus Wahlström. Representative sets and irrelevant vertices: New tools for kernelization. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 450–459. IEEE Computer Society, 2012.

14 Dániel Marx and László A Végh. Fixed-parameter algorithms for minimum-cost edge-connectivity augmentation. *ACM Transactions on Algorithms (TALG)*, 11(4):27, 2015.

15 Dennis M Moyles and Gerald L Thompson. An algorithm for finding a minimum equivalent graph of a digraph. *Journal of the ACM (JACM)*, 16(3):455–460, 1969.

16 Hiroshi Nagamochi. An approximation for finding a smallest 2-edge-connected subgraph containing a specified spanning tree. *Discrete Applied Mathematics*, 126(1):83–113, 2003. 5th Annual International Computing and combinatorics Conference.

17 László A. Végh. Augmenting undirected node-connectivity by one. *SIAM J. Discrete Math.*, 25(2):695–718, 2011.

18 T. Watanabe and A. Nakamura. Edge-connectivity augmentation problems. *J. Comput. System Sci.*, 35:96 – 144, 1987.

Dynamic Clustering to Minimize the Sum of Radii*

Monika Henzinger[1], Dariusz Leniowski[2], and Claire Mathieu[3]

1 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
`monika.henzinger@univie.ac.at`
2 **University of Vienna, Faculty of Computer Science, Vienna, Austria**
`dariusz.leniowski@univie.ac.at`
3 **ENS, CNRS, PSL Research University, Paris, France**
`cmathieu@di.ens.fr`

Abstract

In this paper, we study the problem of opening centers to cluster a set of clients in a metric space so as to minimize the sum of the costs of the centers and of the cluster radii, in a dynamic environment where clients arrive and depart, and the solution must be updated efficiently while remaining competitive with respect to the current optimal solution. We call this *dynamic sum-of-radii clustering* problem.

We present a data structure that maintains a solution whose cost is within a constant factor of the cost of an optimal solution in metric spaces with bounded doubling dimension and whose worst-case update time is logarithmic in the parameters of the problem.

1998 ACM Subject Classification G.1.6 Optimization

Keywords and phrases dynamic algorithm, clustering, approximation, doubling dimension

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.48

1 Introduction

The main goal of clustering is to partition a set of objects into homogeneous and well separated subsets (clusters). Clustering techniques have long been used in a wide variety of application areas, see for example the excellent surveys [25, 21].

There are several ways to model clustering. Among them, the problem of sum-of-radii (or sum-of-diameter) clustering has been extensively studied: the clients are located in a metric space and one must open facilities to minimize facility opening cost (or keep the number of open facilities limited to at most k) plus the sum of the cluster radii (or, in other applications, cluster diameters). To give a concrete example, imagine a telecommunications agency setting up mobile towers that provide wireless access to selected clients, incurring costs for setting up towers as well as for configuring a tower to serve the customers lying within a certain distance, where that latter contribution to the cost increases with the maximum distance served by the tower.

Assume the number of facilities is limited to k. For sum-of-diameter clustering, Doddi, Marathe, Ravi, Taylor and Widmayer [15] prove hardness of approximation to within better than a factor of 2. More recently, NP-hardness was proved for the sum-of-radii problem even for shortest path metrics on weighted planar graphs [24], or, in the case of sum-of-diameters, even for metrics of constant doubling dimension [17]. Turning to approximation algorithms,

* The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

Charikar and Panigraphy [13] design and analyze an O(1) approximation algorithm for sum-of-radii and for sum-of-diameter clustering with k clusters. They start from a linear-programming relaxation, use a primal-dual type approach and, along the way, design a bicriteria algorithm. They also design an incremental algorithm that handles arrivals of clients, merging clusters as needed so that at any time the clustering has $O(k)$ clusters and the cost is O(1) times the optimal cost for k clusters.

There have been many other papers on sum-of-radii or sum-of-diameters clustering. A few papers focus on the problems of partitioning the clients into a constant number of clusters as quickly as possible [20, 24]. Some papers concern themselves with bicriteria results such as [2]. Consider the special case of a metric that is Euclidean in two dimensions. Lev-Tov and Peleg [23] give a polynomial time approximation scheme (PTAS) for the related problem of covering input clients by a min-cost set of disks centered at the servers, where both clients and potential servers are located in the Euclidean plane and are part of the input. Recently, Behsaz and Salavatipour [4] gave a PTAS for the minimum sum of diameters problem on the plane with Euclidean distances. See also [1, 18] for other work on the two-dimensional geometric setting.

The problems are complicated by situations where the set of clients may change over time, for example documents in a very large database that must be efficiently searchable and maintained. This then leads to various models: online [14, 16], incremental[13], streaming, or dynamic. The dynamic setting, where clients may not only arrive but also depart, has been empirically studied at least since 1993 [12], and is the focus of the present paper, with the joint goals of maintaining clusterings whose objective value is close to optimal, and of updating the cluster quickly after each event.

This paper can be interpreted as part of a recent focus on exploiting primal-dual techniques in the dynamic setting. In the online setting (where new elements arrive but never depart), primal-dual techniques are extremely successful [11]. Initially it seemed that such techniques were inherently restricted to settings with arrivals only, and no departures, but there recently has been exciting progress to handle the dynamic setting as well, starting with [8, 7] and continuing with [3, 9, 26, 10]. Of particular notice for us is recent work by Gupta et al [19] for the set cover problem (and Bhattacharya et al [6], restricted to vertex cover) in the dynamic setting where elements arrive and depart.

In this paper, we study the *dynamic sum-of-radii clustering problem*, defined as follows: The original input consists of a (possibly infinite) set V of potential *clients* or *points*, a finite set $F \subseteq V$ of *facilities* with an *opening cost* f_j for each facility j, and a metric d over V. For the online input, a set C of live clients evolves over time: at each timestep t, either a new client arrives and is added to C, or a client from C departs and is removed from C, or a query is made for the approximate cost of an optimal solution (*cost query*), or a query asks for the entire current solution (*solution query*). For the output, at each timestep the algorithm maintains a set of *open* facilities, each open facility j being associated to a radius R_j, such that every client of C is *covered*, i.e. belongs to some open ball $B(j, R_j)$, and the goal is to minimize the *cost*, namely, the sum over open facilities j of $f_j + R_j$.

The dynamic sum-of-radii clustering problem can actually be interpreted as a special case of dynamic set cover: our metric space is the universe, our clients are the elements, and for each center c and radius r, the ball $B(c, r)$ defines a set of cost r consisting of those clients covered by the ball. The dynamic set cover algorithm from [19], specialized to our setting, maintains competitive ratio $O(\log n)$ and has update time $O(f \log n)$, where f is the maximum number of sets containing any element; [19] also gives an $O(f^3)$ approximation in time $O(f^2)$ for set cover, and for the related dynamic k-coverage problem, they give a constant approximation fully dynamic algorithm with $O(f \log n)$ update time.

The *doubling dimension* of a metric space (V, d) is said to be bounded by κ if any ball $B(x, r)$ in (V, d) can be covered by 2^κ balls of radius $r/2$ [22]. For example, D-dimensional Euclidean space has doubling dimension $\Theta(D)$.

In this paper we give an algorithm for constant doubling dimension, that maintains a solution whose cost is within a constant factor of the optimal cost, and with logarithmic update times for arrival or departure events. The algorithm answers cost queries in constant time and solution queries in linear time in the size of the solution, up to a factor of $\log(W/f_{min})$, where W is the diameter of the metric space and f_{min} is the minimum opening cost of any facility (which is strictly positive without loss of generality, since facilities of cost 0 can remain open at all times). The universe is known ahead of time, and only the collection of active clients changes dynamically. Note that for the above mentioned algorithm by [19] f would be $\Theta(2^{2\kappa} \log(W/f_{min}))$.

▶ **Theorem 1.** *There exists an algorithm for the dynamic sum-of-radii clustering problem, when clients and facilities live in a metric space with doubling dimension κ, such that at every timestep the solution has cost at most $\mathrm{O}(2^{2\kappa})$ times the cost of an optimal solution at that time, and such that the update time is $\mathrm{O}(2^{6\kappa} \log(W/f_{min}))$, where W is the diameter of the space and in the current number of clients and f_{min} is the minimum opening cost. A cost query can be answered in constant time, a solution query in time $\mathrm{O}(s \log(W/f_{min}))$, where s is the size of the output.*

The $2^{6\kappa}$ factor in the update time is due to fixed-radius nearest neighbor query that we solve using only the basic data structure already present in the algorithm. However, it can be improved – for example in case of finite metric spaces where a $\mathrm{O}(1)$ lookup table over the space is possible (e.g., graphs with the shortest path distance), the update time reduces to $\mathrm{O}(\log(W/f_{\min}))$ at the cost of additional preprocessing time. More generally, if the metric space allows to answer fixed-radius nearest neighbor queries in time $\mathrm{O}(d)$, then the update time becomes $\mathrm{O}(d \log(W/f_{\min}))$, at the cost of preprocessing time necessary to construct the oracle.

We show the following structural property for the sum-of-radii clustering problem (Theorem 8): there exists a collection Π of pairs $\langle j, r\rangle$ where $j \in F$ and r is a non-negative integer, each with an associated area $A(j, r)$ of V, and an abstract tree $\mathcal{T}$ over Π, with the following properties:

1. $\mathcal{T}$ has height $\mathrm{O}(\log(W/f_{\min}))$ and degree at most $2^{4\kappa}$.
2. The collection $\mathcal{A}$ of areas is a laminar family, its laminar structure is given by $\mathcal{T}$, and for each area, $A(j, r) \subseteq B(j, 7 \cdot 5^r)$
3. For any subset $\mathcal{C}$ of V, there exists a collection $\mathcal{S}$ of areas covering $\mathcal{C}$ and whose cost, $\sum_{\langle j,r\rangle \in \mathcal{S}} f_j + 7 \cdot 5^r$, is $O(2^{2\kappa})$ times the optimal cost for C.

Our algorithm has two phases. First, in the preprocessing phase (Section 2), the algorithm constructs Π, the laminar family of areas $\mathcal{A}$ and corresponding abstract tree $\mathcal{T}$. Thanks to the last property above, it suffices to restrict attention to solutions that use only areas $A(j, r)$ for coverage, with $\langle j, r\rangle \in \Pi$. Second, in the dynamic phase (Section 3), while clients arrive and depart, the algorithm maintains an optimal set of pairs $\langle j, r\rangle$ of Π such that the corresponding areas $A(j, r)$ cover all current clients. The hierarchical structure of $\mathcal{T}$ makes this simple, so that each update takes time proportional to the height times $2^{6\kappa}$.

The main contribution of the paper is the definition of Π and the corresponding laminar family of areas $\mathcal{A}$. The latter is reminiscent of the cover tree data structure of [5]. However, the cover tree is tailored to the nearest neighbor problem and its covers, to the best of our knowledge, lack the structural properties of areas that we need to prove the approximation

factor for the sum-of-radii clustering problem. We expect that our new structure can be used for other clustering-type problems.

2 Preprocessing phase

2.1 Discretization of radii

Given the set of clients $\mathcal{C} \subseteq V$, let OPT denote the cost of an optimum solution for $\mathcal{C}$.

▶ **Lemma 2.** *For all $\mathcal{C} \subseteq V$, there exists a solution such that every ball $B(j, R)$ has $f_{\min} \leq f_j \leq R \leq 5 \cdot W$, the radius R is an integer power of 5, and the cost is $O(OPT)$.*

Proof. Consider the unknown optimal solution. If some ball is such that $\max(f_j, R) > W$ then replace the entire solution by a ball centered at the facility of cost $f_{\min}$ and of radius W. Else, for each ball $B(j, R)$ of the optimal solution:

- if $f_j > R$ then increase the radius of the ball from R to f_j
- Increase R to the smallest integer power of 5 that is greater than or equal to R.

The new solution satisfies the desired constraints, and the cost has increased by a factor of 10 at most. ◀

A *logradius* is an integer r such that $f_{\min} \leq 5^r \leq 5 \cdot W$. Let $\rho_{\min} = \lfloor \log_5 f_{\min} \rfloor$ and $\rho_{\max} = \lceil \log_5 W \rceil$. Then the number of different logradii, $\rho_{\max} - \rho_{\min} + 1$, is $\mathrm{O}(\log(W/f_{\min}))$.

2.2 Maximal subsets of distant facilities

We construct a set Π of pairs $\langle j, r \rangle$ where j is a facility and r is a logradius, satisfying the following properties:

1. (Covering) For every facility $j \in J$ and every logradius r such that $f_j \leq 5^r$, there exists a facility $j' \in J_r$ with $d(j, j') \leq 5^{r+1}$ and $\langle j', r \rangle \in \Pi$.
2. (Separating) For all distinct $\langle j', r \rangle, \langle j', r \rangle \in \Pi$, we have $d(j, j') > 5^{r+1}$.

For each logradius $r \in [\rho_{\min}, \rho_{\max}]$:
- let $J'_r = \{j \in F \mid f_j \leq 5^r\}$.
- let J_r be a maximal subset of J'_r such that any two facilities in J_r are at distance greater than 5^{r+1}.

$\Pi \leftarrow \bigcup_r \{\langle j, r \rangle \mid j \in J_r\}$.

Note that for $r = \rho_{\max}$, the set J_r contains just one facility.

2.3 Hierarchical decomposition of Π

Construct an abstract tree $\mathcal{T}$ over Π as follows (with ties broken arbitrarily):
- the root of $\mathcal{T}$ is the unique pair $\langle j, \rho_{\max} \rangle$.
- for all $r < \rho_{\max}$ and $j \in J_r$:
  - let j' be the facility of J_{r+1} closest to j
  - $\mathrm{parent}(j, r) \leftarrow \langle j', r + 1 \rangle$

By construction, $\mathcal{T}$ has height at most $\rho_{\max} - \rho_{\min} + 1$ and the parent of a pair $\langle j, r \rangle$ is a pair of the form $\langle j', r + 1 \rangle$.

The following Lemma is simple, but it captures the essential way in which using larger balls will greatly simplify the structure, and is the main step towards constructing a laminar set of areas for covering clients.

▶ **Lemma 3.** *(Nesting of balls) If* $\text{parent}(j,r) = \langle j', r+1\rangle$*, then* $B(j, 7\cdot 5^r) \subseteq B(j', 7\cdot 5^{r+1})$.

Proof. We have $\langle j,r\rangle \in \Pi$, so $j \in J'_r \subseteq J'_{r+1}$. By the Covering property of Π the maximum distance from any point in $B(j, 7\cdot 5^r)$ to j' is $d(j,j') + 7\cdot 5^r \leq 5^{r+2} + 7\cdot 5^r \leq 7\cdot 5^{r+1}$. ◀

▶ **Lemma 4.** *For any point* p *and radius* r*, the set of pairs*

$$\Pi(p,r) = \{\langle j,r\rangle \in \Pi \mid d(p,j) < 2^\alpha 5^{r+1}\}$$

has at most $2^{(\alpha+1)\kappa}$ *elements, where* κ *is the doubling dimension of the metric space.*

Proof. By definition of doubling dimension, $B(p, 2^\alpha \cdot 5^{r+1})$ can be covered by a set of at most $(2^\kappa)^{\alpha+1}$ balls of radius $(1/2)\cdot 5^{r+1}$. By the Separating property of Π, any two pairs $\langle j,r\rangle$ of $\Pi(p,r)$ are at distance greater than 5^{r+1} from each other, hence must belong to different balls of the set, and so $\Pi(p,r)$ has cardinality at most $(2^\kappa)^{\alpha+1}$. ◀

▶ **Lemma 5.** *A node* $\langle j,r\rangle$ *of* $\mathcal{T}$ *has at most* $2^{4\kappa}$ *children*

Proof. Children of $\langle j,r\rangle$ have logradius $r-1$, so by the Covering property of Π their distance to j is at most 5^{r+1}, so they belong to $\Pi(j, r-1)$ for $\alpha = 3$, and so Lemma 4 applies. ◀

2.4 Hierarchical decomposition of V into a laminar family of areas

Recall that a collection $\mathcal{A}$ of sets is *laminar* if for any two $A, B \in \mathcal{A}$, either $A \cap B = \emptyset$ or $A \subseteq B$ or $B \subseteq A$. We partition V into a laminar family of *areas*, denoted by $\mathcal{A}$, such that no two same-logradius areas overlap.

For each $\langle j,r\rangle \in \Pi$, initialize $A(j,r) \leftarrow \emptyset$.
For each point $p \in V$:
- let r^* be minimum such that there exists pairs $\langle j, r^*\rangle$ with $p \in B(j, 7\cdot 5^{r^*})$.
- Among all such pairs, let $\langle j^*, r^*\rangle$ denote the one minimizing $d(p, j^*)$.
- Add p to the set $A(j^*, r^*)$ and to every set $A(j', r')$ with (j', r') ancestor of (j^*, r^*) in $\mathcal{T}$.

▶ **Lemma 6.** *For every* $\langle j,r\rangle \in \Pi$*,* $A(j,r) \subseteq B(j, 7\cdot 5^r)$.

Proof. Let $p \in A(j,r)$. Either it's been added directly, in which case it belongs to $B(j, 7\cdot 5^r)$, or it's been inherited, in which case it also belongs to it by Lemma 3. ◀

▶ **Lemma 7.** *For every subset* $\mathcal{C} \subseteq V$ *of clients there exists* $S \subseteq \Pi$ *such that* $\mathcal{C}$ *is covered by* $\cup\{A(j,r) : \langle j,r\rangle \in S\}$ *and* $\sum_{\langle j,r\rangle \in S}(f_j + 7\cdot 5^r) = \mathrm{O}(2^{2\kappa}\cdot OPT)$.

Proof. Let S^* be a solution of cost $\mathrm{O}(OPT)$ satisfying the properties of Lemma 2. For each ball $B(j,5^r)$ of S^*, put in S all the pairs $\langle j', r\rangle \in \Pi$ such that $d(j,j') \leq 8\cdot 5^r$.

We claim that $\mathcal{C}$ is covered by $\cup\{A(j',r) : \langle j',r\rangle \in S\}$. Indeed, consider a client $p \in \mathcal{C}$ and a ball $B(j,5^r)$ of S^* containing p. By the Covering property of Π, there exists $\langle j',r\rangle \in \Pi$ with $d(j,j') \leq 5^{r+1}$. Then $d(p,j') \leq 5^{r+1} + 5^r < 7\cdot 5^r$, and so in the definition of areas covering p we must have $r^* \leq r$. Along the path from $\langle j^*, r^*\rangle$ to the root of $\mathcal{T}$, there exists a pair for logradius r, $\langle j'', r\rangle$. By definition of areas and by Lemma 6, $p \in A(j'',r) \subseteq B(j'', 7\cdot 5^r)$, so $d(j,j'') \leq d(j,p) + d(p,j'') \leq 8\cdot 5^r$, and therefore $\langle j'', r\rangle \in S$ and p is covered.

In terms of costs, since all these areas are associated to pairs within distance $8\cdot 5^r < 2\cdot 5^{r+1}$ from j, by Lemma 4 for $\alpha = 1$, there are at most $2^{2\kappa}$ of them. ◀

By Lemma 6 and the definition of areas, we note that $\cup_{J_r} A(j,r) = \cup_{J_r} B(j, 7\cdot 5^r)$, so we also give hereafter an equivalent description of the same laminar family, illustrating the way in which the parent-child relations in tree $\mathcal{T}$ and the proximity relations in the metric space are balanced against one another. (This also has the advantage of being constructive even if V is infinite).

Partition $\cup_{J_{\rho_{\min}}} B(j, 7\cdot 5^{\rho_{\min}})$, using the facilities of $J_{\rho_{\min}}$ as centers, into Voronoi cells $A(j, \rho_{\min})$.

For $r \in (\rho_{\min}, \rho_{\max}]$:

- Partition $\cup_{j\in J_r} B(j, 7\cdot 5^r) \setminus \cup_{j\in J_{r-1}} B(j, 7\cdot 5^{r-1})$, using the facilities of J_r as centers, into Voronoi cells $A(j,r)$.
- For each $\langle j,r\rangle \in \Pi$, $A(j,r) \leftarrow A(j,r) \cup \bigcup\{A(j', r-1) : \text{parent}(j', r-1) = \langle j,r\rangle\}$

The construction of this section can be summarized in the following structural Theorem.

▶ **Theorem 8.** *Let a metric space (V,d) of doubling dimension κ be given, as well as a subset F of elements of V called* facilities*, with an associated* cost *f_j for each $j \in F$. Then there exists an abstract tree $\mathcal{T}$ whose nodes are indexed by facilities $j \in F$ and non-negative integers $r \geq 0$, and, for each node $\langle j,r\rangle$, an associated* area *$A(j,r) \subseteq V$ with the following properties*

1. *$\mathcal{T}$ has height $\mathrm{O}(\log(W/f_{\min}))$ and degree at most $2^{4\kappa}$, and for each $\langle j,r\rangle \in \mathcal{T}$ and its parent node $\langle j', r+1\rangle$, $B(j, 7\cdot 5^r) \subseteq B(j', 7\cdot 5^{r+1})$.*
2. *$\mathcal{A}$ is a laminar family, its laminar structure is given by $\mathcal{T}$, and for each area $A(j,r)$, $A(j,r) \subseteq B(j, 7\cdot 5^r)$*
3. *For any subset $\mathcal{C}$ of V, for any collection of balls $\mathcal{B}$ centered at facilities of F and covering $\mathcal{C}$, there exists a collection $\mathcal{S}$ of areas covering $\mathcal{C}$, such that $\sum_{\langle j,r\rangle\in\mathcal{S}} f_j + 7\cdot 5^r = O(2^{2\kappa}) \sum_{B(j,R)\in\mathcal{B}} (f_j + R)$.*

3 Data structure

3.1 Solving the offline restricted problem

Given $\mathcal{C} \subseteq V$, we wish to compute the solution of minimum cost among all *solutions that are restricted to covering $\mathcal{C}$ using areas $A(j,r)$ for $\langle j,r\rangle \in \Pi$*, where using area $A(j,r)$ has cost $f_j + c_2 \cdot 5^r$. We call that the *restricted* problem. By Theorem 8 the optimal restricted cost is a $O(2^{2\kappa})$ approximation of the optimal (unrestricted) cost.

Computing the optimal solution to the restricted problem in an offline manner is straightforward, thanks to the laminar structure of the candidate areas. We first compute, for each node $\langle j,r\rangle$ of $\mathcal{T}$, the cost $c_{j,r} = f_j + c_2\cdot 5^r$ of area $A(j,r)$, as well as the number $n_{j,r}$ of clients that are in area $A(j,r)$ but not in any of the areas of children nodes: since areas $A(j', r-1)$ are all disjoint by laminarity, we have $n_{j,r} = |\mathcal{C}\cap A(j,r)| - \cup_{\langle j', r-1\rangle : \text{parent}(j', r-1) = j} |\mathcal{C}\cap A(j', r-1)|$. We then compute the optimal cost $x_{j,r}$ of covering the clients of $\mathcal{C}\cap A(j,r)$ using only areas of the subtree of $\mathcal{T}$ rooted at $\langle j,r\rangle$, using the following bottom-up recurrence:

For $\langle j,r\rangle \in \Pi$ in bottom-up order in $\mathcal{T}$:

$$x_{j,r} = \begin{cases} c_{j,r} \text{ if } n_{j,r} > 0 \\ \min\left(c_{j,r}, \sum\{x_{j',r-1} : \langle j,r\rangle = \text{parent}(j', r-1)\}\right) \text{ otherwise.} \end{cases}$$

Indeed, if $n_{j,r} \neq 0$ then the solution must use area $A(j,r)$; but then by laminarity area $A(j,r)$ covers all clients in that subtree, so no other area is needed in the solution, and

the cost is exactly the cost $c_{j,r}$ of $A(j,r)$. If on the other hand $n_{j,r} = 0$, then we have an alternative possibility: we could do without $A(j,r)$. Then, by disjointness of sibling areas the problem separates into independent subproblems, one for each child of $\langle j,r\rangle$, hence the recurrence simply sums their costs.

The cost of the optimal restricted solution is then $x_{j,\rho_{\max}}$ for the root $\langle j,\rho_{\max}\rangle$ of $\mathcal{T}$.

Given $c_{j,r}$ and $x_{j,r}$, computing the optimal restricted solution, a collection S of areas, is done recursively:

$$S(j,r) = \begin{cases} \emptyset \text{ if } x_{j,r} = 0 \\ \{A(j,r)\} \text{ if } x_{j,r} = c_{j,r} \\ \cup\{S(j',r-1) : \text{parent}(j',r-1) = \langle j,r\rangle\} \text{ otherwise.} \end{cases}$$

Thus the algorithm to compute the optimal set S of areas covering $\mathcal{C}$ in the restricted problem, given the values of $c_{j,r}, x_{j,r}$ explores a tree $\mathcal{T}'$ that, as it is a partial subtree of $\mathcal{T}$, also has height at most $O(\log(W/f_{\min}))$ and degree at most $2^{4\kappa}$; moreover its internal nodes are all ancestors of areas added to the solution S, so the running time to compute S itself is $O(2^{4\kappa}\log(W/f_{\min})|S|)$.

3.2 The dynamic data structure

The dynamic data structure supports insertions of clients, deletions of clients, queries for the cost of the optimal restricted solution, and queries for the set of open facilities and areas of the optimal restricted solution.

The algorithm will maintain two dynamic data structures:

1. a list of the currently existing clients $\mathcal{C} \subseteq V$, with, for each client p, the $\langle j,r\rangle \in \Pi$ such that $p \in A(j,r)$ and r is minimum; and
2. an *annotated dependency tree* $\mathcal{T}_A$, keeping for each node $v = \langle j,r\rangle$ the following additional information:
   a. its cost $c_v = f_j + 7\cdot 5^r$,
   b. the number n_v of currently existing clients that belong to $A(j,r)$ but not to any descendant area,
   c. the value x_v, which is the minimum cost needed to cover all clients belonging to $A(j,r)$ using only areas $A(j',r')$ for $\langle j',r'\rangle \in \Pi$, and
   d. the value $y_v = \sum_{u \text{ child of } v} x_u$.

To initialize the data structures, from the preprocessing phase the algorithm is given the set Π of pairs $\langle j,r\rangle$, as well as the laminar family of areas $\mathcal{A}$ with its dependency tree $\mathcal{T}$ using the following representation, which can be easily computed in time linear in its size: (1) An array of size $\rho_{\max} - \rho_{\min} + 1$, keeping for each logradius $r \in [\rho_{\min},\rho_{\max}]$ a list of all the facilities of J_r, and (2) An annotated tree data structure obtained from $\mathcal{T}$ by setting every n_v, x_v, y_v equal to 0, and $c_{j,r} = f_j + 7\cdot 5^r$. The initial set of clients is $\mathcal{C} = \emptyset$.

Answering queries is done as in Section 3.1.

We next describe the client deletions. When a client p is deleted, we start from $\langle j,r\rangle$ in $\mathcal{T}_A$, such that $p \in A(j,r)$ and r is minimum; we decrement n_v and we traverse the path from $\langle j,r\rangle$ up to the root of $\mathcal{T}_A$, updating x_v and $y_{\text{parent}(v)}$ for every node visited along the way using the recurrence from Section 3.1. This takes time proportional to the height of the tree, $O(\log(W/f_{\min}))$.

Similarly, when a client p is inserted, we first find $\langle j, r\rangle$ in $\mathcal{T}_A$, such that $p \in A(j, r)$ and r is minimum, in a way to be described shortly; we increment n_v, and then we traverse the path from $\langle j, r\rangle$ up to the root of $\mathcal{T}_A$, similarly updating x_v and $y_{\text{parent}(v)}$.

Thus, it only remains to determine the pair $\langle j^*, r^*\rangle$ with smallest logradius such that $p \in A(j^*, r^*)$. By Lemma 6, $p \in B(j^*, 7 \cdot 5^{r^*})$. Thus we will first find all pairs $\langle j, r\rangle$ such that $p \in B(j, 7 \cdot 5^r)$, based on them determine r^*, and then look for $\langle j^*, r^*\rangle$ in that set of balls. Thanks to Lemma 3, the first part can be done using a simple recursive algorithm starting from the root of $\mathcal{T}_A$ (see below). The second part simply uses the definition of areas, i.e., it finds the pair $\langle j^*, r^*\rangle$ where j^* has with minimum distance to p out of all pairs $\langle j, r^*\rangle$ with$p \in B(j, 7 \cdot 5^{r^*})$.

$$\text{Pairs}(p, j, r) = \begin{cases} \emptyset \text{ if } p \notin B(j, 7 \cdot 5^r) \\ \{\langle j, r\rangle\} \cup \bigcup\{\text{Pairs}(p, j', r-1) : \text{parent}(j', r-1) = \langle j, r\rangle\} \text{ otherwise.} \end{cases}$$

- let r^* be minimum such that there exists pairs $\langle j, r^*\rangle$ in the set $\text{Pairs}(p, j_{\text{root}}, \rho_{\max})$.
- Among all such pairs, output the pair $\langle j^*, r^*\rangle$ minimizing $d(p, j^*)$.

The running time is dominated by the first part, which is $O(2^{4\kappa})$ times the number of pairs $\langle j, r\rangle$ such that $p \in B(j, 7 \cdot 5^r)$. There are $\log(W/f_{\min})$ possible values of r. For each r, by Lemma 4 there are at most $2^{2\kappa}$ pairs $\langle j, r\rangle \in \Pi$ such that $p \in B(j, c_2 \cdot 5^r)$ and the algorithm has to test the $O(2^{4\kappa})$ children of each of them. Thus the running time to do an insertion is $O(2^{6\kappa} \log(W/f_{min}))$.

References

1. Helmut Alt, Esther M. Arkin, Hervé Brönnimann, Jeff Erickson, Sándor P. Fekete, Christian Knauer, Jonathan Lenchner, Joseph S. B. Mitchell, and Kim Whittlesey. Minimum-cost coverage of point sets by disks. In *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, SCG'06, pages 449–458, New York, NY, USA, 2006. ACM. doi:10.1145/1137856.1137922.
2. Sayan Bandyapadhyay and Kasturi R. Varadarajan. Approximate clustering via metric partitioning. *CoRR*, abs/1507.02222, 2015. URL: http://arxiv.org/abs/1507.02222.
3. Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in o(log n) update time. *SIAM J. Comput.*, 44(1):88–113, 2015. doi:10.1137/130914140.
4. Babak Behsaz and Mohammad R. Salavatipour. On minimum sum of radii and diameters clustering. *Algorithmica*, 73(1):143–165, September 2015. doi:10.1007/s00453-014-9907-3.
5. Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In William W. Cohen and Andrew Moore, editors, *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, volume 148 of *ACM International Conference Proceeding Series*, pages 97–104. ACM, 2006. doi:10.1145/1143844.1143857.
6. Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in $o(1)$ amortized update time. *CoRR*, abs/1611.00198, 2016. URL: http://arxiv.org/abs/1611.00198.
7. Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Design of dynamic algorithms via primal-dual method. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 206–218, 2015. doi:10.1007/978-3-662-47672-7_17.

8 Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 785–804, 2015. doi:10.1137/1.9781611973730.54.

9 Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 398–411, 2016. doi:10.1145/2897518.2897568.

10 Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $o(\log^3 n)$ worst case update time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 470–489, 2017. doi:10.1137/1.9781611974782.30.

11 Niv Buchbinder and Joseph (Seffi) Naor. The design of competitive online algorithms via a primal: Dual approach. *Found. Trends Theor. Comput. Sci.*, 3(2–3):93–263, February 2009. doi:10.1561/0400000024.

12 Fazli Can. Incremental clustering for dynamic information processing. *ACM Trans. Inf. Syst.*, 11(2):143–164, April 1993. doi:10.1145/130226.134466.

13 Moses Charikar and Rina Panigrahy. Clustering to minimize the sum of cluster diameters. *Journal of Computer and System Sciences*, 68(2):417–441, 2004. doi:10.1016/j.jcss.2003.07.014.

14 János Csirik, Leah Epstein, Csanád Imreh, and Asaf Levin. Online clustering with variable sized clusters. *Algorithmica*, 65(2):251–274, 2013. doi:10.1007/s00453-011-9586-2.

15 Srinivas R. Doddi, Madhav V. Marathe, Sekharipuram S. Ravi, David S. Taylor, and Peter Widmayer. Approximation Algorithms for Clustering to Minimize the Sum of Diameters. *Nordic journal of computing*, 7(3):185–203, 2000.

16 Dimitris Fotakis and Paraschos Koutris. Online sum-radii clustering. *CoRR*, abs/1109.5325, 2011. URL: http://arxiv.org/abs/1109.5325.

17 Matt Gibson, Gaurav Kanade, Erik Krohn, Imran A. Pirwani, and Kasturi Varadarajan. On clustering to minimize the sum of radii. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'08, pages 819–825, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=1347082.1347172.

18 Matt Gibson, Gaurav Kanade, Erik Krohn, Imran A. Pirwani, and Kasturi Varadarajan. On metric clustering to minimize the sum of radii. In *Proceedings of the 11th Scandinavian Workshop on Algorithm Theory*, SWAT'08, pages 282–293, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-69903-3_26.

19 Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. *CoRR*, abs/1611.05646, 2016. URL: http://arxiv.org/abs/1611.05646.

20 P. Hansen and B. Jaumard. Minimum sum of diameters clustering. *Journal of Classification*, 4(2):215–226, 1987.

21 Pierre Hansen and Brigitte Jaumard. Cluster analysis and mathematical programming. *Math. Program.*, 79(1-3):191–215, October 1997. doi:10.1007/BF02614317.

22 Robert Krauthgamer and James R. Lee. Navigating nets: Simple algorithms for proximity search. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'04, pages 798–807, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=982792.982913.

23 Nissan Lev-Tov and David Peleg. Polynomial time approximation schemes for base station coverage with minimum total radii. *Comput. Netw. ISDN Syst.*, 47(4):489–501, March 2005. doi:10.1016/j.comnet.2004.08.012.

24 Guido Proietti and Peter Widmayer. Partitioning the nodes of a graph to minimize the sum of subgraph radii. In *Proceedings of the 17th International Conference on Algorithms and Computation*, ISAAC'06, pages 578–587, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11940128_58.

25 Satu Elisa Schaeffer. Survey: Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, August 2007. doi:10.1016/j.cosrev.2007.05.001.

26 Shay Solomon. Fully dynamic maximal matching in constant update time. *IEEE FOCS*, 2016. URL: http://arxiv.org/abs/1604.08491.

Shortest Paths in the Plane with Obstacle Violations

John Hershberger[1], Neeraj Kumar[2], and Subhash Suri[3]

1 **Mentor Graphics Corp., Wilsonville, OR, USA**
`john_hershberger@mentor.com`
2 **University of California, Santa Barbara, CA, USA**
`neeraj@cs.ucsb.edu`
3 **University of California, Santa Barbara, CA, USA**
`suri@cs.ucsb.edu`

Abstract

We study the problem of finding shortest paths in the plane among h convex obstacles, where the path is allowed to pass through (violate) up to k obstacles, for $k \leq h$. Equivalently, the problem is to find shortest paths that become obstacle-free if k obstacles are removed from the input. Given a fixed source point s, we show how to construct a map, called a *shortest k-path map*, so that all destinations in the same region of the map have the same combinatorial shortest path passing through at most k obstacles. We prove a tight bound of $\Theta(kn)$ on the size of this map, and show that it can be computed in $O(k^2 n \log n)$ time, where n is the total number of obstacle vertices.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Shortest paths, Polygonal obstacles, Continuous Dijkstra, Obstacle crossing, Visibility

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.49

1 Introduction

Given a set of polygonal obstacles in the plane and an integer parameter k, which k obstacles should we remove to obtain the shortest obstacle-free path between two points s and t? Equivalently, what is the shortest path that is allowed to violate (pass through) up to k obstacles? We call a path violating at most k obstacles a *k-path*, generalizing a traditional obstacle-free path, which is a 0-path. More precisely, we assume a polygonal environment P containing h disjoint convex obstacles in the plane, with a total of n vertices, all lying inside a rectangle R (the outer boundary). The complement of the obstacles within R is called *free space*. Given a fixed source point s in free space, we want to compute shortest k-paths, for $k \leq h$, to all other points of free space. The description of these shortest paths can be compactly encoded as a finite partition of the plane, called the *shortest k-path map*. We use the notation $\pi_k(t)$ to denote the shortest k-path from s to t, with the fixed source s being implicit, and denote the length of this path by $d_k(t)$.

In this paper, we investigate structural and computational aspects of shortest k-paths. The problem differs from the 0-path problem in nontrivial ways even in the plane. In particular, two shortest 0-paths originating at a common source cannot intersect, by the triangle inequality, and this non-crossing property of 0-paths is an essential ingredient for computing them in optimal time [15]. In contrast, two shortest k-paths can cross each other, for any $k > 0$. The geometric k-path problem is interesting both theoretically, as

part of the broad category of optimization with violations [6, 21] or network augmentation problems [2, 10], and practically, for applications such as robot motion planning, where it may be beneficial to modify a robot's environment to shorten frequently used paths. (The geometric k-path problem can be seen as a more complex form of network augmentation, since removal of a single obstacle can create many additional "edges" in the path space.) Besides robot motion planning, the problem can also model situations in which the obstacles are "avoidable" at additional cost, for instance by paying a bridge or tunnel toll in a road network.

Our approach to solving the k-path problem is to compute a *shortest k-path map* SPM_k, which is a partition of the plane into equivalence classes of cells (regions), where all destination points inside a cell have the same combinatorial structure of shortest k-paths to s. Once the map is known, the shortest k-path to any destination can be computed by performing a point location query on the map [8, 18].

Our Results. We show that SPM_k has $O(kn)$ regions and $O(kn)$ edges and that this bound is tight (Section 3). We present an $O(k^2 n \log n)$ time algorithm for computing SPM_k (Section 4), using the continuous Dijkstra framework, which constructs each SPM_j for $0 \leq j \leq k$ sequentially. The running time of the algorithm is optimal for $k = O(1)$. Due to space limitations, some of the proofs are omitted from this version of the paper.

Related Work. The problem of computing shortest paths in the presence of obstacles has a long history in computational geometry, dating back to the 1970s. The case of polygonal obstacles in the plane, in particular, has been a subject of intense research [3, 4, 11, 17, 22, 23, 25, 26, 28], culminating in an optimal $O(n \log n)$ time algorithm using the continuous Dijkstra framework [15]. Many other variations of the problem, including shortest paths inside a simple polygon [12, 14, 19], among weighted regions [24], and among curved obstacles [7, 16], have also been studied. The general flavor of our problem is related to geometric optimization where a small number of constraints can be violated. This line of work has been pursued in [6, 13, 21, 27], in the context of low-dimensional linear programming, separability with outliers, and geometric optimization. Our problem can also be viewed as a form of network augmentation, where the goal is to add edges to the network to improve connectivity, diameter, or spanning ratio etc. [1, 2, 5, 10].

The prior work most closely related to our research is a recent result by Maheshwari et al. [20], which presents an $O(n^3)$ time algorithm for computing the 1-violation path inside a simple polygon: that is, a shortest path inside a simple n-gon that is allowed to leave the polygon once. Our work deals with finding k-violation paths, for arbitrary k, in an environment containing possibly $O(n)$ convex obstacles.

2 Properties of k-paths

Given a point p in free space, a shortest *k-path* $\pi_k(p)$ connects s to p, crosses the interiors of at most k obstacles, and has minimum length among all such paths. On occasion, we also need to reason about paths crossing *exactly* k obstacles, and we refer to such a path as an $(= k)$-path. We begin with the easy observation that the problem can be solved in polynomial (quadratic) time, using a Dijkstra-like search on a "visibility graph."

▶ **Theorem 1.** *Given a polygonal domain P with h convex obstacles and n vertices, a source point s and a destination t, we can compute a shortest k-path from s to t in worst-case time $O((kn + h^2) \log n + kh^2)$.*

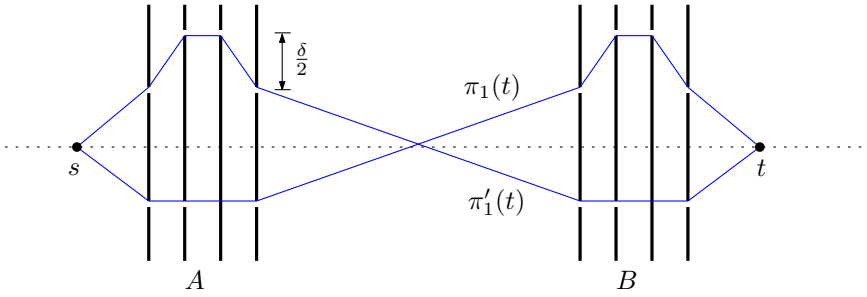


Figure 1 Two intersecting 1-paths.

The visibility graph-based approach is inherently quadratic in the worst case, because the number of obstacles can be $h = \Omega(n)$. It also is limited to computing the shortest k-path to only one point (or a fixed set of points) at a time, although it can be extended to support queries in $O(h(k + \log n))$ time apiece after quadratic preprocessing.

The main result of our paper is an algorithm to compute shortest k-paths from s to all points of free space in *sub-quadratic* time $O(k^2 n \log n)$. We do this by computing a *shortest k-path map* of free space; we also prove a tight bound of $\Theta(kn)$ on the combinatorial complexity of SPM_k. Note that the length of a shortest k-path to a point is unique, although some points (along bisectors forming the boundaries of regions in the shortest path map) can be reached by multiple shortest k-paths. For simplicity, however, we assume that the obstacles are in general position, so that the shortest k-path to each obstacle vertex is unique. (Otherwise, if a vertex is reached from s by multiple shortest k-paths, we pick one of them arbitrarily.)

We begin by highlighting a conceptual difficulty with shortest k-paths. The shortest paths to two different destinations can cross each other, which poses an inherent difficulty for the continuous Dijkstra framework of geometric shortest paths [15], since that method depends on the fact that two Euclidean shortest paths from a common source cannot intersect.

▶ **Lemma 2.** *There exist obstacle configurations such that for two destinations t_1, t_2 in free space, the shortest k-paths $\pi_k(t_1)$ and $\pi_k(t_2)$ cross each other, for $k > 0$.*

Proof. The construction, shown in Figure 1, has two identical obstacle bundles A and B placed parallel to the y-axis. Each bundle contains four vertical strips with perforations (single-point openings that split the original strip into disjoint sub-strips). The horizontal spacing between the strips in a bundle is infinitesimal, but for clarity the strips are shown separated in the figure. The points s and t both lie on the x-axis at distance 1 to the left and right of bundles A and B, respectively. We show that there are two shortest 1-paths from s to t, which cross each other, as shown in the figure. We then conclude that by perturbing t up and down slightly we obtain two destination points t_1 and t_2 with their shortest 1-paths crossing, as claimed.

Within each bundle, the openings form an *upper* and a *lower* group. In the upper group, strips 2 and 3 have an opening at $y = (1 + \delta/2)$, and strips 1 and 4 have openings at $y = 1$. In the lower group, all except strip 3 have an opening at $y = -1$. If the distance between the bundles is D, then a shortest 0-path has length $2\sqrt{2} + D + 2\delta$, and a shortest 2-path has length $2\sqrt{2} + D$. A path with exactly one crossing in an upper group has length at least $2\sqrt{2} + D + 3\delta/2$, and a shortest path with one crossing in a lower group has length $2\sqrt{2} + \sqrt{D^2 + 4} + \delta < 2\sqrt{2} + D + 2/D + \delta$. By choosing $D = 10$, say, and $\delta = 4/D$, we can force a shortest 1-path to go through exactly one group of each type. This gives two

intersecting shortest k-paths, $\pi_1(t)$ and $\pi'_1(t)$. Now, let t_1 (resp. t_2) be a destination point obtained by shifting t vertically up (resp. vertically down) infinitesimally. Then it is easy to see that the shortest 1-paths $\pi_1(t_1)$ and $\pi_1(t_2)$ cross each other. ◀

Fortunately, as we show in this section, shortest k-paths can always be decomposed into appropriate non-crossing subpaths to which the continuous Dijkstra method can be applied, working on multiple copies of free space connected using the metaphor of a *k-level garage*. Toward that goal, we establish a series of lemmas.

▶ **Lemma 3.** *A shortest path with exactly k crossings can be decomposed into a shortest path with exactly $(k-1)$ crossings, a straight line segment inside an obstacle, and a shortest path with zero crossings.*

Proof. Let $\pi = (v_1, v_2, \ldots, v_m)$ be an $(= k)$-path from v_1 to v_m. Going backward from v_m along π, let v_i be the first vertex such that the segment $\overline{v_{i-1}v_i}$ intersects one or more obstacles. Let H be the obstacle that is closest to v_i along the segment $\overline{v_{i-1}v_i}$. By the convexity of H, the segment $\overline{v_{i-1}v_i}$ intersects H at two points, which we call p and q, and the segment $\overline{pq}$ lies entirely within H. By subpath optimality, the path from v_1 to p is a shortest path with exactly $k-1$ crossings; by construction, the segment $\overline{pq}$ lies inside the obstacle; and the subpath from q to v_m crosses no obstacles. ◀

▶ **Corollary 4.** *In a shortest k-path, the path segments preceding and following any obstacle crossing are collinear with the path segment inside the obstacle.*

Lemma 3 allows us to break any $\pi_k(t)$ into a $(k-1)$-path $\pi_{k-1}(p)$, a subpath line segment $\overline{pq}$, and an obstacle-free subpath between q and t. We label the last two subpaths with the number of obstacles crossed by the prefix of the path, and call these labels the *prefix counts*. In particular, the prefix count for the subpath $\overline{pq}$ is $k-1$, and the prefix count for the subpath from q to t is k. By a recursive application of Lemma 3, we can decompose $\pi_k(t)$ into $2k+1$ disjoint subpaths whose labels are in non-decreasing order.

The key consequence of this decomposition is the following lemma, which says that subpaths with the same prefix count cannot cross.

▶ **Lemma 5.** *Let $\pi_k(t)$ and $\pi'_k(t')$ be two subpaths whose prefix counts are the same. Then $\pi_k(t)$ and $\pi'_k(t')$ do not cross each other.*

Proof. The proof follows from a simple application of the triangle inequality: if two subpaths with the same prefix count intersect, then we can reconnect the prefix of each path to the suffix of the other, and possibly perform a local shortcut, either shortening at least one path or leaving them the same length but without a crossing. Since the intersecting subpaths are either both inside some obstacle or in free space, avoiding the intersection does not increase the number of obstacle crossings for either path. For instance, in the example in Figure 1, the intersecting edges of the two crossing shortest k-paths have different prefix counts. ◀

The next two lemmas establish properties of shortest k-paths that will be useful later.

▶ **Definition 6.** A point p is *k-visible* from the source s if the segment $\overline{sp}$ passes through at most k obstacles. A *k-visibility edge* is a shortest k-path with exactly one edge.

▶ **Lemma 7.** *If p is not $(k-1)$-visible from s, then the path $\pi_k(p)$ must be an $(= k)$-path.*

Proof. By contradiction. Suppose $\pi_k(p)$ passes through fewer than k obstacles. Since p is not $(k-1)$-visible from s, $\pi_k(p)$ must have at least one bend. The path can then be shortened by going through the obstacle causing this bend, thereby increasing the number of crossings by 1. The resulting path is shorter than $\pi_k(p)$ and has at most k crossings, contradicting the optimality of $\pi_k(p)$. ◀

Let $d_k(p)$ be the length of a shortest k-path to a point p. Clearly, a path that crosses j obstacles and contains at least two segments can be made even shorter if it is allowed to pass through more obstacles. Thus, it follows that for any point p that is not $(k-1)$-visible from s, we must have $d_j(p) > d_{j+1}(p)$, for $j < k$.

▶ **Lemma 8.** *For any point p that is not $(k-1)$-visible from s, the lengths of the shortest j-paths form a decreasing sequence:*

$$d_0(p) > d_1(p) > \ldots > d_i(p) > \ldots > d_k(p)$$

3 Shortest Path Map SPM_k: Properties and Bounds

Having established the basic properties of shortest k-paths, we now begin our discussion of the shortest k-path map SPM_k.

▶ **Definition 9.** Given a shortest k-path $\pi_k(p)$, we define the *k-predecessor* of p to be the vertex of P (including s) that is adjacent to p in $\pi_k(p)$. The partition of free space into connected regions with the same k-predecessor is called the *shortest k-path map*, and denoted SPM_k. The subset of SPM_k for which the shortest path $\pi_k(p)$ to every point p has exactly k crossings is called the shortest $(=k)$*-path map* and denoted by $SPM_{=k}$. See Figure 2 for an example.

Unlike SPM_0, in which the predecessor of a region is always inside or on the boundary of the region, the predecessor of a region in SPM_k may lie outside the region. Moreover, multiple regions in SPM_k may have the same predecessor. (See Figure 2.) Thus, we need to maintain additional information with polygon vertices to disambiguate the predecessor relation. In particular, let v be the *k-predecessor* of p, namely, the vertex adjacent to v in $\pi_k(p)$. Suppose the line segment $\overline{vp}$ crosses $(k-i)$ obstacles, for some $0 \le i \le k$. Then the length $d_k(p)$ of $\pi_k(p)$ is the sum of the length of the i-path to v and the length of segment $\overline{vp}$. We need to maintain the values $d_i(v)$ for all obstacle vertices v and all integers $i = 0, 1, \ldots, k$. In other words,

> For a point p in $SPM_{=k}$, we identify the k-predecessor of p by the pair (v, i), where v is a vertex of P and $i \in \{0, 1, \ldots, k\}$, such that $d_k(p) = d_i(v) + |\overline{vp}|$ and the segment $\overline{vp}$ crosses $(k-i)$ obstacles.

Thus, the total number of k-predecessors is $O(kn)$. However, this alone does not bound the number of regions in $SPM_{=k}$ because multiple regions can have the same k-predecessor and the same crossing sequence. Toward our goal of bounding the combinatorial complexity of the map, let us begin with the notion of k-visibility.

We define V_k to be the region consisting of k-visible points, which is star-shaped and therefore simply connected (Figure 3). Now if $\pi_k(p)$ crosses fewer than k obstacles, then by Lemma 7, p must lie in V_{k-1}. The path $\pi_k(p)$ is a straight line segment and the k-predecessor of p is s. Therefore, we have the following.

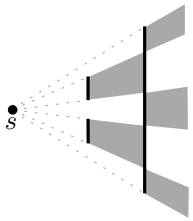


Figure 2 The 1-predecessor of all points in the shaded region of SPM_1 is $(s, 0)$.

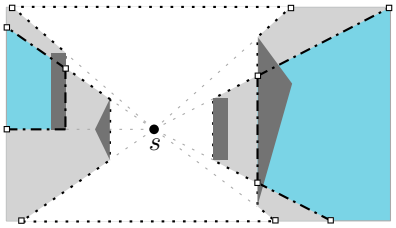


Figure 3 The boundary ∂V_1 of the region V_1 is dash-dotted, and it encloses the boundary ∂V_0, which is shown with dotted segments. The region $V_1 \setminus V_0$ is shown shaded gray.

▶ **Lemma 10.** *All points p such that $\pi_k(p)$ has fewer than k crossings lie in V_{k-1}. Outside of V_{k-1}, SPM_k is the same as $SPM_{=k}$, the shortest path map with exactly k crossings.*

This simplifies our discussion and allows us to decompose SPM_k into two distinct regions, V_{k-1} and $SPM_{=k}$. In the following, we study structural properties of these regions and use them to compute upper bounds on their respective sizes. Later, we combine them to compute an upper bound on the size of the map SPM_k.

3.1 k-Visibility Region

We first bound the complexity of the boundary of V_k, the region visible from s by a segment crossing at most k obstacles.

▶ **Lemma 11.** *The number of edges on the boundary ∂V_k is $O(n + h) = O(n)$.*

Proof. Every vertex of ∂V_k is either a vertex of P or a projection of one of the $2h$ tangents from s to an obstacle of P. The edges on the boundary ∂V_k are therefore sub-segments of the tangents or parts of obstacle boundaries. Each projection vertex belongs to a segment of ∂V_k collinear with s, and the endpoint x farther from s is the end of a maximal segment $\overline{sx}$ that crosses exactly k obstacles. Therefore, each of the $2h$ tangents gives rise to at most one segment of ∂V_k and at most two vertices. ◀

More interestingly, the bound on the total complexity of these regions is less than the sum of the individual bounds.

▶ **Lemma 12.** *The total number of edges on all ∂V_i, for $0 \le i \le k$, is $O(n + hk)$.*

By connecting s to all vertices on boundary ∂V_{k-1}, we can easily decompose V_{k-1} into constant complexity regions in SPM_k.

3.2 The k-Level Garage and the Structure of $SPM_{=k}$

We now introduce our main idea for computing the shortest k-path map. By Lemma 3, an $(= k)$-path from s to a point p is the concatenation of a $(k-1)$-path to the boundary of some obstacle H, a shortest path inside H, and a shortest path in free space from the other side of H to p. This suggests an incremental construction of $SPM_{=k}$ from $SPM_{=(k-1)}$. We describe this construction using the metaphor of a k-level *parking garage* with elevators.[1]

[1] The garage metaphor is also used in the context of finding homotopically different paths in [9], but the properties and technical details of our k-garage are quite different.

The idea is to create multiple copies of the input polygonal domain and stack them in levels such that the *shortest paths at each level have the same prefix count and therefore do not intersect.* The planar subdivision of free space at the top level is $SPM_{=k}$.

▶ **Definition 13** (k-garage)**.** We construct the *k-garage structure* by stacking k copies (or floors) of the input polygonal domain P on top of one another, with special connections at the obstacle boundaries. We connect the obstacle H on floor i to its counterpart on floor $i+1$ such that any path that enters H on floor i can exit only on the next higher floor—in a sense, obstacles act as elevators.

Our algorithm to construct $SPM_{=k}$ makes use of the *continuous Dijkstra method*, which simulates the expansion of a unit speed wavefront from the source s in free space. The wavefront at time T contains all points p whose shortest path distance from s is T. The boundary of the wavefront is a set of circular arcs called *wavelets*, each generated by an obstacle vertex (including s) already covered by the wavefront. The generating vertex v is called the *generator* of the wavelet and is identified by the pair (v, w), where w is the time at which v was reached by the wavefront. The generators can be thought of as sources *additively weighted* with delays, since they start emitting wavelets at time w after the start of the simulation. The locus of the meeting points of two adjacent wavelets is a *bisector* curve. Taken together with the obstacle boundaries, bisector curves partition free space into regions of the shortest path map.

We extend the continuous Dijkstra method to our k-garage structure. Each level of the garage is a plane with polygonal obstacles on which wavefronts propagate as usual, but the wavelets can now move to higher floors by entering the obstacles (elevators). More precisely, when the wavefront hits an obstacle H, it is absorbed by the outer boundary of H and is immediately re-emitted into the interior of H. When that wavefront reaches the inner boundary on the other (previously unreached) side of H, it is absorbed and immediately re-emitted on the next higher floor of the garage. This vertical movement therefore adds no delay. In this modified setting, the wavefront at time T contains points on all floors that are at distance T from the source.

The region V_{k-1} is removed from the polygonal domain on floor k of the k-garage because the shortest k-path is known for every point p in V_{k-1}—it is simply the line segment $\overline{sp}$—and leaving these points in the polygonal domain on floor k would create redundant copies of this path. We defer the exact details of our algorithm to Section 4. In the following, we note some properties of the k-garage structure useful to our algorithm.

1. If π is a shortest s–t path from s on floor 0 to t on floor k, then the downward projection $\pi^{\downarrow}$ of π, obtained by projecting π into the planar domain P, is a shortest k-path to t. (To see this, suppose for contradiction we have another k-path π_c from s to t that is shorter. Then by applying Lemma 3 recursively, we can break π_c into $2k+1$ disjoint subpaths ordered by their prefix counts. We now lift the paths into the levels of the garage and concatenate them in order: if the prefix counts of the current and the next subpath are the same, join their common endpoint at the same level as the prefix count; otherwise join their common endpoint at the next level. This transforms the path π_c into a shortest path $\pi_c^{\uparrow}$ from s on floor 0 to t on floor k. Since the vertical movement between the garage floors incurs no delay, the lifted path $\pi_c^{\uparrow}$ is shorter than π, which is a contradiction.)
2. Since wavefront propagation on floor i is affected only by wavelets coming from floors below it, we can think of wavefront propagation on floor i as occurring in a polygonal domain with *multiple* sources. On floor $i > 0$, all sources correspond to generators of wavelets coming from lower floors.

3. To compute the sources at floor $i > 0$, we need to consider only wavelets coming from floor $i - 1$. This follows from Lemma 8, which implies that even if wavelets were allowed to ascend multiple floors in an elevator, a wavelet from floor $i - 1$ would reach floor i no later than the wavelets from other lower floors.
4. The planar subdivision formed by bisectors of colliding wavelets on floor i is the shortest path map for $(= i)$-paths, $SPM_{=i}$. Note that since the obstacles are convex, a shortest path to a point on floor i cannot cross the same obstacle (on any floor) more than once, or else it can be made even shorter.

This suggests a natural way of computing the shortest path map $SPM_{=k}$. We construct maps $SPM_{=i}$ for $i = 0, 1, \ldots, k$ iteratively. Each iteration $i > 0$ is defined by ordinary shortest path propagation with a set of sources that come from the previous iteration. In the following section we use these observations to compute a bound on the size of the shortest k-path map SPM_k.

3.3 Complexity of SPM_k

The shortest k-path map SPM_k on the top floor of the k-garage is precisely $SPM_{=k}$ in the portion of free space that is outside V_{k-1}, as shown in Lemma 7. The boundary of V_{k-1} has linear size, and so we only need to bound the complexity of $SPM_{=k}$. To bound the complexity of $SPM_{=k}$, we consider the embedded planar graph G_k formed by $SPM_{=k}$, V_{k-1}, and the obstacle polygons. We note the following property of planar graphs, which is a direct consequence of Euler's formula.

▶ **Lemma 14.** *Let f be the number of faces in a planar graph $G = (V, E)$. If all the vertices of G have degree three or more, then the size of G is $O(f)$.*

Observe that the "interesting" vertices in G_k are the points where bisectors meet obstacle boundaries or meet each other, and therefore have degree at least three. If f is the number of faces, then by Lemma 14 the complexity of the map due to these vertices is $O(f)$. In addition to this, G_k can also have $O(n)$ vertices of degree two corresponding to the vertices of obstacle polygons, giving a total complexity bound of $O(f + n)$.

Therefore, in order to compute a bound on the complexity of $SPM_{=k}$, it suffices to bound the number of faces f in the graph G_k. We begin with the following well-known result [15].

▶ **Lemma 15.** *The shortest path map of m sources weighted by their delays in a polygonal domain with n vertices and h holes has $f \leq m + n + h \leq m + 2n$ faces. By planarity, the total complexity of the map is $O(f + n)$.*

The key to the proof of the preceding lemma is that each shortest path map region is star-shaped and connected to the predecessor of all points in the region. Since the total number of predecessors is at most $(m + n)$, the number of faces due to these regions is also at most $(m + n)$. Crucially, this lemma does not immediately apply to $SPM_{=k}$, because some predecessors of regions on the k^{th} floor belong to regions *below the k^{th} floor*. That is, some of the m sources are not in the polygonal domain, so the argument that each region is connected to its predecessor does not hold. Fortunately, the argument of Lemma 15 is a topological one, and we can create a topological domain in which the argument applies.

Every point $p \in \partial P$ outside of V_{k-1} is labeled by a $(k-1)$-crossing distance $d_{k-1}(p)$. If p belongs to an obstacle H, and there exists some $q \in \partial H$ such that $d_{k-1}(q) + |\overline{qp}| < d_{k-1}(p)$, then $\pi_k(p)$ may reach p by passing through H. The wavefront that determines $SPM_{=k}$ will be initialized with a weighted source that reaches p by "elevator" passing through H. If

$q \in \partial H$ minimizes $d_{k-1}(q) + |\overline{qp}|$, then the predecessor of q on $\pi_{k-1}(q)$ is the generator of the wavelet that first reaches p in the wavefront. We partition each edge of ∂H into maximal sub-edges with the same predecessor. For each sub-edge with predecessor v, we construct a triangular "flap" by drawing the segments from the sub-edge endpoints to v. Shortest paths propagate from v toward the k^{th} garage floor inside the flap, and in the pseudo-polygonal domain obtained by gluing all the flaps onto the boundary of free space, each shortest path map region is connected to its predecessor. If these flaps were projected into the plane, they would likely overlap, but topologically they do not alter the structure of the domain, and they add only two edges per flap.

▶ **Lemma 16.** *Let P be a polygonal domain with n vertices and h holes. If P is extended by gluing at most m triangular flaps to its boundary, then the shortest path map of m sources weighted by their delays in this extended polygonal domain has $f \leq m+n+h \leq m+2n$ faces and total complexity $O(m+n)$.*

The preceding lemma applies to the propagation of shortest paths on each floor of the k-garage and also to propagation inside the obstacles (elevators). In both cases the key to bounding the complexity of an iterated construction is bounding the number of sources that propagate into the next level, whether elevator or garage floor. In each elevator and on each garage level $i > 0$, the sources are located on the domain boundary. For simplicity we partition the sources at obstacle vertices, so each source is a maximal (sub-)edge ℓ on some obstacle boundary ∂H, with an associated generator (v, w). We refer to such a source as a *boundary source* and represent it by the triple (v, w, ℓ). Shortest paths from a source (v, w, ℓ) enter the domain through edge ℓ, and their predecessor is vertex v with weight (delay) w. As noted above, each boundary source defines a triangular flap glued onto the boundary of the propagation domain; the flap is the convex hull of ℓ and v.

When boundary sources propagate into some domain (either P or the interior of an obstacle), they define a shortest path map S in the domain. We say that if the region of S corresponding to a source $s = (v, w, \ell)$ intersects a domain edge, then s *claims* the intersection interval on that edge. An *entry claim* of a source (v, w, ℓ) is a claim on edge ℓ itself; entry claims can be ignored for further propagation, since a path that enters the domain through ℓ and exits through the same edge can be shortened. *Exit claims* (ones on edges other than ℓ) define the sources for the next level of shortest path propagation. Within any edge, a maximal sequence of exit claims with the same source is called an *exit claim cluster*. If an exit claim cluster on an edge e has source (v, w, ℓ), then the corresponding boundary source at the next level is (v, w, ℓ'), where ℓ' is the minimal subsegment of e containing the cluster. As noted, entry claims inside ℓ' do not affect shortest path propagation at the next level.

▶ **Lemma 17.** *Let S be the shortest path map obtained by propagating m boundary sources into a polygonal domain with n vertices. Then the number of exit claim clusters of S is at most $m + O(n)$.*

We are now ready to bound the complexity of $SPM_{=k}$.

▶ **Lemma 18.** *The number of faces f_k in $SPM_{=k}$ is $O(n(k+1))$. The complexity of $SPM_{=k}$ has the same asymptotic bound.*

Proof. The proof is by induction. Our goal is to show that there exists a constant C such that the number of faces f_k in $SPM_{=k}$ is at most $Cn(k+1)$ for all $k \geq 0$.

We begin with the inductive step. Let m be the number of exit claim clusters in $SPM_{=(k-1)}$. This is the number of boundary sources in "elevator" propagation across the

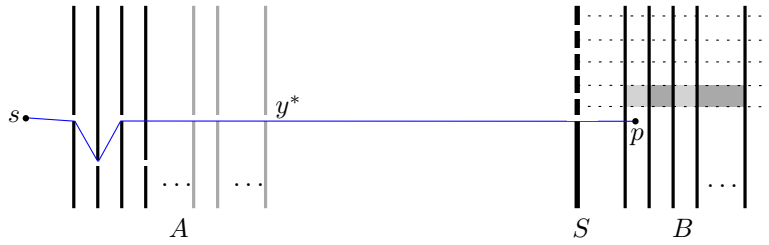


Figure 4 A shortest k-path map with complexity $\Omega(nk)$. Bundle A has $2k$ black strips and k gray strips; bundle B has k strips. The thick strip S has $\Omega(n)$ openings. A shortest k-path $\pi(p)$ from s is shown. Observe that $\pi(p)$ crosses $(k-1)$ strips in bundle A and therefore can cross only the first strip in bundle B.

obstacle interiors, going from level $k-1$ to level k. By Lemma 17, the resulting number of exit claim clusters is $m' = m + O(n)$. But m' is the number of boundary sources in the construction of $SPM_{=k}$, and once again by Lemma 17, the resulting number of exit claim clusters is $m'' = m' + O(n) = m + O(n)$, that is, $m'' \leq m + c_1 n$ for some constant c_1.

To establish the base case, recall that a shortest path map with no crossings (SPM_0) has complexity $O(n)$, which implies that the number of exit claims on its boundary is $O(n)$, i.e., at most $c_2 n$ for some constant c_2. Combining the base case and inductive step, we have shown that the number of exit claim clusters on the boundary of $SPM_{=k}$ is at most $c_2 n + k \cdot c_1 n$. The number of faces of $SPM_{=k}$ is at most equal to the number of boundary sources, which is at most $Cn(k+1)$, for $C = \max(c_1, c_2)$. Lemma 14 establishes the total complexity bound. ◀

A Matching Lower Bound. We will now bound the size of SPM_k from below by constructing a map with $\Omega(nk)$ regions. We construct an arrangement of obstacles as shown in Figure 4. We start with two obstacle bundles A and B placed parallel to the y-axis. Within each bundle, the horizontal spaces between strips are infinitesimal, but they are shown enlarged for clarity. The source s lies on the x-axis with bundle A placed right next to it. Bundle A consists of $3k$ perforated strips. In the first $2k$ strips, the odd numbered ones have openings at $y = 0$ and the even numbered ones have openings at $y = -0.5$. The next k strips have an opening at $y = 0$. Bundle B is placed at a distance D to the right of A and consists of k strips with no openings.

The last k strips in bundle A ensure that shortest k-paths starting at s must exit from the opening of the last strip in A (denoted by y^*); a path that crosses the last strip in A at some point other than y^* can be shortened while preserving the same number of crossings. Observe that a shortest path starting at s can reach y^* with i crossings, where $0 \leq i \leq k$. However, each crossing avoided results in an additional length of 1 unit. Therefore a shortest path with i crossings at y^* has an additional length of $(k-i)$ units. Also note that a shortest path with i crossings prior to y^* can cross the first $(k-i)$ of the k strips in bundle B, but cannot cross any farther. Therefore, to the right of strip j in bundle B, we get a region with k-predecessor $(y^*, k-j)$ and a total path length (to a point on the x-axis) of $D + j$. This gives us a total of k regions.

We extend this construction to $\Omega(nk)$ regions by adding a vertical strip S, which acts as a *path splitter*. This special strip has a total of m single-point openings at $y = 0, 1, \ldots, m$, denoted by y_i. We place S at an infinitesimal distance to the left of bundle B, creating k new regions for each opening of S. Note that in the range $0 \leq y \leq m$, a path that crosses S other than at one of the perforations y_i can be shortened by detouring through the nearest y_i and

inserting one more crossing before y^*. Hence a shortest k-path always passes through one of the y_i. This gives a total of $O(mk)$ regions: the k-predecessor of the region at $y = i$ and to the right of strip j of bundle B will be $(y_i, k - j)$, with a total path length of $\sqrt{D^2 + i^2} + j$.

The total number of vertices in our construction is $3k \times 4 + k \times 2 + (m + 1) \times 2 = 14k + 2m + 2$. By choosing $m = (n - 14k - 2)/2$ and assuming $k < n/28$, we have $m = \Theta(n)$ and the total number of regions in SPM_k is $\Omega(nk)$. This gives us the following lemma.

▶ **Lemma 19.** *The worst-case complexity of SPM_k is $\Omega(nk)$.*

Combining Lemmas 11, 18, and 19, we get the main result of this section.

▶ **Theorem 20.** *The shortest k-path map SPM_k has size $\Theta(kn)$.*

4 Computing SPM_k

In this section we describe an $O(k^2 n \log n)$ algorithm to construct SPM_k. Recall from our discussion about the k-garage (Definition 13), we can construct $SPM_{=k}$ iteratively, one level at a time. To compute the map at each level, we propagate the sources from the previous level and then perform wavefront propagation at the current level. For this, we use the algorithm for shortest paths in the presence of polygonal obstacles by Hershberger and Suri [15] as a subroutine. Except for a few small modifications required for our setting, most of the algorithm carries over unchanged. In the following, we briefly review the key ideas and discuss the necessary modifications.

The Hershberger-Suri algorithm uses the continuous Dijkstra method, which simulates the propagation of a unit speed wavefront in free space. The wavefront is a collection of circular wavelets. It changes its shape as it propagates and hits obstacles. Each wavelet originates at a *generator*, which may be a point source or an obstacle vertex (an *intermediate source*). A generator for a wavelet γ is identified by the pair (v, w), where v is an input vertex and w is the time at which v starts emitting γ. The Hershberger-Suri algorithm simulates wavefront propagation over a planar subdivision called the *conforming subdivision* of free space. For each subdivision edge e, and every point $p \in e$, the algorithm identifies the generator whose wavelet first reaches p. Combining these results for all $p \in e$ gives the *wavefront for* e. The key idea of the algorithm is to localize interesting events (such as wavelet collisions) within a constant number of cells in the subdivision. Each free-space edge e of this subdivision is contained in the union of a constant number of cells, called its *well-covering region* $\mathcal{U}(e)$. The wavefront for edge e is computed by combining and propagating the wavefront through $\mathcal{U}(e)$. The computed wavefronts are then merged to compute the shortest path map. This is the main result relevant to our algorithm:

▶ **Lemma 21** ([15]). *Given a set of polygonal obstacles with n vertices and a set of $O(n)$ sources with delays, one can compute the shortest path map in $O(n \log n)$ time and $O(n \log n)$ space.*

From the discussion preceding Lemma 17, recall that the sources on floor i are identified by triples (v, w, ℓ), where ℓ is a (sub-)edge of some obstacle H, (v, w) is a weighted point source on some floor $j < i$, and the wavelet γ generated by (v, w) enters floor i from the interior of H (an elevator) passing through edge ℓ. Each source (v, w, ℓ) defines a triangular flap glued onto the boundary of free space at ℓ. Conceptually, we think of the wavelet γ from (v, w, ℓ) as propagating in the flap before it enters floor i. Algorithmically, we can ignore the flap and start the propagation in free space at edge ℓ. This calls for a slight modification in the initialization step of the Hershberger-Suri algorithm. In particular, we do the following for each edge e of the conforming subdivision:

1. Find all boundary sources (v, w, ℓ) such that the well-covering region $\mathcal{U}(e)$ contains ℓ.
2. Initialize *covertime*(e), which is the time at which e would be engulfed by the wavefront, minimizing over all boundary sources (v, w, ℓ) with $\ell \in \mathcal{U}(e)$, and for each such source considering paths from v with delay w, constrained to pass through ℓ.
3. For each source (v, w, ℓ) with $\ell \in \mathcal{U}(e)$, propagate its wavelet γ to e inside $\mathcal{U}(e)$.

In the following lemma we show how to compute the boundary sources for each step of wavefront propagation.

▶ **Lemma 22.** *Given m boundary sources in a polygonal domain with n vertices, we can compute the exit claims of the sources in $O((m+n)\log(m+n))$ time and space.*

Proof. We apply the Hershberger-Suri algorithm, modified for boundary sources as described above. The algorithm computes the shortest path map for the sources inside the polygonal domain in total time and space $O((m+n)\log(m+n))$. The shortest path map partitions the boundary into $O(m+n)$ intervals, each claimed by its own source. The boundary sources form another set of m intervals. Overlaying these two sets of intervals in additional linear time and space, we identify the exit claims, i.e., those with a claiming source from a different segment. ◀

With these primitives in place, we are ready to describe our algorithm. The input is a polygonal domain P with convex obstacles. We will use M to denote the set of boundary sources passed as input to the Hershberger-Suri algorithm. The algorithm computes two things: the $(k-1)$-visibility region V and the $(=k)$-path map $SPM_{=k}$, which combined together form SPM_k. The length of the shortest path to any point p can then be easily computed by first locating the region containing p in the map SPM_k and then connecting p to the k-predecessor of this region as described in the beginning of Section 3.

Algorithm to construct SPM_k.

1. Set $M = \{s\}$ and call the Hershberger-Suri algorithm to compute SPM_0 for the polygonal domain P. Initialize V to be the empty region $\emptyset$.
2. Repeat for each $i \in 1, 2, \ldots, k$:
   a. Using Lemma 22, propagate the sources in SPM_{i-1} through the obstacles in P to compute the set of boundary sources M_{new} for $SPM_{=i}$.
   b. Identify all the regions in $SPM_{=(i-1)}$ for which the predecessor is s. Observe that this is precisely the region $V' = V_{i-1} \setminus V_{i-2}$. Set P to be the new polygonal domain with this region removed.
   c. If $V = \emptyset$, then set $V = V'$. Otherwise merge V with V' at the common vertices.
   d. Set $M = M_{new}$ and call the Hershberger-Suri algorithm to compute $SPM_{=i}$ for the polygonal domain P.
3. Merge $SPM_{=k}$ with V at the boundary of regions of $SPM_{=k}$ that have s as predecessor (i.e. $V' = V_k \setminus V_{k-1}$), to obtain SPM_k.

Observe that after Step 2c of iteration i, the region V is equal to V_{i-1}. Because V_{i-1} contains V_{i-2} and because both regions have linear size (by Lemma 11), Step 2c takes linear time. Therefore, the total running time is dominated by k calls to the Hershberger-Suri algorithm with $O(nk)$ sources (Theorem 20). We have the following result.

▶ **Theorem 23.** *If P is a polygonal domain bounded by convex obstacles with a total of n vertices, the shortest k-path map for P with respect to a source point s can be computed in $O(k^2 n \log n)$ time and $(kn \log n)$ space.*

5 Conclusion

In this paper, we studied the problem of finding shortest paths that are allowed to pass through a bounded number of convex obstacles. We showed that although two such k-paths may cross each other, they can be decomposed into non-crossing subpaths based on prefix-counts. This decomposition allows us to compute shortest k-paths efficiently, using the continuous Dijkstra framework. We showed that the size of the shortest k-path map is $\Theta(kn)$ and that it can be computed in worst-case time $O(k^2 n \log n)$ using $(kn \log n)$ space. Our algorithm's time complexity is optimal when $k = O(1)$.

References

1 M. Abellanas, A. García, F. Hurtado, J. Tejel, and J. Urrutia. Augmenting the connectivity of geometric graphs. *Computational Geometry*, 40(3):220–230, 2008.

2 R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice hall, 1993.

3 T. Asano. An efficient algorithm for finding the visibility polygon for a polygonal region with holes. *IEICE TRANSACTIONS (1976-1990)*, 68(9):557–559, 1985.

4 T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. *Algorithmica*, 1(1-4):49–63, 1986.

5 J.-L. De Carufel, C. Grimm, A. Maheshwari, and M. Smid. Minimizing the continuous diameter when augmenting paths and cycles with shortcuts. In *15th Scandinavian Symposium and Workshops on Algorithm Theory*, pages 27:1–27:14, 2016.

6 T. M. Chan. Low-dimensional linear programming with violations. *SIAM Journal on Computing*, 34(4):879–893, 2005.

7 D. Z. Chen and H. Wang. Computing shortest paths among curved obstacles in the plane. *ACM Trans. Algorithms*, 11(4):26:1–26:46, 2015.

8 H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

9 S. Eriksson-Bique, J. Hershberger, V. Polishchuk, B. Speckmann, S. Suri, T. Talvitie, K. Verbeek, and H. Yıldız. Geometric k shortest paths. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1616–1625, 2015.

10 M. Farshi, P. Giannopoulos, and J. Gudmundsson. Improving the stretch factor of a geometric network by edge augmentation. *SIAM Journal on Computing*, 38(1):226–240, 2008.

11 S. K. Ghosh and D. M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM Journal on Computing*, 20(5):888–910, 1991.

12 L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1-4):209–233, 1987.

13 S. Har-Peled and V. Koltun. Separability with outliers. *16th International Symposium on Algorithms and Computation*, pages 28–39, 2005.

14 J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *Computational Geometry*, 4(2):63–97, 1994.

15 J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM Journal on Computing*, 28(6):2215–2256, 1999.

16 J. Hershberger, S. Suri, and H. Yıldız. A near-optimal algorithm for shortest paths among curved obstacles in the plane. In *Proceedings of the Twenty-Ninth Annual Symposium on Computational Geometry*, pages 359–368, 2013.

17 S. Kapoor and S. N. Maheshwari. Efficient algorithms for Euclidean shortest path and visibility problems with polygonal obstacles. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, pages 172–182, 1988.

18 D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.

19 D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.

20 A. Maheshwari, S. C. Nandy, D. Pattanayak, S. Roy, and M. Smid. Geometric path problems with violations. *Algorithmica*, pages 1–24, 2016.

21 J. Matoušek. On geometric optimization with few violated constraints. *Discrete & Computational Geometry*, 14(4):365–384, 1995.

22 J. S. B. Mitchell. A new algorithm for shortest paths among obstacles in the plane. *Annals of Mathematics and Artificial Intelligence*, 3(1):83–105, 1991.

23 J. S. B. Mitchell. Shortest paths among obstacles in the plane. *International Journal of Computational Geometry & Applications*, 6(3):309–332, 1996.

24 J. S. B. Mitchell and C. H. Papadimitriou. The weighted region problem: finding shortest paths through a weighted planar subdivision. *Journal of the ACM (JACM)*, 38(1):18–73, 1991.

25 M. H. Overmars and E. Welzl. New methods for computing visibility graphs. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, pages 164–171, 1988.

26 H. Rohnert. Shortest paths in the plane with convex polygonal obstacles. *Information Processing Letters*, 23(2):71–76, 1986.

27 T. Roos and P. Widmayer. k-violation linear programming. *Information Processing Letters*, 52(2):109–114, 1994.

28 J. A. Storer and J. H. Reif. Shortest paths in the plane with polygonal obstacles. *Journal of the ACM (JACM)*, 41(5):982–1012, 1994.

Contracting a Planar Graph Efficiently*

Jacob Holm[†1], Giuseppe F. Italiano[‡2], Adam Karczmarz[§3], Jakub Łącki[¶4], Eva Rotenberg[5], and Piotr Sankowski[§‖6]

1 **University of Copenhagen, Denmark**
jaho@di.ku.dk

2 **University of Rome Tor Vergata**
giuseppe.italiano@uniroma2.it

3 **University of Warsaw, Poland**
a.karczmarz@mimuw.edu.pl

4 **Google Research, New York**
jlacki@google.com

5 **University of Copenhagen, Denmark**
roden@di.ku.dk

6 **University of Warsaw, Poland**
sank@mimuw.edu.pl

Abstract

We present a data structure that can maintain a simple planar graph under edge contractions in linear total time. The data structure supports adjacency queries and provides access to neighbor lists in $O(1)$ time. Moreover, it can report all the arising self-loops and parallel edges.

By applying the data structure, we can achieve optimal running times for decremental bridge detection, 2-edge connectivity, maximal 3-edge connected components, and the problem of finding a unique perfect matching for a static planar graph. Furthermore, we improve the running times of algorithms for several planar graph problems, including decremental 2-vertex and 3-edge connectivity, and we show that using our data structure in a black-box manner, one obtains conceptually simple optimal algorithms for computing MST and 5-coloring in planar graphs.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases Planar graphs, algorithms, data structures, connectivity, coloring.

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.50

* A full version of the paper is available at https://arxiv.org/abs/1706.10228.

† This research is supported by the Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research under the Sapere Aude research career programme.

‡ Partly supported by the Italian Ministry of Education, University and Research under Project AMANDA (Algorithmics for MAssive and Networked DAta).

§ Supported by the Polish National Science Center grant number 2014/13/B/ST6/01811.

¶ When working on this paper Jakub Łącki was partly supported by the EU FET project MULTIPLEX no. 317532 and the Google Focused Award on "Algorithms for Large-scale Data Analysis" and Polish National Science Center grant number 2014/13/B/ST6/01811. Part of this work was done while Jakub Łącki was visiting the Simons Institute for the Theory of Computing.

‖ The work of P. Sankowski is a part of the project TOTAL that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 677651).

1 Introduction

An edge contraction is one of the fundamental graph operations. Given an undirected graph and an edge e, contracting the edge e consists in removing it from the graph and merging its endpoints. The notion of a contraction has been used to describe a number of prominent graph algorithms, including Edmonds' algorithm for computing maximum matchings [4] or Karger's minimum cut algorithm [11].

Edge contractions are of particular interest in planar graphs, as a number of planar graph properties are easiest described using contractions. For example, it is well-known that a graph is planar precisely when it cannot be transformed into K_5 or $K_{3,3}$ by contracting edges or removing vertices or edges. Moreover, contracting an edge preserves planarity.

While a contraction operation is conceptually very simple, its efficient implementation is challenging. By using standard data structures (e.g. balanced binary trees), one can maintain adjacency lists of a graph in polylogarithmic amortized time. However, in many planar graph algorithms this becomes a bottleneck. As an example, consider the problem of computing a 5-coloring of a planar graph. There exists a very simple algorithm based on contractions [17], but efficient implementations use some more involved planar graph properties [5, 17, 18]. For example, the algorithm by Matula, Shiloach and Tarjan [17] uses the fact that every planar graph has either a vertex of degree at most 4 or a vertex of degree 5 adjacent to at least four vertices each having degree at most 11. Similarly, although there exists a very simple algorithm for computing a MST of a planar graph based on edge contractions, various different methods have been used to implement it efficiently [5, 15, 16].

Our Results. We show a data structure that can efficiently maintain a planar graph subject to edge contractions in $O(n)$ total time, assuming the standard word-RAM model with word size $\Omega(\log n)$. It can report groups of parallel edges and self-loops that emerge. It also supports constant-time adjacency queries and maintains the neighbor lists and degrees explicitly. The data structure can be used as a black-box to implement planar graph algorithms that use contractions. In particular, it can be used to give clean and conceptually simple implementations of the algorithms for computing 5-coloring or MST that do not manipulate the embedding. More importantly, by using our data structure we give improved algorithms for a few problems in planar graphs. In particular, we obtain optimal algorithms for decremental 2-edge-connectivity, finding unique perfect matching, and computing maximal 3-edge-connected subgraphs. We also obtain improved algorithms for decremental 2-vertex and 3-edge connectivity, where the bottleneck in the state-of-the-art algorithms [7] is detecting parallel edges under contractions. For detailed theorem statements, see Sections 3 and 4.

Related work. The problem of detecting self-loops and parallel edges under contractions is implicitly addressed by Giammarresi and Italiano [7] in their work on decremental (edge-, vertex-) connectivity in planar graphs. Their data structure uses $O(n \log^2 n)$ total time.

In their book, Klein and Mozes [12] show that there exists a data structure maintaining a planar graph under edge contractions and deletions and answering adjacency queries in $O(1)$ worst-case time. The update time is $O(\log n)$. This result is based on the work of Brodal and Fagerberg [1], who showed how to maintain a bounded outdegree orientation of a dynamic planar graph so that edge insertions and deletions are supported in $O(\log n)$ amortized time.

Gustedt [9] showed an optimal solution to the union-find problem, in the case when at any time, the actual subsets form disjoint, connected subgraphs of a given planar graph G. In other words, in this problem the allowed unions correspond to the edges of a planar graph and the execution of a union operation can be seen as a contraction of the respective edge.

Our Techniques. It is relatively easy to give a simple *vertex merging data structure* for general graphs, that would process any sequence of contractions in $O(m \log^2 n)$ total time and support the same queries as our data structure in $O(\log n)$ time. To this end, one can store the lists $N(v)$ of neighbors of individual vertices as balanced binary trees. Upon a contraction of an edge uv, or a more general operation of merging two (not necessarily adjacent) vertices u, v, $N(u)$ and $N(v)$ are merged by inserting the smaller set into the larger one (and detecting loops and parallel edges by the way, at no additional cost). If we used hash tables instead of balanced BSTs, we could achieve $O(\log n)$ expected amortized update time and $O(1)$ query time. In fact, such an approach was used in [7].

To obtain the speed-up we take advantage of planarity. Our general idea is to partition the graph into small pieces and use the above simple-minded vertex merging data structures to solve our problem separately for each of the pieces and for the subgraph induced by the vertices contained in multiple pieces (the so-called boundary vertices). Due to the nature of edge contractions, we need to specify how the partition evolves when our graph changes.

The data structure builds an r-division (see Section 2) $\mathcal{R} = P_1, P_2, \ldots$ of G_0 for $r = \log^4 n$. The set $\partial\mathcal{R}$ of boundary vertices (i.e., those shared among at least two pieces) has size $O(n/\log^2 n)$. Let (V_0, E_0) denote the original graph, and (V, E) denote the current graph (after performing some number of contractions). Then we can denote by $\phi : V_0 \to V$ a function such that the initial vertex $v_0 \in V_0$ is contracted into $\phi(v_0)$. We use vertex merging data structures to detect parallel edges and self-loops in the "top-level" subgraph $G[\phi(\partial\mathcal{R})]$, which contains only edges between boundary vertices, and separately for the "bottom-level" subgraphs $G[\phi(V(P_i))] \setminus G[\phi(\mathcal{R})]$. At any time, each edge of G is contained in exactly one of the defined subgraphs, and thus, the distribution of responsibility for handling individual edges is based solely on the initial r-division.

However, such an assignment of responsibilities gives rise to additional difficulties. First, a contraction of an edge in a lower-level subgraph might cause some edges "flow" from this subgraph to the top-level subgraph (i.e., we may get new edges connecting boundary vertices). As such an operation turns out to be costly in our implementation, we need to prove that the number of such events is only $O(n/\log^2 n)$.

Another difficulty lies in the need of keeping the individual data structures synchronized: when an edge of the top-level subgraph is contracted, pairs of vertices in multiple lower-level subgraphs might need to be merged. We cannot afford iterating through all the lower-level subgraphs after each contraction in $G[\phi(\partial\mathcal{R})]$. This problem is solved by maintaining a system of pointers between representations of the same vertex of V in different data structures and another clever application of the smaller-to-larger merge strategy.

Such a two-level data structure would yield a data structure with $O(n \log\log n)$ total update time. To obtain a linear time data structure, we further partition the pieces P_i and add another layer of maintained subgraphs on $O(\log^4 \log^4 n) = O(\log^4 \log n)$ vertices. These subgraphs are so small that we can precompute in $O(n)$ time the self-loops and parallel edges for every possible graph on t vertices and every possible sequence of edge contractions.

We note that this overall idea of recursively reducing a problem with an r-division to a size when microencoding can be used has been previously exploited in [9] and [14] (Gustedt [9] did not use r-divisions, but his concept of a *patching* could be replaced with an r-division). Our data structure can be also seen as a solution to a more general version of the planar union-find problem studied by Gustedt [9]. However, maintaining the status of each edge e of the initial graph G (i.e., whether e has become a self-loop or a parallel edge) subject to edge contractions turns out to be a serious technical challenge. For example, in [9], the requirements posed on the bottom-level union-find data structures are in a sense relaxed and it is not necessary for those to be synchronized with the top-level union-find data structure.

Organization of the Paper. The remaining part of this paper is organized as follows. In Section 2, we introduce the needed notation and definitions, whereas in Section 3 we define the operations that our data structure supports. Then, in Section 4 we present a series of applications of our data structure. In Section 5, we provide a detailed implementation of our data structure. Due to space constraints, many of the proofs, along with the pseudocode for example algorithms using our data structure, can be found in the full version of this paper [10].

2 Preliminaries

Throughout the paper we use the term *graph* to denote an undirected *multigraph*, that is we allow the graphs to have parallel edges and self-loops. Formally, each edge e of such a graph is a pair $(\{u, w\}, \mathrm{id}(e))$ consisting of a pair of vertices and a unique identifier used to distinguish between the parallel edges. For simplicity, we skip this third coordinate and use just uw to denote one of the edges connecting vertices u and w. If the graph contains no parallel edges and no self-loops, we call it *simple*.

For any graph G, we denote by $V(G)$ and $E(G)$ the sets of vertices and edges of G, respectively. A graph G' is called a subgraph of G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. We define $G_1 \cup G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2))$ and $G_1 \setminus G_2 = (V(G_1), E(G_1) \setminus E(G_2))$. For $S \subseteq V(G)$, we denote by $G[S]$ the *induced subgraph* $(S, \{uv : uv \in E(G), \{u, v\} \subseteq S\})$.

For a vertex $v \in V$, we define $N(v) = \{u : uv \in E, u \neq v\}$ to be the *neighbor set* of v.

A *cycle* of a graph G is a nonempty set $C \subseteq E(G)$, such that for some ordering of edges $C = \{u_1w_1, \ldots, u_kw_k\}$, we have $w_i = u_{i+1}$ for $1 \leq i < k$ and $w_k = u_1$, and the vertices $u_1, \ldots, u_k$ are distinct. The *length* of a cycle C is simply $|C|$. Note that this definition allows cycles of length 1 (self-loop) or 2 (a pair of parallel edges), but does not allow non-simple cycles of length 3 or more. A *cut* is a minimal (w.r.t. inclusion) set $C \subseteq E(G)$, such that $G \setminus C$ has more connected components than G.

Let $G = (V, E)$ be a graph and $xy = e \in E$. We use $G - e$ to denote the graph obtained from G by removing e and G/e to denote the graph obtained by contracting an edge e (in the case of a contraction e may not be a self-loop, i.e., $x \neq y$). We will often look at contraction from the following perspective: as a result of contracting e, all edge endpoints equal to x or y are replaced with some new vertex z. In some cases it is convenient to assume $z \in \{x, y\}$. This yields a 1-to-1 correspondence between the edges of $G - e$ and the edges of G/e. Formally, we assume that the contraction preserves the edge identifiers, i.e., $e_1 \in E(G - e)$ and $e_2 \in E(G/e)$ are corresponding if and only if $\mathrm{id}(e_1) = \mathrm{id}(e_2)$.

Note that contracting an edge may introduce parallel edges and self-loops. Namely, for each edge that is parallel to e in G, there is a self-loop in G/e. And for each cycle of length 3 that contains e in G, there is a pair of parallel edges in G/e.

Planar graphs. An embedding of a planar graph is a mapping of its vertices to distinct points and edges to non-crossing curves in the plane. We say that a planar graph G is *plane*, if some embedding of G is assumed. A face of a connected plane G is a maximal open connected set of points not in the image of any vertex or edge in the embedding of G.

Duality. Let G be a plane graph. We denote by G^* the dual graph of G. Each edge of G naturally corresponds to an edge of G^*. We denote by e^* the edge of G^* that corresponds to $e \in E(G)$. More generally, if $E_1 \subseteq E(G)$ is a set of edges of G, we set $E_1^* = \{e^* | e \in E_1\}$.

We exploit the following relations between G and G^*. Deleting an edge e of G corresponds to contracting the edge e^* in G^*, that is $(G - e)^* = G^*/e^*$. Moreover, $C \subseteq E$ is a cut in G iff C^* is a cycle in G^*. In particular, a bridge e in G corresponds to a self-loop in G^* and a two-edge cut in G corresponds to a pair of parallel edges in G^*.

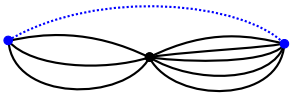


Figure 1 Contracting the blue dotted edge will merge two groups of parallel edges.

Planar graph partitions. Let G be a simple planar graph. Let a *piece* be subgraph of G with no isolated vertices. For a piece P, we denote by ∂P the set of vertices $v \in V(P)$ such that v is adjacent to some edge of G that is not contained in P. ∂P is also called the set of *boundary vertices* of P. An r-division $\mathcal{R}$ of G is a partition of G into $O(n/r)$ edge-disjoint pieces such that each piece $P \in \mathcal{R}$ has $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices. For an r-division $\mathcal{R}$, we also denote by $\partial\mathcal{R}$ the set $\bigcup_{P_i \in \mathcal{R}} \partial P_i$. Clearly, $|\partial\mathcal{R}| = O(n/\sqrt{r})$.

▶ **Lemma 1** ([8, 13, 19]). *An r-division of a planar graph G can be computed in linear time.*

3 The Data Structure Interface

In this section we specify the set of operations that our data structure supports so that it fits our applications. It proves beneficial to look at the graph undergoing contractions from two perspectives.

1. The *adjacency viewpoint* allows us to track the neighbor sets of the individual vertices, as if G was simple at all times.
2. The *edge status viewpoint* allows us to track, for all the original edges E_0, whether they became self-loops or parallel edges, and also track how E_0 is partitioned into classes of pairwise-parallel edges.

Let $G_0 = (V_0, E_0)$ be a planar graph used to initialize the data structure. Recall that any contraction alters both the set of vertices and the set of edges of the graph. Throughout, we let $G = (V, E)$ denote the *current* version of the graph, unless otherwise stated.

Each edge $e \in E(G)$ can be either a self-loop, an edge parallel to some other edge $e' \neq e$ (we call such an edge *parallel*), or an edge that is not parallel to any other edge of G (we call it *simple* in this case). An edge $e \in E(G)$ that is simple might either get contracted or might change into a parallel edge as a result of contracting other edges. Similarly, a parallel edge might either get contracted or might change into a self-loop. Note that, during contractions, neither can a parallel edge ever become simple, nor can a self-loop become parallel.

Observe that parallelism is an equivalence relation on the edges of G. Once two edges e_1, e_2 connecting vertices $u, v \in V$ become parallel, they stay parallel until some edge e_3 (possibly equal to e_1 or e_2) parallel to both of them gets contracted. However, groups of parallel edges might merge (Figure 1) and this might also be a valuable piece of information.

To succinctly describe how the groups of parallel edges change, we report parallelism in a directed manner, as follows. Each group $Y \subseteq E$ of parallel edges in G is assumed to have its *representative* edge $\alpha(Y)$. For $e \in Y$ we define $\alpha(e) = \alpha(Y)$. When two groups of parallel edges $Y_1, Y_2 \subseteq E$ merge as a result of a contraction, the data structure chooses $\alpha(Y_i)$ for some $i \in \{1, 2\}$ to be the new representative of the group $Y_1 \cup Y_2$ and reports an ordered pair $\alpha(Y_{3-i}) \to \alpha(Y_i)$ to the user. We call each such pair a *directed parallelism*. After such an event, $\alpha(Y_{3-i})$ will not be reported as a part of a directed parallelism anymore. The choice of i can also be made according to some fixed strategy, e.g., if the edges are assigned weights $\ell(\cdot)$ then we may choose $\alpha(Y_i)$ so that $\ell(\alpha(Y_i)) \leq \ell(\alpha(Y_{3-i}))$. This is convenient in what Klein and Mozes [12] call *strict optimization problems*, such as MST, where we can discard one of any two parallel edges based only on these edges.

Note that at any point of time the set of directed parallelisms reported so far can be seen as a forest of rooted trees $\mathcal{T}$, such that each tree T of $\mathcal{T}$ represents a group Y of parallel edges of G. The root of T is equal to $\alpha(Y)$.

When some edge is contracted, all edges parallel to it are reported as self-loops. Clearly, each edge e is reported as a self-loop at most once. Moreover, it is reported as a part of a directed parallelism $e \to e'$, $e' \neq e$, at most once.

We are now ready to define the complete interface of our data structure.

- `init`$(G_0 = (V_0, E_0), \ell)$: initialize the data structure. ℓ is an optional weight function.
- $(s, P, L) :=$ `contract`(e), for $e \in E$: contract the edge e. Let $e = uv$. The call `contract`(e) returns a vertex s resulting from merging u and v, and two lists P, L of new directed parallelisms and self-loops, respectively, reported as a result of contraction of e.
- `vertices`(e), for $e \in E$: return $u, v \in V$ such that $e = uv$.
- `neighbors`(u), for $u \in V$: return an iterator to the list $\{(v, \alpha(uv)) : v \in N(u)\}$.
- `deg`(u), for $u \in V$: find the number of neighbors of u in G.
- `edge`(u, v), for $u, v \in V$: if $uv \in E$, then return $\alpha(uv)$. Otherwise, return **nil**.

The following theorem summarizes the performance of our data structure.

▶ **Theorem 2.** *Let $G = (V, E)$ be a planar graph with $|V| = n$ and $|E| = m$. There exists a data structure supporting* `edge`*,* `vertices`*,* `neighbors` *and* `deg` *in $O(1)$ worst-case time, and whose initialization and any sequence of* `contract` *operations take $O(n + m)$ expected time, or $O(n + m)$ worst-case time, if no* `edge` *operations are performed. The data structure supports iterating through the neighbor list of a vertex with $O(1)$ overhead per element.*

4 Applications

Decremental Edge- and Vertex-Connectivity. In the *decremental k-edge (k-vertex) connectivity* problem, the goal is to design a data structure that supports queries about the existence of k edge-disjoint (vertex-disjoint) paths between a pair of given vertices, subject to edge deletions. We obtain improved algorithms for decremental 2-edge-, 2-vertex- and 3-edge-connectivity in dynamic planar graphs. For decremental 2-edge-connectivity we obtain an optimal data structure with both updates and queries supported in amortized $O(1)$ time. In the case of 2-vertex- and 3-edge-connectivity, we achieve the amortized update time of $O(\log n)$, whereas the query time is constant. For all these problems, we improve upon the 20-year-old update bounds by Giammarresi and Italiano [7] by a factor of $O(\log n)$.

▶ **Theorem 3.** *Let $G = (V, E)$ be a planar graph and let $n = |V|$. There exists a deterministic data structure that maintains G subject to edge deletions and can answer 2-edge connectivity queries in $O(1)$ time. Its total update time is $O(n)$.*

Proof. Denote by G_0 the initial graph. Suppose wlog. that G_0 is connected. Let $B(G)$ be the set of all bridges of G. Note that two vertices u, v are in the same 2-edge-connected component of G iff they are in the same connected component of the graph $(V, E \setminus B(G))$.

Observe that if e is a bridge, then deleting e from G does not influence the 2-edge-components of G. Hence, when a bridge e is deleted, we may ignore this deletion. We denote by G' be the graph obtained from G_0 by the same sequence of deletions as G, but ignoring the bridge deletions. This way, G' is connected at all times and the 2-edge-connected components of G' and G are the same. It is also easy to see that $E(G) \setminus B(G) = E(G') \setminus B(G')$ and $B(G) = B(G') \cap E(G)$. Moreover, the set $E(G')$ shrinks in time whereas $B(G')$ only grows.

First we show how the set $B(G')$ is maintained. Recall that $e \in E(G')$ is a bridge of G' iff e^* is a self-loop of G'^*. We build the data structure of Theorem 2 for G'^*, which initially equals G_0^*. As deleting a non-bridge edge e of G' translates to a contraction of a non-loop edge e^* in G'^*, we can maintain $B(G')$ in $O(n)$ total time by detecting self-loops in G'^*.

Denote by H the graph $(V, E(G') \setminus B(G'))$. To support 2-edge connectivity queries, we maintain the graph H with the decremental connectivity data structure of Łącki and Sankowski [14]. This data structure maintains a planar graph subject to edge deletions in linear total time and supports connectivity queries in $O(1)$ time. When an edge e is deleted from G, we first check whether it is a bridge and if so, we do nothing. If e is not a bridge, the set $E(G')$ shrinks and thus we remove the edge e from H. The deletion of e might cause the set $B(G')$ to grow. Any new edge of $B(G')$ is also removed from H afterwards.

To conclude, note that each 2-edge connectivity query on G translates to a single connectivity query in H. All the maintained data structures have $O(n)$ total update time. ◀

As an almost immediate consequence of Theorem 3 we improve upon [6] and obtain an optimal algorithm for the *unique perfect matching* problem when restricted to planar graphs.

▶ **Corollary 4.** *Given a planar graph $G = (V, E)$ with $n = |V|$, in $O(n)$ time we can find a unique perfect matching of G or detect that the number of perfect matchings in G is not 1.*

To obtain improved bounds for 2-vertex connectivity and 3-edge connectivity we use the data structure of Theorem 2 to remove bottlenecks in the existing algorithms by Giammarresi and Italiano [7].

▶ **Theorem 5.** *Let $G = (V, E)$ be a planar graph and let $n = |V|$. There exists a deterministic data structure that maintains G subject to edge deletions and can answer 2-vertex connectivity and 3-edge connectivity queries in $O(1)$ time. Its total update time is $O(n \log n)$.*

Maximal 3-Edge-Connected Subgraphs. A k-edge-connected component of a graph G is a maximal (w.r.t. inclusion) subset S of vertices, such that each pair of vertices in S is k-edge-connected. However, if $k \geq 3$, in the subgraph of G induced by S, some pairs of vertices may not be k-edge-connected (see [2] for an example). Thus, for $k \geq 3$, maximal k-edge-connected subgraphs can be different from k-edge-connected components. Very recently, Chechik et al. [2] showed how to compute maximal k-edge-connected subgraphs in $O((m + n \log n)\sqrt{n})$ time for any constant k, or $O(m\sqrt{n})$ time for $k = 3$. Using the results of [7] one can compute maximal 3-edge-connected subgraphs of a planar multigraph in $O(m + n \log n)$ time. Our new approach allows us to improve this to an optimal $O(m + n)$ time bound.

▶ **Lemma 6.** *The maximal 3-edge-connected subgraphs of a planar graph can be computed in linear time.*

Simple Linear-Time Algorithms. Finally, we present two examples showing that Theorem 2 might be a useful black-box in designing linear time algorithms for planar graphs. The details and the relevant pseudocode can be found in the full version of this paper [10].

▶ **Example 7.** Every planar graph G can be 5-colored in expected linear time.

Proof. A textbook proof of the 5-color theorem proceeds by induction as follows (see Figure 2). Each simple planar graph has a vertex u of degree at most 5. The case when u has degree less than 5 is easy: for any $v \in N(u)$, we can color G/uv inductively, uncontract the edge uv and finally recolor u with a color not used among the vertices $N(u)$. When, however, u has

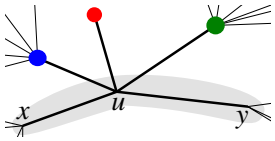


Figure 2 The degree ≤ 5 vertex and its two independent neighbors may be colored using the remaining two colors.

degree exactly 5, there exist two neighbors of x, y of u such that x and y are not adjacent, as otherwise G would contain K_5. We could thus obtain a planar graph G' by contracting both ux and uy. After inductively coloring G' and "uncontracting" ux and uy, we obtain a coloring of G that is valid, except that x, y and u have the same colors assigned. Thus, at most 4 colors are used among the neighbors of u and we recolor u to the remaining color in order to get a valid coloring of G.

Note that this proof can be almost literally converted into a linear time 5-coloring algorithm (see the full version [10] for the pseudocode) using the data structure of Theorem 2 built for G. We only need to maintain a subset Q of vertices of G with degree at most 5. The subset Q can be easily maintained in linear total time, since all vertices that potentially change their degrees after the call `contract`(e) are endpoints of the reported parallel edges. ◀

▶ **Example 8.** An MST of a planar graph G can be computed in linear time.

5 Maintaining a Planar Graph Under Contractions

In this section we prove Theorem 2. We defer the discussion on supporting arbitrary weights $\ell(\cdot)$ to the full version [10]. Hence, in the following, we assume all edges have equal weights.

5.1 A Vertex Merging Data Structure

We first consider a more general problem, which we call the *bordered vertex merging* problem. The data structure presented below will constitute a basic building block of the multi-level data structure. Let us now describe the data structure for the bordered vertex merging problem in detail. Suppose we have a dynamic *simple* planar graph $G = (V, E)$ and a *border set* $B \subseteq V$. Assume G is initially equal to $G_0 = (V_0, E_0)$ and no edge of E_0 connects two vertices of B. The data structure handles the following update operations.

- Merge (or in other words, an identification) of two vertices $u, v \in V$ $(u \neq v)$, such that the graph is still planar. If $\{u, v\} \not\subseteq B$, then u and v have to be connected by an edge and in such a case the merge is equivalent to a contraction of uv.
- Insertion of an edge $e = uv$ (where $uv \notin E$ is not required), preserving planarity.

After each update operation the data structure reports the parallel edges and self-loops that emerge. Once reported, each set of parallel edges is merged into one representative edge. Moreover, the data structure reports and removes any edges that have both endpoints in B. Thus, the following invariants are satisfied before the first and after each modification:

1. G is planar and simple.
2. No edge of E has both its endpoints in B.

Clearly, merging vertices alters the set V by replacing two vertices u, v with a single vertex. Thus, at each step, each vertex of G corresponds to a set of vertices of the initial graph G_0. We explicitly maintain a mapping $\phi : V_0 \to V$ such that for $a \in V_0$, $\phi(a)$ is a

vertex of the current vertex set V "containing" a. The reverse mapping $\phi^{-1} : V \to 2^{V_0}$ is also stored explicitly. We now define how the merge of u and v influences the set B. When $\{u, v\} \subseteq B$, the resulting vertex is also in B. When $u \in B, v \notin B$ (or $v \in B, u \notin B$, resp.), the resulting vertex is included in B in place of u (v, resp.). Finally, for $u, v \notin B$, the resulting vertex does not belong to B either.

Let $\tilde{E}$ be the set of inserted edges. At any time, the edges of E constitute a subset of $E_0 \cup \tilde{E}$ in the following sense: for each $e = xy \in E$ there exists an edge $e' = uv \in E_0 \cup \tilde{E}$ such that $\mathrm{id}(e) = \mathrm{id}(e')$, and vertices u and v have been merged into x and y, respectively.

Note that some modifications might break the second invariant: both an edge insertion and a merge might introduce an edge e with both endpoints in B. We call such an edge a *border edge*. Each border edge e that is not a self-loop is reported and deleted from (or not inserted to) G. Apart from reporting and removing new edges of $B \times B$ appearing in E, we also report the newly created parallel edges that might arise after the modification and remove them. The reporting of parallel edges is done in the form of directed parallelisms, as described in Section 3. Again, it is easy to see that each edge of $E_0 \cup \tilde{E}$ is reported as the first coordinate of a directed parallelism at most once.

Note that an edge e may be first reported parallel (in a directed parallelism of the form $e' \to e$, where $e' \neq e$) and then reported border.

The Graph Representation. The data structure for the bordered vertex merging problem internally maintains G using the data structure of the following lemma for planar graphs.

▶ **Lemma 9** ([1]). *There exists a deterministic, linear-space data structure, initialized in $O(n)$ time, and maintaining a dynamic, simple planar graph H with n vertices, so that:*

- *adjacency queries in H can be performed in $O(1)$ worst-case time,*
- *edge insertions and deletions can be performed in $O(\log n)$ amortized time.*

▶ **Fact 10.** *The data structure of Lemma 9 can be easily extended so that:*

- *Doubly-linked lists $N(v)$ of neighbors, for $v \in V$, are maintained within the same bounds.*
- *For each edge xy of H, some auxiliary data associated with e can be accessed and updated in $O(1)$ worst-case time.*

In addition to the data structure of Lemma 9 representing G, for each unordered pair x, y of vertices adjacent in G, we maintain an edge $\alpha(x, y) = e$, where e is the unique edge in E connecting x and y. Recall that in fact $\alpha(x, y)$ corresponds to some of the original edges of E_0 or one of the inserted edges $\tilde{E}$. By Fact 10, we can access $\alpha(x, y)$ in constant time.

The mapping ϕ is stored in an array, whereas the sets $\phi^{-1}(\cdot)$ – in doubly-linked lists.

Suppose we merge two vertices $u, v \in V$. Instead of creating a new vertex w, we merge one of these vertices into the other. Suppose we merge u into v. In terms of the operations supported by the data structure of Lemma 9, we need to remove each edge ux and insert an edge vx, unless v has been adjacent to x before.

To update our representation, we only need to perform the following steps:

- For each $v_0 \in \phi^{-1}(u)$, set $\phi(v_0) = v$ and add v_0 to $\phi^{-1}(v)$.
- Compute the list $N_u = \{(x, \alpha(u, x)) : x \in N(u)\}$. Remove all edges adjacent to u from G. For each $(x, \alpha(u, x)) \in N_u$, $x \neq v$, check whether $x \in N(v)$ (this can be done in $O(1)$ time, by Lemma 9). If so, report the parallelism $\alpha(u, x) \to \alpha(v, x)$. Otherwise, if vx is not a border edge, insert an edge vx to G and set $\alpha(v, x) = \alpha(u, x)$. If, on the other hand, $v \in B$ and $x \in B$ (i.e., vx is a border edge), report $\alpha(u, x)$ as a border edge.

Observe that our order of updates issued to G guarantees that G remains planar at all times.

The decision whether we merge u into v or v into u heavily affects both the correctness and efficiency of the data structure. First, if one of u, v (say v) is contained in B, whereas the other (say u) is not, we merge u into v. If, however, we have $\{u, v\} \subseteq B$ or $\{u, v\} \subseteq V \setminus B$, we pick a vertex (say u) with a smaller set $\phi^{-1}(u)$ and merge u into v.

To handle an insertion of a new edge $e = xy$, we first check whether xy is a border edge. If so, we discard e and report it. Otherwise, check whether x and y are adjacent in G. If so, report the parallelism $e \to \alpha(x, y)$. If not, add an edge xy to G and set $\alpha(x, y) = e$.

▶ **Lemma 11.** *Let G be a graph initially equal to a simple planar graph $G_0 = (V_0, E_0)$ such that $n = |V_0|$. There is a data structure for the bordered vertex merging problem that processes any sequence of modifications of G_0, along with reporting parallelisms and border edges, in $O((n + f) \log^2 n + m)$ total time, where m is the total number of edge insertions and f is the total number of insertions of edges connecting non-adjacent vertices.*

Proof. Clearly, by Lemma 9, building the initial representation takes $O(n \log n)$ time, as we insert $O(n)$ edges to G. The reporting of parallel edges and border edges takes $O(n + m)$ time, since each (initial or inserted) edge is reported as a border edge or occurs as the first coordinate of a reported directed parallelism at most once.

Also note that, by Lemma 9, an insertion of a parallel edge costs $O(1)$ time, for a total of $O(m)$ time over all insertions, as G is not updated in that case. Recall that, by Fact 10, accessing and updating values $\alpha(x, y)$ for $xy \in E(G)$ takes $O(1)$ time.

The total cost of maintaining the representation of G is $O(g \log n)$, where g is the total number of edge updates to the data structure of Lemma 9. We prove that $g = O((n+f) \log n)$. To this end, we look at the merge of u into v from a different perspective: instead of removing an edge $e = ux$ and inserting an edge vx, imagine that we simply change an endpoint u of e to v, but the edge itself does not lose its identity. Then, new edges in G are only created either during the initialization or by inserting an edge connecting the vertices that have not been previously adjacent in G. Hence, there are $O(n + f)$ creations of new edges.

Consider some edge $e = xy$ of G immediately after its creation. Denote by $q(e)$ the pair $(|\phi^{-1}(x)|, |\phi^{-1}(y)|)$. The value of $q(e)$ always changes when some endpoint of e is updated. Suppose a merge of u into v ($u \neq v$) causes the change of some endpoint u of e to v. We either we have $u \notin B$ and $v \in B$ or $|\phi^{-1}(v)| \geq |\phi^{-1}(u)|$ before the merge. The former situation can arise at most once per each endpoint of e, since we always merge a non-border vertex into a border vertex, if such case arises. In the latter case, on the other hand, one coordinate of $q(e)$ grows at least by a factor of 2, and clearly this can happen at most $O(\log n)$ times, as the size of any $\phi^{-1}(x)$ is never more than n. Since there are $O(n + f)$ "created" edges, and each such edge undergoes $O(\log n)$ endpoint updates, indeed we have $g = O((n + f) \log n)$.

A very similar argument can be used to show that the total time needed to maintain the mapping ϕ along with the reverse mapping ϕ^{-1} is $O(n \log n)$. ◀

A Micro Data Structure. In order to obtain an optimal data structure, we need the following specialized version of the bordered vertex merging data structure that handles very small graphs in linear total time. Suppose we disallow inserting new edges into G. Additionally, assume we are allowed to perform some preprocessing in time $O(n)$. Then, due to a monotonous nature of allowed operations on G, when the size of G_0 is very small compared to n, we can maintain G faster than by using the data structure of Lemma 11.

▶ **Lemma 12.** *After preprocessing in $O(n)$ time, we can repeatedly solve the bordered vertex merging problem without edge insertions for planar simple graphs G_0 with $t = O(\log^4 \log^4 n)$ vertices in $O(t)$ time.*

5.2 A Multi-Level Data Structure

Recall that our goal is to maintain G under contractions. Below we describe in detail how to take advantage of graph partitioning and bordered vertex merging data structures to obtain a linear time solution. To simplify the further presentation, we assume that the initial version $G_0 = (V_0, E_0)$ of G is simple and of constant degree. The standard reduction assuring that is described in the full version [10].

We build an r-division $\mathcal{R} = \{P_1, P_2, \ldots, \}$ of G with $r = \log^4 n$, where $n = |V_0|$ (see Lemma 1). Then, for each piece $P_i \in \mathcal{R}$, we build an r-division $\mathcal{R}_i = \{P_{i,1}, P_{i,2}, \ldots\}$ of P_i with $r = \log^4 \log^4 n$. By Lemma 1, building all the necessary pieces takes $O(n)$ time in total. Since G_0 is of constant degree, any vertex $v \in V_0$ is contained in $O(1)$ pieces of $\mathcal{R}$. Analogously, for any $v \in P_i$, v is contained in $O(1)$ pieces of $\mathcal{R}_i$.

As G undergoes contractions, let $\phi : V_0 \to V$ be a mapping such that for each $v \in V_0$, v "has been merged" into $\phi(v)$. As we later describe, a vertex resulting from contracting an edge uv will be called either u or v, which guarantees that $V \subseteq V_0$ at all times. Of course, initially $\phi(v) = v$ for each $v \in V = V_0$.

Let $\overline{G} = (V, \overline{E})$ denote the maximal simple subgraph of G, i.e., the graph G with self-loops discarded and each group Y of parallel edges replaced with a single edge $\alpha(Y)$. The key component of our data structure is a 3-level set of (possibly micro-) bordered vertex merging data structures $\Pi = \{\pi\} \cup \{\pi_i : P_i \in \mathcal{R}\} \cup \{\pi_{i,j} : P_i \in \mathcal{R}, P_{i,j} \in \mathcal{R}_i\}$. The data structures Π form a tree such that π is the root, $\{\pi_i : P_i \in \mathcal{R}\}$ are the children of π and $\{\pi_{i,j} : P_{i,j} \in \mathcal{R}_i\}$ are the children of π_i. For $\mathcal{D} \in \Pi$, let $par(\mathcal{D})$ be the parent of $\mathcal{D}$ and let $A(\mathcal{D})$ be the set of ancestors of $\mathcal{D}$. We call the value $h(\mathcal{D}) = |A(\mathcal{D})|$ a *level* of $\mathcal{D}$. The data structures of levels 0 and 1 are stored as data structures of Lemma 11, whereas the data structures of level 2 are stored as micro structures of Lemma 12.

Each data structure $\mathcal{D} \in \Pi$ has a defined set $V_\mathcal{D} \subseteq V_0$ of *interesting vertices*, defined as follows: $V_\pi = \partial\mathcal{R}$, $V_{\pi_i} = \partial P_i \cup \partial\mathcal{R}_i$ and $V_{\pi_{i,j}} = V(P_{i,j})$. The data structure $\mathcal{D}$ maintains a certain subgraph $G_\mathcal{D}$ of $\overline{G}$ defined inductively as follows (recall that we define $G_1 \setminus G_2$ to be a graph containing all vertices of G_1 and edges of G_1 that do not belong to G_2)

$$G_\mathcal{D} = \overline{G}[\phi(V_\mathcal{D})] \setminus \left(\bigcup_{\mathcal{D}' \in A(\mathcal{D})} G_{\mathcal{D}'} \right).$$

▶ **Fact 13.** *For any $\mathcal{D} \in \Pi$, $G_\mathcal{D}$ is a minor of G_0.*

▶ **Fact 14.** *For any $uv = e \in \overline{E}$, there exists $\mathcal{D} \in \Pi$ such that $e \in E(G_\mathcal{D})$.*

Each $\mathcal{D} \in \Pi$ is initialized with the graph $G_\mathcal{D}$, according to the initial mapping $\phi(v) = v$ for any $v \in V_0$. We define the set of *ancestor vertices* $AV_\mathcal{D} = V_\mathcal{D} \cap \left(\bigcup_{\mathcal{D}' \in A(\mathcal{D})} V_{\mathcal{D}'}\right)$.

Now we discuss what it means for the bordered vertex merging data structure $\mathcal{D}$ to maintain the graph $G_\mathcal{D}$. Note that the vertex set used to initialize $\mathcal{D}$ is $V_\mathcal{D}$. We write $\phi_\mathcal{D}, \phi_\mathcal{D}^{-1}$ to denote the mappings ϕ, ϕ^{-1} maintained by $\mathcal{D} \in \Pi$, respectively. Throughout a sequence of contractions, we maintain the following invariants for any $\mathcal{D} \in \Pi$:

- There is a 1-1 mapping between the sets $\phi(V_\mathcal{D})$ and $\phi_\mathcal{D}(V_\mathcal{D})$ such that for the corresponding vertices $x \in \phi(V_\mathcal{D})$ and $y \in \phi_\mathcal{D}(V_\mathcal{D})$ we have $\phi_\mathcal{D}^{-1}(y) = \phi^{-1}(x) \cap V_\mathcal{D}$. We also say that x is *represented* in $\mathcal{D}$ in this case.
- There is an edge $xy \in E(G_\mathcal{D})$ if and only if there is an edge $x'y'$ in the graph maintained by $\mathcal{D}$, where $x', y' \in \phi_\mathcal{D}(V_\mathcal{D})$ are the corresponding vertices of x and y, respectively.
- The border set $B_\mathcal{D}$ of $\mathcal{D}$ is always equal to $\phi_\mathcal{D}(AV_\mathcal{D})$.

Thus, the graph maintained by $\mathcal{D}$ is isomorphic to $G_{\mathcal{D}}$ but can technically use a different vertex set. Observe that in $G_{\mathcal{D}}$ there are no edges between the vertices $\phi(AV_{\mathcal{D}})$ and the following fact describes how this is reflected in $\mathcal{D}$.

▶ **Fact 15.** *In the graph stored in $\mathcal{D}$, no two vertices of $B_{\mathcal{D}}$ are adjacent.*

Note that as the sets $V_{\mathcal{D}}$ and $V_{\mathcal{D}'}$ might overlap for $\mathcal{D} \neq \mathcal{D}'$, the vertices of V can be represented in multiple data structures.

▶ **Lemma 16.** *Suppose for $v \in V$ we have $v \in V(G_{\mathcal{D}_1})$ and $v \in V(G_{\mathcal{D}_2})$. Then, $v \in V(G_{\mathcal{D}})$, where $\mathcal{D}$ is the lowest common ancestor of $\mathcal{D}_1$ and $\mathcal{D}_2$.*

By Lemma 16, each vertex $v \in V$ is represented in a unique data structure of minimal level, a lowest common ancestor of all data structures where v is represented. We denote such a data structure by $\mathcal{D}(v)$. Observe that for any $\mathcal{D} \in \Pi$ the vertices $\{v : \mathcal{D}(v) = \mathcal{D}\}$ are represented in $\mathcal{D}$ by $\phi_{\mathcal{D}}(V_{\mathcal{D}}) \setminus \phi_{\mathcal{D}}(AV_{\mathcal{D}})$.

We now describe the way we index the vertices of V. This is required, as upon a contraction, our data structure returns an identifier of a new vertex. We also reuse the names of the initial vertices V_0, as the bordered vertex merging data structures do. Namely, a vertex $v \in V$ is labeled with $\phi_{\mathcal{D}(v)}(v') \in V_0$, where v' represents v in $\mathcal{D}(v)$.

Note that, as the bordered vertex merging data structures always merge one vertex involved into the other, for any $\mathcal{D} \in \Pi$ we have $\phi_{\mathcal{D}}(V_{\mathcal{D}}) \setminus \phi_{\mathcal{D}}(AV_{\mathcal{D}}) \subseteq V_{\mathcal{D}} \setminus AV_{\mathcal{D}}$. Hence the label sets used by distinct sets $\{v : \mathcal{D}(v) = \mathcal{D}\}$ are distinct, since the sets of the form $V_{\mathcal{D}} \setminus AV_{\mathcal{D}}$ are pairwise disjoint. Such a labeling scheme makes it easy to find the data structure $\mathcal{D}(v)$ by looking only at the label.

For brevity, in the following we sometimes do not distinguish between the set V and the set of labels $\bigcup_{\mathcal{D}\in\Pi} (\phi_{\mathcal{D}}(V_{\mathcal{D}}) \setminus \phi_{\mathcal{D}}(AV_{\mathcal{D}}))$.

▶ **Lemma 17.** *Let $uv = e \in \overline{E}$ and $h(\mathcal{D}(u)) \geq h(\mathcal{D}(v))$. Then $e \in E(G_{\mathcal{D}(u)})$ and either $\mathcal{D}(u) = \mathcal{D}(v)$ or $\mathcal{D}(u)$ is a descendant of $\mathcal{D}(v)$.*

▶ **Lemma 18.** *Let uv be an edge of some $G_{\mathcal{D}}$, $\mathcal{D} \in \Pi$. If $\{u, v\} \subseteq V(G_{\mathcal{D}'})$, where $\mathcal{D}' \neq \mathcal{D}$, then $\mathcal{D}'$ is a descendant of $\mathcal{D}$ and both u and v are represented as border vertices of $\mathcal{D}'$.*

▶ **Lemma 19.** *Let $v \in \phi(V_{\mathcal{D}})$, where $\mathcal{D} \in \Pi$. Then, v is represented in $O(|\phi_{\mathcal{D}}^{-1}(v)|)$ data structures $\mathcal{D}'$ such that $par(\mathcal{D}') = \mathcal{D}$.*

We also use the following auxiliary components for each $\mathcal{D} \in \Pi$:

- For each $x \in \phi_{\mathcal{D}}(AV_{\mathcal{D}})$ we maintain a pointer $\beta_{\mathcal{D}}(x)$ into $y \in \phi_{par(\mathcal{D})}(AV_{\mathcal{D}})$, such that x and y represent the same vertex of the maintained graph G.
- A dictionary (we use a balanced BST) $\gamma_{\mathcal{D}}$ mapping a pair $(\mathcal{D}', x)$, where $\mathcal{D}'$ is a child of $\mathcal{D}$ and $x \in \phi_{\mathcal{D}}(V_{\mathcal{D}})$, to a vertex $y \in \phi_{\mathcal{D}'}(AV_{\mathcal{D}})$ iff x and y represent the same vertex of V.

Another component of our data structure is the forest $\mathcal{T}$ of reported parallelisms: for each reported parallelism $e \to \alpha(e)$, we make e a child of $\alpha(e)$ in $\mathcal{T}$. Note that the forest $\mathcal{T}$ allows us to go through all the edges parallel to $\alpha(e)$ in time linear in their number.

▶ **Lemma 20.** *For $v_0 \in V_0$, we can compute $\phi(v_0)$ and find $\mathcal{D}(\phi(v_0))$ in $O(1)$ time.*

▶ **Lemma 21.** *Let $v \in V(G_{\mathcal{D}})$. For any $\mathcal{D}'$, such that $par(\mathcal{D}') = \mathcal{D}$, we can compute the vertex v' representing v in $G_{\mathcal{D}'}$ (or detect that such v' does not exist) in $O(\log |V_{\mathcal{D}}|)$ time.*

We now describe how to implement the call $(s, P, L) := \texttt{contract}(e)$, where $uv = e \in E$, $u, v \in V$. Suppose the initial endpoints of e were $u_0, v_0 \in V_0$. First, we iterate through the tree $T_e \in \mathcal{T}$ containing e to find $\alpha(e)$. By Lemma 20, we can find the vertices u, v along with the respective data structures $\mathcal{D}(u), \mathcal{D}(v)$, based on u_0, v_0 in $O(1)$ time. Assume wlog. that $h(\mathcal{D}(u)) \geq h(\mathcal{D}(v))$. By Lemma 17, $\alpha(e)$ is an edge of $G_{\mathcal{D}(u)}$. Although we are asked to contract e, we conceptually contract $\alpha(e)$, by issuing a merge of u and v to $\mathcal{D}(u)$. To reflect that we were actually asked to contract e, we include all the edges of $T_e \setminus \{e\}$ in L as self-loops. The merge might make $\mathcal{D}(u)$ report some parallelisms $e_1 \to e_2$. In such a case we report $e_1 \to e_2$ to the user (by including it in P) and update the forest $\mathcal{T}$.

We now have to reflect the contraction of e in all the required data structures $\mathcal{D} \in \Pi$, so that our invariants are satisfied. Assume wlog. that u is merged into v in $\mathcal{D}$. If before the contraction, both u and v were the vertices of some $G_{\mathcal{D}'}$, $\mathcal{D}' \neq \mathcal{D}$, then by Lemma 18, $\mathcal{D}'$ is a descendant of $\mathcal{D}$. By a similar argument as in the proof of Lemma 11, we can afford to iterate through $\phi_{\mathcal{D}}^{-1}(u)$ without increasing the asymptotic performance of the u-into-v merge performed by $\mathcal{D}$, as long as we spend $O(\log |V_{\mathcal{D}}|)$ time per element of $\phi_{\mathcal{D}}^{-1}(u)$. By Lemma 19, there are $O(|\phi_{\mathcal{D}}^{-1}(u)|)$ data structures $\mathcal{D}_1, \mathcal{D}_2, \ldots$ that are the children of $\mathcal{D}$ and contain the representation of u. For each such $\mathcal{D}_i$, we first use the dictionary $\gamma_{\mathcal{D}}$ to find the vertex x representing u in $\mathcal{D}_i$, and update $\beta_{\mathcal{D}_i}(x)$ to v. Then, using Lemma 21, we check whether $v \in V(G_{\mathcal{D}_i})$ in $O(\log |V_{\mathcal{D}}|)$ time. If not, we set $\gamma_{\mathcal{D}}(\mathcal{D}_i, v)$ to x. Otherwise, we merge u and v in $\mathcal{D}_i$ and handle this merge – in terms of updating the auxiliary components β and γ – analogously as for $\mathcal{D}$. This is legal, as $u, v \in \phi_{\mathcal{D}_i}(AV_{\mathcal{D}_i})$ and thus u and v are border vertices in $\mathcal{D}_i$, by Fact 15. The merge may cause $\mathcal{D}_i$ to report some parallelisms. We handle them as described above in the case of the data structure $\mathcal{D}$. Note however that merging border vertices cannot cause reporting of new border edges (i.e., those with both endpoints in $B_{\mathcal{D}_i}$).

The merge of u and v in $\mathcal{D}$ might also create some new edges $e' = xy$ between the vertices $\phi_{\mathcal{D}}(AV_{\mathcal{D}})$ in $G_{\mathcal{D}}$. Note that in this case $\mathcal{D}$ reports xy as a border edge and also we know that $h(\mathcal{D}(x)) < h(\mathcal{D})$ and $h(\mathcal{D}(y)) < h(\mathcal{D})$. Hence, e' should end up in some of the ancestors of $\mathcal{D}$. We insert e' to $par(\mathcal{D})$. $par(\mathcal{D})$ might also report xy as a border edge and in that case e' is inserted to the grandparent of $\mathcal{D}$. It is also possible that e' will be reported a parallel edge in some of the ancestors of $\mathcal{D}$: in such a case an appropriate directed parallelism is added to P.

Note that all the performed merges and edge insertions are only used to make the graphs represented by the data structures satisfy their definitions. Fact 13 implies that the represented graphs remain planar at all times.

We now describe how the other operations are implemented. To compute $u, v \in V$ such that $\{u, v\} = \texttt{vertices}(e)$, where $e \in E$, we first use Lemma 20 to compute $u = \phi(u_0)$ and $v = \phi(v_0)$, where u_0, v_0 are the initial endpoints of e. Clearly, this takes $O(1)$ time.

To maintain the values $\texttt{deg}(v)$ of each $v \in V$, we simply set $\texttt{deg}(s) := \texttt{deg}(u) + \texttt{deg}(v) - 1$ after a call $(s, P, L) := \texttt{contract}(e)$. Additionally, for each directed parallelism $e_1 \to e_2$ we decrease $\texttt{deg}(x)$ and $\texttt{deg}(y)$ by one, where $\{x, y\} = \texttt{vertices}(e_1)$.

For each $u \in V$ we maintain a doubly-linked list $\mathcal{E}(u) = \{\alpha(uv) : uv \in \overline{E}\}$. Additionally, for each $e \in \overline{E}$ we store the pointers to the two occurrences of e in the lists $\mathcal{E}(\cdot)$. Again after a call $(s, P, L) := \texttt{contract}(e)$, where $e = uv$, we set $\mathcal{E}(s)$ to be a concatenation of the lists $\mathcal{E}(u)$ and $\mathcal{E}(v)$. Finally, we remove all the occurrences of edges $\{\alpha(e)\} \cup \{e_1 : (e_1 \to e_2) \in P\}$ from the lists $\mathcal{E}(\cdot)$. Now, the implementation of the iterator $\texttt{neighbors}(u)$ is easy, as the endpoints not equal to u of the edges in $\mathcal{E}(u)$ form exactly the set $N(u)$.

▶ **Lemma 22.** *The operations* `vertices`, `deg` *and* `neighbors` *run in* $O(1)$ *worst-case time.*

To support the operation $\texttt{edge}(u, v)$ in $O(1)$ time, we first turn all the dictionaries $\gamma_{\mathcal{D}}$ into hash tables with $O(1)$ expected update time and $O(1)$ worst-case query time [3]. Our data structure thus ceases to be deterministic, but we obtain a more efficient version of Lemma 21 that allows us to compute the representation of a vertex in a child data structure $\mathcal{D}'$ in $O(1)$ time. By Lemma 17, the edge uv can be contained in either $\mathcal{D}(u)$ or $\mathcal{D}(v)$, whichever has greater level. Wlog. suppose $h(\mathcal{D}(u)) \geq h(\mathcal{D}(v))$. Again, by Lemma 17, $\mathcal{D}(u)$ is a descendant of $\mathcal{D}(v)$. Thus, we can find v in $\mathcal{D}(u)$ by applying Lemma 21 at most twice.

▶ **Lemma 23.** *If the dictionaries $\gamma_{\mathcal{D}}$ are implemented as hash tables, the operation* `edge` *runs in $O(1)$ worst-case time.*

▶ **Lemma 24.** *The cost of all operations on the data structures $\mathcal{D} \in \Pi$ is $O(n)$.*

Proof of Theorem 2. To initialize our data structure, we initialize all the data structures $\mathcal{D} \in \Pi$ and the auxiliary components. This takes $O(n)$ time. The time needed to perform any sequence of operations `contract` is proportional to the total time used by the data structures Π, as the cost of maintaining the auxiliary components can be charged to the operations performed by the individual structures of Π. By Lemma 24, this time is $O(n)$. If the dictionaries $\gamma_{\mathcal{D}}$ are implemented as hash tables, this bound is valid only in expectation.

By combining the above with Lemmas 22 and 23, the theorem follows. ◀

References

1 Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representation of sparse graphs. In *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, Proceedings*, pages 342–351, 1999. doi:10.1007/3-540-48447-7_34.

2 Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Veronika Loitzenbauer, and Nikos Parotsidis. Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1900–1918, 2017. doi:10.1137/1.9781611974782.124.

3 Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994. doi:10.1137/S0097539791194094.

4 Jack Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, pages 449–467, 1965.

5 Greg N. Frederickson. On linear-time algorithms for five-coloring planar graphs. *Inf. Process. Lett.*, 19(5):219–224, 1984. doi:10.1016/0020-0190(84)90056-5.

6 Harold N. Gabow, Haim Kaplan, and Robert Endre Tarjan. Unique maximum matching algorithms. *J. Algorithms*, 40(2):159–183, 2001. Announced at STOC'99. doi:10.1006/jagm.2001.1167.

7 Dora Giammarresi and Giuseppe F. Italiano. Decremental 2- and 3-connectivity on planar graphs. *Algorithmica*, 16(3):263–287, 1996. doi:10.1007/BF01955676.

8 Michael T. Goodrich. Planar separators and parallel polygon triangulation. *J. Comput. Syst. Sci.*, 51(3):374–389, 1995. doi:10.1006/jcss.1995.1076.

9 Jens Gustedt. Efficient union-find for planar graphs and other sparse graph classes. *Theor. Comput. Sci.*, 203(1):123–141, 1998. doi:10.1016/S0304-3975(97)00291-0.

10 Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, Eva Rotenberg, and Piotr Sankowski. Contracting a planar graph efficiently, 2017. arXiv:1706.10228.

11 David R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-out algorithm. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'93, pages 21–30, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

12 Philip N. Klein and Shay Mozes. Optimization algorithms for planar graphs, 2017. URL: `http://planarity.org`.

13 Philip N. Klein, Shay Mozes, and Christian Sommer. Structured recursive separator decompositions for planar graphs in linear time. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 505–514, 2013. `doi:10.1145/2488608.2488672`.

14 Jakub Łącki and Piotr Sankowski. Optimal decremental connectivity in planar graphs. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, pages 608–621, 2015. `doi:10.4230/LIPIcs.STACS.2015.608`.

15 Martin Mareš. Two linear time algorithms for mst on minor closed graph classes. *Archivum mathematicum*, 40(3):315–320, 2002.

16 Tomomi Matsui. The minimum spanning tree problem on a planar graph. *Discrete Applied Mathematics*, 58(1):91–94, 1995. `doi:10.1016/0166-218X(94)00095-U`.

17 David W. Matula, Yossi Shiloach, and Robert E. Tarjan. Two linear-time algorithms for five-coloring a planar graph. Technical report, Stanford University, Stanford, CA, USA, 1980.

18 Neil Robertson, Daniel P. Sanders, Paul Seymour, and Robin Thomas. Efficiently four-coloring planar graphs. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC'96, pages 571–575, New York, NY, USA, 1996. ACM. `doi:10.1145/237814.238005`.

19 Freek van Walderveen, Norbert Zeh, and Lars Arge. Multiway simple cycle separators and I/O-efficient algorithms for planar graphs. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 901–918, 2013. `doi:10.1137/1.9781611973105.65`.

Minimizing Maximum Flow Time on Related Machines via Dynamic Posted Pricing*

Sungjin Im[†1], **Benjamin Moseley**[‡2], **Kirk Pruhs**[§3], and **Clifford Stein**[¶4]

1 **Electrical Engineering and Computer Science, University of California at Merced, CA, USA**
sim3@ucmerced.edu
2 **Washington University in St. Louis, MO, USA**
bmoseley@wustl.edu
3 **Dept. of Computer Science, University of Pittsburgh, PA, USA**
kirk@cs.pitt.edu
4 **Department of Industrial Engineering and Operations Research, Columbia University, New York, NY, USA**
cliff@ieor.columbia.edu

Abstract

We consider a setting where selfish agents want to schedule jobs on related machines. The agent submitting a job picks a server that minimizes a linear combination of the server price and the resulting response time for that job on the selected server. The manager's task is to maintain server prices to (approximately) optimize the maximum response time, which is a measure of social good. We show that the existence of a pricing scheme with certain competitiveness is equivalent to the existence of a monotone immediate-dispatch algorithm. Our main result is a monotone immediate-dispatch algorithm that is $O(1)$-competitive with respect to the maximum response time.

1998 ACM Subject Classification F.2.2 [Nonnumerical Algorithms and Problem]: Sequencing and scheduling

Keywords and phrases Posted pricing scheme, online scheduling, related machines, maximum flow time, competitiveness analysis

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.51

1 Introduction

1.1 Motivation and Background

Many large companies foster a competitive internal environment to create flexibility, challenge the status quo, and motivate employees. However, it is recognized that internal competition has to be managed so that the costs do not outweigh the benefits [9, 6]. In this paper, we consider one such management task. Namely, we consider managing compute servers, used

* This work was partially done while the authors were participating in the Algorithms and Uncertainty program at the Simons Institute for the Theory of Computing at the University of California at Berkeley.
† Supported in part by NSF grants CCF-1409130 and CCF-1617653.
‡ Supported in part by a Google Research Award, a Yahoo Research Award and NSF Grant CCF-1617724.
§ Supported in part by NSF grants CCF-1421508 and CCF-1535755, and an IBM Faculty Award.
¶ Supported in part by NSF grant CCF-1421161.

by competing self-interested agents, to optimize social good. As agents are self-interested, you would expect them to greedily choose the server that will finish their task first. In the setting that we consider, this can lead to schedules with highly suboptimal social good. Thus the manager might reasonably want to implement some mechanism that will incentivize the agents to produce a schedule with high social good. Following the lead of [10] we consider a dynamic posted price mechanism; that is, the manager maintains a dynamically changing price for each server (like Amazon's EC2). Thus a self-interested agent would take into account both response time and price when selecting a server.

In [10] various common models of compute servers are considered, and it is assumed that:

- all agents sequentially select servers at the same moment of time,
- jobs on one machine are scheduled in a First-Come-First-Served manner,
- agents greedily pick the server that minimizes a linear combination of the resulting response time of the agent's job and the current price for that server, and
- the social good is measured by (technically the inverse of) the makespan of the schedule.

The main result in [10] is a pricing scheme that guarantees that selfish agents construct a schedule that is $O(1)$-competitive with respect to makespan on related machines. In this scheme the prices are essentially set so that the resulting schedule is identical to the schedule produced by the (non-pricing based) online algorithm, called Slow-Fit in [4], that was shown to be $O(1)$-competitive in [3]. Slow-Fit assigns each job to the slowest machine that would not result in a response time greater than some constant times the current estimate of the optimal makespan (and doubles the estimate if this isn't possible).

[11] posed the question of whether this $O(1)$-approximation result could be extended to the (arguably more natural) setting where agents may submit jobs over time, and the social good is the natural generalization of makespan, namely the maximum response time of any job. The maximum response time is the maximum time a job waits in the system after its arrival to be completed.

The combinatorics of the scheduling of related machines with the objective of maximum flow are a bit tricky, and it wasn't until recently that any $O(1)$-competitive online algorithm (or even any polynomial-time $O(1)$-approximation offline algorithm) was known [5]. [5] called this online algorithm Double-Fit. Double-Fit delays assigning jobs, and collects them into batches. Periodically the jobs in a batch are assigned to machines. In assigning a batch of jobs, the jobs are considered in decreasing order of size. Each job is then assigned to a server using essentially a two level generalization of Slow-Fit. The conclusion of [5] states that:

> Note that our algorithm Double-Fit is not immediate dispatch, i.e., it does not dispatch a job to a machine immediately upon arrival. We are unable to extend the ideas here to obtain an $O(1)$-competitive immediate dispatch algorithm, and it is not clear to us whether such an algorithm exists.

Immediate dispatch algorithms have some practical advantages, most notably, they do not require the maintenance of a global queue of unprocessed work, which could be a potential bottleneck to scalability.

1.2 Our Results

We start by observing that the open questions from [11] and [5] are related. More precisely, we observe that a monotone immediate-dispatch algorithm can be converted into a dynamic posted price algorithm, preserving the competitive ratio. Similarly, we observe that a posted price algorithm can be converted into a monotone immediate-dispatch algorithm, preserving the competitive ratio. An algorithm is monotone if the speed of the server that an agent

would pick is monotonically increasing in the size of a job. Monotonicity is a natural property in that the bigger a job is, the more critical it is that it be assigned to a fast server (in the extreme, an infinite sized job must be assigned to the fastest server if one is to achieve a competitive ratio independent of server speeds).

Using this equivalence, we establish the existence of an $O(1)$-competitive posted price scheme by giving a monotone immediate-dispatch $O(1)$-competitive algorithm, which we call Immediate-Double-Fit (IDF). *Thus we affirmatively answer both the open question from [11] and the open question from [5].*

The algorithm IDF immediately assigns jobs using the same strategy as Double-Fit does. After observing that IDF is monotone, we turn to analyzing IDF's competitiveness. The fact that Double-Fit assigns jobs within a batch in size order was critical to the analysis of Double-Fit in [5]. Intuitively the main difficulty of analyzing IDF is that there may be no relationship between the size of a job and when it is assigned a server, thus making the analysis of Double-Fit in [5] inapplicable. Not surprisingly, the key to our being able to analyze IDF was finding the "right" inductive hypothesis, which is substantially different than the inductive hypothesis used in [5]. Perhaps somewhat surprisingly, our inductive hypothesis is actually simpler than the one used in [5], and as a consequence, we also get a slightly better bound on the competitive ratio of IDF, namely 25/2, than the bound of 27/2 on the competitive ratio of Double-Fit obtained in [5].

One intuitive take away point from these results is that dynamic posted prices gives management essentially the same power as being able to impose arbitrary job to server assignments, when the setting is related machines and the objective is maximum flow time.

1.3 Other Related Work

[10] showed that for unrelated machines, every pricing scheme can lead to schedules that are $\Omega(m)$-competitive with respect to makespan. In the unrelated machine setting the processing time of a job is machine dependent. They also showed that static pricing schemes (where server prices do not change over time) are in some sense equivalent to the natural greedy algorithm.

Intuitively prices are necessary to achieve $O(1)$-competitiveness for maximum response time on related machine. To understand why, note that it is well-known that FIFO is optimal on a single machine for the objective of maximum response time. In addition to being optimal, FIFO is the unique scheduling policy that allows each agent to know with certainty the response time of its job on each server. However, [5] showed that the natural greedy algorithm is $\Omega(m)$-competitive for maximum response time on related machines.

There is a significant literature on mechanism design for scheduling, starting with the paper [14] that instigated the study of mechanism design within the algorithmic community. Much of this work focuses on finding and/or analyzing coordination mechanisms with respect to the price of anarchy, which compares the social good of some equilibrium to the optimal social good. We mention a few such results that seem most closely related to the results in this paper. A coordination mechanism for identical machines with constant price of anarchy with respect to makespan can be found in [7]. [13] studies coordination mechanisms for four classes of multiprocessor machine scheduling problems and derive upper and lower bounds for the price of anarchy with respect to makespan of these mechanisms. [1] considers coordination mechanisms for unrelated machines in which agents control subsets of jobs, and each player's objective is to minimize the weighted sum of completion time of her jobs.

There is a significant literature on mechanism design using posted prices, most of it focused on auctions and markets (see [15] for an overview). [8] focuses on server problems, motivated in part by the SFPark system (SF-park.org), which sets parking prices in San

Francisco based on parking congestion. [8] gives pricing schemes for the classical problems of k-server, metrical task systems, and metrical matching that in some cases achieve competitive ratios that are close to the optimal competitive ratios for general online algorithms.

There is significant literature on online scheduling. Good starting points into this literature are [16] and [12]. Probably the most relevant results come from [2], which gave scalable algorithms for minimizing maximum flow time on unrelated machines, and for minimizing weighted maximum flow time on related machines. Resource augmentation is required for both problems in order to achieve constant approximation.

2 Notation and Definitions

In the standard related machines environment the m servers/machines have associated speeds, $s_1, \ldots, s_m$. We assume without loss of generality that $s_1 \leq s_2 \leq \ldots \leq s_m$. A collection of n jobs arrive over time. The release time r_j of job j is when it is submitted to be scheduled. Further, each job j has a size p_j. A (nonpreemptive) schedule specifies for each job j, a starting time λ_j and an assigned server i_j, with the restriction the time intervals $[\lambda_j, C_j]$ should be disjoint for all jobs assigned to the same machine. Here $C_j = \lambda_j + p_j/s_{i_j}$ is the time that job j completes. If j has been run on server i for $\tau \leq p_j/s_i$ units of time, then its unprocessed volume is $p_j - s_i\tau$. The *flow/response time* of the job is defined as $F_j := C_j - r_j$, and the objective we consider is to minimize the maximum flow time $F_{\max} = \max_j F_j$. The makespan of a schedule is the maximum completion time.

In this paper, we assume that an online scheduler learns job j's size p_j at time r_j when it is released. An online scheduler is called *immediate dispatch* if it always assigns a job to a machine at the job's release time. The scheduler need not start job j at time r_j, but the scheduler must make an irrevocable decision about which machine the job will eventually run on. Let $A_t(p)$ be the speed of the machine that an algorithm A would assign a job of size p to if it was released at the current time t; here it is assumed that job identity plays no role in A's assignment decision. Then algorithm A is monotone if for all possible instances, and for all possible t, $A_t(p)$ is non-decreasing in p.

A dynamic posted pricing scheme is a special type of online scheduler. Jobs assigned to a server are processed in First-Come-First-Served order. So every processor is always processing the earliest released, uncompleted job assigned to it. The online schedule maintains a dynamically changing price for each server. Let $c_i(t)$ denote the price/cost for server i at time t. Let $L_i(t)$ denote the unprocessed volume of jobs previously assigned to server i at time t, divided by machine i's speed. In other words, it takes $L_i(t)$ units of time for machine i to complete its unprocessed workload assuming that no more jobs arrive. Then job j is assigned to the server i that minimizes $L_i(r_j) + p_j/s_i + c_i(t)$. Intuitively the job selfishly assigns itself to the machine that minimizes its flow time plus the machine cost. It is important that the prices $c_i(r_j)$ are posted prior to job j'arrival; that is, they cannot depend on the value of p_j, and may only depend on past events. For notational convenience we may drop the current time from the notation if it is clear from the context. For example, we may simply use c_i in place of $c_i(t)$.

One take away point from [10], as well as earlier work on posted price mechanisms, is that dealing with ties can be annoying. The issue of ties manifests itself in two ways in our setting. Firstly, in order to show that our pricing scheme is monotone, we need that the machine speeds are distinct, that is that $s_1 < s_2 < \ldots < s_m$. This can be achieved with probability one by decreasing each machine speed by some random infinitesimal amount, at the cost of raising the competitive ratio by an infinitesimal amount. The mechanism

can simulate a slower machine by delaying the start date of each job appropriately. Thus technically our mechanism involves both pricing and slightly delaying some jobs.

The second way in which the issue of ties manifests itself is when there are two different servers that simultaneously minimize $L_i(r_j) + p_j/s_i + c_i(t)$. But this is also handled by the random decrement of the processor speeds, as this sort of tie will then arise with probability zero. Thus we assume in our analysis that there is a unique server i that minimizes $L_i(r_j) + p_j/s_i + c_i(t)$.

[10] takes a different approach to this second issue of ties. They assume that in the case of ties, the job may be adversarially assigned to any minimizing server. This has the advantage of imposing minimal assumptions on the actions of the agent, but it has the disadvantage of cluttering/complicating the algorithmic design/analysis process. The majority of the effort in [10] is related to handling ties.

3 Algorithm and Analysis

In Subsection 3.1 we establish the equivalence of posted price algorithms and monotone immediate-dispatch algorithms. In Subsection 3.2 we describe the Immediate-Double-Fit algorithm. In Subsection 3.3 we first note that the Immediate-Double-Fit algorithm is monotone, and then give an inductive argument bounding its competitiveness.

3.1 Equivalence between Monotonicity and Post-pricing Scheme

▶ **Lemma 1.** *An immediate-dispatch, monotone algorithm A can be converted into a posted pricing algorithm/scheme B. In particular, there is a pricing algorithm B where each job is assigned to the same machines in both A and B. Thus, both algorithms produce exactly the same schedule.*

Proof. We explain how to convert A into B. Assume that a job j is released at time t. Price any machine on which A would never run j no matter what its processing time is at infinite. For notational convenience, drop them from our ordering and assume that m machines remain. Assume according to algorithm A that at size p_i the speed of the selected processor changes from s_i to s_{i+1} for $i \in [1, m-1]$; more precisely, $A(p) \leq s_i$ for all $p < p_i$ and $A(p) \geq s_{i+1}$ for all $p > p_i$. Define $g_c(p)$ as the cost function that takes a job size p and returns the minimum cost the job has to pay under the pricing c. Let L_i denote the load on machine i just before j is assigned. By setting the price vector c so that the following is satisfied for all $1 \leq i \leq m-1$:

$$L_i + c_i + p_i/s_i = L_{i+1} + c_{i+1} + p_i/s_{i+1},$$

we get a cost function $g_c(p)$ where the cost for a job of size $p \in (p_{i-1}, p_i)$ is minimized on machine i. Hence under this post-pricing scheme, each job is assigned exactly to the same machine as it were by the given algorithm A. Also by setting c_1 to be sufficiently large and using the fact that $s_1 < s_2 < ... < s_m$, we can ensure that all prices are positive. ◀

Although it is not needed to establish our main results, we now prove the converse of Lemma 1.

▶ **Lemma 2.** *A pricing algorithm A is an immediate dispatch, monotone algorithm.*

Proof. It it obvious that it is an immediate dispatch algorithm. To establish monotonicity, consider the arrival of a job. Let c_i be the price for machine i. Note that job of size p pays $L_i + c_i + p/s_i$ if it chooses machine i. Let $g(p)$ denote the minimum cost a job of size p has to

pay on any machine. Due to the greedy nature of clients, we have $g(p) := \min_i(L_i + c_i + p/s_i)$, which is a piece-wise linear function. This implies that if there is a value of p for which machine i minimizes the cost, the set of such values must form an interval. Let p_i be the size of a job where a greedy client would change from a machine with speed s_i to a machine with speed s_k when we increase p. Note that the uniqueness of p_i follows from the above observation. Knowing that $L_i + c_i + p_i/s_i = L_k + c_k + p_i/s_k$ and $L_i + c_i + p/s_i > L_k + c_k + p/s_k$ for $p > p_i$, we conclude that $s_i < s_k$, thus proving monotonicity of the algorithm. ◀

3.2 Description of the Immediate-Double-Fit Algorithm

We start by making some simplifying assumptions, and defining some concepts and notation. Assume for the moment that the algorithm knows Opt, the objective value of the optimal solution. If not, we show at the end of the analysis how to remove this assumption using the standard doubling trick. For simplicity, we assume that jobs arrive at distinct times. We can easily extend our analysis to remove this assumption by considering jobs released at the same time from the largest to smallest, but this would complicate the analysis.

Time is broken into epochs. The length of a epoch depends on when jobs are released. Define epochs to be of length $\epsilon\mathsf{Opt}$, where ϵ is an arbitrarily small parameter such that at most one job arrives in each epoch. The first epoch begins at time 0 before any job arrives. At the start of an epoch we assign the job that arrived in the last epoch. We now describe how to assign an arriving job j. Let $[i_j, m]$ be the machines i on which p_j/s_i is at most Opt. The algorithm is parameterized by constants $\alpha, \beta \geq 1$ which will be fixed later.

We are now ready to describe the Immediate-Double-Fit (IDF) Algorithm. When a new job j arrives, IDF does the following:

1. If there is a machine in $[i_j, m]$ with load less than $\alpha\mathsf{Opt}$, then schedule j on the slowest such machine. We say in this case that j was placed in the *saturation* phase.
2. Else if there is a machine in $[i_j, m]$ with load less than $\beta\mathsf{Opt}$ then schedule j on the slowest such machine. We say in this case that j was placed in the *slow fit* phase.
3. Else the algorithm admits failure.

3.3 Analysis

We begin by establishing in Lemma 3 that the IDF algorithm is monotone. We then turn to analyzing IDF's competitiveness. We show that IDF never admits failure for proper choice of α and β, under the assumption that its estimation of Opt is correct. Noting that the algorithm is immediate dispatch for sufficiently small ϵ the algorithm can be converted to a pricing algorithm as shown in Lemma 3. We then finish by showing how to apply the standard doubling trick to remove the assumption that the algorithm knows Opt.

▶ **Lemma 3.** *Algorithm A is a monotone algorithm.*

Proof. Let $p < q$ be two possible job sizes. Let $[i_p, m]$ be the machines on which a job of size p would run less than Opt time units. That is, $p/s_k \leq \mathsf{Opt}$ for $k \in [i_p, m]$. Similarly define $[i_q, m]$. Note that $i_p \leq i_q$. If a job of size q was placed on a machine i during the saturation phase, then this machine has load less than $\alpha\mathsf{Opt}s_i$. By definition of the algorithm, a job of size p would also be assigned during the saturation phase to a machine no faster than machine i, since machine i's load is less than $\alpha\mathsf{Opt}s_i$, and p can run on machine i. If instead a job of size q was placed on machine i during the slow fit phase, then all machines in the range $[i_q, m]$ have load at least $\alpha\mathsf{Opt}$. Thus a job of size p could either be placed on a machine slower than i_q during the saturation phase, or on a machine no faster than i in the slow fit phase by definition of the algorithm. ◀

We now turn to analyzing IDF's competitiveness. For notational compactness, we will now starting using A to denote the algorithm IDF. We will show that the following statements hold by induction on epochs:

$\mathcal{A}(i,k)$ is the statement $A_i(k) \leq A_i^*(k) + c\mathsf{Opt}S_i$

and

$\mathcal{B}(i,k)$ is the statement $B_i(k) \leq B_i^*(k) + c\mathsf{Opt}S_i$

where

- $S_i = \sum_{k=i}^{m} s_k$ is the total speed of machines $[i,m]$.
- $A_i(k)$ is the total load on machines $[i,m]$ under the algorithm A just before jobs are assigned at the start of epoch k.
- $B_i(k)$ is the total load on machines $[i,m]$ under the algorithm A just after all jobs are assigned by the algorithm at the start of epoch k.
- Define *restricted opt* to be the optimum under the restriction that jobs can only be assigned to machines at the start of the epoch. Note that by making ϵ sufficiently small, this is does not change the optimal solution.
- $A_i^*(k)$ is the total load on machines $[i,m]$ for the restricted opt just before jobs are assigned by restricted opt at the start of epoch k.
- $B_i^*(k)$ is the total load on machines $[i,m]$ for the restricted opt just after all jobs are assigned by restricted opt at the start of epoch k.

In order for our induction to go through, we will need the various parameters to satisfy the following inequalities:

- $\alpha \geq \beta - c + 1$
- $c \geq \alpha + 1$
- $1 + c + \epsilon \leq \beta$

We observe in Lemma 4 that these inductive statements imply that IDF/A is $\beta + 1$ competitive. We then show in Lemma 5 that $\mathcal{B}(i,k)$ implies $\mathcal{A}(i,k+\epsilon)$. We then complete the inductive proof by showing in Lemma 6 that $\mathcal{A}(i,k)$ implies $\mathcal{B}(i,k)$.

▶ **Lemma 4.** *If* $\forall i \forall k\, [\mathcal{A}(i,k)$ *and* $\mathcal{B}(i,k)]$ *then* A *is* $(\beta+1)$*-competitive.*

▶ **Lemma 5.** $\forall i\ \ \mathcal{B}(i,k)$ *implies* $\forall i\ \ \mathcal{A}(i,k+\epsilon)$.

Proof. The proof is by reverse induction on i. For a base case, $i = m+1$, the claim is vacuously true. For the inductive case, assume that $\mathcal{A}(i+1,k+\epsilon)$ holds and our goal is to prove $\mathcal{A}(i,k+\epsilon)$.

For the first case, say that machine i at epoch $k+\epsilon$ has at most $c\mathsf{Opt}s_i$ work assigned to it. This implies that $A_i(k+\epsilon) \leq A_{i+1}(k+\epsilon) + c\mathsf{Opt}s_i$. Knowing that $A_{i+1}(k+\epsilon) \leq A_{i+1}^*(k+\epsilon) + c\mathsf{Opt}S_{i+1}$ is true (that is, $\mathcal{A}(i+1,k+\epsilon)$ holds), we have the following.

$$\begin{aligned} A_i(k+\epsilon) &\leq A_{i+1}(k+\epsilon) + c\mathsf{Opt}s_i \\ &\leq A_{i+1}^*(k+\epsilon) + c\mathsf{Opt}S_{i+1} + c\mathsf{Opt}s_i \\ &= A_{i+1}^*(k+\epsilon) + c\mathsf{Opt}S_i \end{aligned}$$

For the second case, machine i at epoch $k+\epsilon$ has strictly more than $c\mathsf{Opt}s_i$ assigned to it. Let a denote the last job assigned to machine i. We know that $p_a/s_i \leq \mathsf{Opt}$ by definition of the algorithm. Knowing this, it must be the case that machine i was loaded to more than

$(c-1)s_i\mathsf{Opt}$ when job a was assigned. Knowing that $c-1 \geq \alpha$ it is the case that job a was assigned by the slow-fit phase of the algorithm. Also we know that machine i is not ready to process job a at epoch $k+\epsilon$ since it hasn't completed all jobs assigned to it that have arrived before job a since at the epoch machine i has a strictly positive load excluding the last job a assigned to it. Since a was assigned by the slow-fit phase, when job a arrived, it must be the case that all machines i, $i+1$, ... ,m have load at least $\alpha\mathsf{Opt}$. This implies that at epoch $k+\epsilon$, all machines i, $i+1$, ... ,m have strictly positive loads.

Thus, we now know that $m, m-1, \ldots, i$ are busy processing some job between epoch k and $k+\epsilon$. We know that $B_i(k) \leq B_i^*(k) + c\mathsf{Opt}S_i$ since $\mathcal{B}(i,k)$ holds. We further know that $B_i^*(k)$ can decrease by at most ϵS_i to get $A_i^*(k+\epsilon)$ as this is the most work Opt can process on machines $m, m-1, \ldots, i$ between epoch k and $k+\epsilon$. The above argument implies that $B_i(k)$ decreases by ϵS_i since all machines i or greater are processing jobs during $[k, k+\epsilon]$. Thus, in the inequality $B_i(k) \leq B_i^*(k) + c\mathsf{Opt}S_i$ the left hand side decreases by at least as much as the right, giving the lemma. ◀

▶ **Lemma 6.** *$\forall i \mathcal{A}(i,k)$ implies $\forall i \mathcal{B}(i,k)$.*

Proof. Assume that a job j arrives in epoch $k-1$. By assumption, only one job arrives in epoch $k-1$. The proof is first by induction on k, and then by reverse induction on i, where i is the machine to which job j is assigned. We handle at the end the case where job j cannot be assigned and the algorithm declares failure.

We consider two cases. In the first case, assume that the load on machine i for the algorithm after jobs have been assigned at the start of epoch k is at most $c\mathsf{Opt}s_i$. Then in this case we know by induction that $B_{i+1}(k) \leq B_{i+1}^*(k) + c\mathsf{Opt}S_{i+1}$. Thus using the assumption that the load on machine i is at most $c\mathsf{Opt}s_i$, we know that

$$B_i(k) \leq B_{i+1}^*(k) + c\mathsf{Opt}S_{i+1} + c\mathsf{Opt}s_i \leq B_i^*(k) + c\mathsf{Opt}S_i$$

Now consider the case that the load on machine i is strictly more than $c\mathsf{Opt}s_i$. Thus we know that the last job put on machine i by the algorithm at the start of epoch k was assigned in the slow fit phase since $c \geq \alpha+1$. If $p_j/s_{i-1} > \mathsf{Opt}$ or $i=1$, then optimal cannot run j on a slower machine than i, and thus $\mathcal{A}(i,k)$ implies $\mathcal{B}(i,k)$ as $B_i(k) - A_i(k)$ and $B_i^*(k) - A_i^*(k)$ both increase by p_j. Otherwise let h be minimal such that all machines in the range $[h, i-1]$ have load at least $\beta\mathsf{Opt}$. Then we know that either $h=1$ or $p_j/s_{h-1} > \mathsf{Opt}$, otherwise the algorithm would have put job j on machine $h-1$. In either case, optimal cannot put job j on a machine with index $\leq h-1$. Thus $\mathcal{A}(h,k)$ implies $\mathcal{B}(h,k)$ as $B_h(k) - A_h(k)$ and $B_h^*(k) - A_h^*(k)$ both increase by p_j.

Now consider what happens to $\mathcal{B}(g,k)$ as g increases from h to i. Assume $g \in (h,i]$. Then $B_{g-1}(k) - B_g(k) \geq \beta\mathsf{Opt}s_{g-1}$. So intuitively $B_g(k)$ decreases at a rate of at least β. Also $B_{g-1}^*(k) - B_g^*(k) \leq (1+\epsilon)\mathsf{Opt}s_{g-1}$, otherwise the load on machine $g-1$ for optimal would be greater than $(1+\epsilon)\mathsf{Opt}s_{g-1}$, contradicting the definition of Opt. Thus intuitively, $B_g^*(k)$ decreases at a rate of at most $\epsilon+1$. Also $c\mathsf{Opt}S_{g-1} - c\mathsf{Opt}S_g = c\mathsf{Opt}s_{g-1}$. So intuitively this term decreases at a rate of exactly c. Thus using the fact that $1+c+\epsilon \leq \beta$, $\mathcal{B}(h,k)$ implies $\mathcal{B}(i,k)$.

Now consider the case that the algorithm couldn't assign job j. Then machine m has load at least $\beta\mathsf{Opt}s_m$. Let h be minimal such that all machines in the range $[h,m]$ have load at least $\beta\mathsf{Opt}$. Then we know that either $h=1$ or $p_j/s_{h-1} > \mathsf{Opt}$, otherwise the algorithm would have put job j on machine $h-1$. In either case, optimal cannot put job j on a machine with index $\leq h-1$. Thus $\mathcal{A}(h,k)$ implies $\mathcal{B}(h,k)$ as $B_h(k) - A_h(k)$ and $B_h^*(k) - A_h^*(k)$ both increase by p_j. Now we just repeat the argument in the last paragraph to prove that

$B_m(k) \leq B^*_m(k) + cs_m\mathsf{Opt}$. Since $B_m(k) \geq \beta s_m \mathsf{Opt}$, we have $B^*_m(k) \geq (\beta - c)s_m\mathsf{Opt}$, which is a contradiction to Opt if $\beta - c > \epsilon + 1$. ◀

▶ **Lemma 7.** *One can verify that $\alpha = 2$, $c = 3$ and $\beta = 4$ satisfies the stated inequalities when $\epsilon = 0$, and thus IDF with these parameters is 5-competitive, assuming that its estimate of Opt is correct.*

Now consider the case that the algorithm does not know Opt. It is easy to see that the whole analysis goes through as long as our estimate of Opt is no smaller than the actual Opt. If our algorithm fails to assign a job, the algorithm sets its new estimate of optimal, Opt', to be $\mathsf{Opt}(\beta + 1)/\alpha$. Then we know that $A_i \leq \alpha S_i \mathsf{Opt}'$ since $A_i \leq (\beta + 1)S_i\mathsf{Opt}$. Lemma 6 still goes through since all machines have load at most $\alpha\mathsf{Opt}'$. More precisely, the proof of Lemma 6 does not need to appeal to the slow fit phase to prove the invariants since no machine is currently saturated. Since our estimate of Opt can be at most $(\beta + 1)/\alpha$ larger than the true Opt, we derive a competitive ratio of $(\beta + 1)^2/\alpha$, which is $25/2$ for the above choice of α and β.

Acknowledgements. We thank Amos Fiat for introducing us to this problem in a talk [11] under the auspices of the Algorithms and Uncertainty program at the Simons Institute for the Theory of Computing at the University of California at Berkeley, and for several subsequent helpful discussions.

References

1 Fidaa Abed, José R. Correa, and Chien-Chung Huang. Optimal coordination mechanisms for multi-job scheduling games. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 13–24, 2014.

2 S. Anand, Karl Bringmann, Tobias Friedrich, Naveen Garg, and Amit Kumar. Minimizing maximum (weighted) flow-time on related and unrelated machines. *Algorithmica*, 77(2):515–536, 2017.

3 James Aspnes, Yossi Azar, Amos Fiat, Serge Plotkin, and Orli Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *Journal of the ACM*, 44(3), May 1997.

4 Yossi Azar, Bala Kalyanasundaram, Serge A. Plotkin, Kirk Pruhs, and Orli Waarts. On-line load balancing of temporary tasks. *Journal of Algorithms*, 22(1):93–110, 1997.

5 Nikhil Bansal and Bouke Cloostermans. Minimizing maximum flow-time on related machines. *Theory of Computing*, 12(1):1–14, 2016.

6 Julian Birkinshaw. Strategies for managing internal competition. *California Management Review*, 44(1):21–38, 2001.

7 George Christodoulou, Elias Koutsoupias, and Akash Nanavati. Coordination mechanisms. In *International Colloquium on Automata, Languages and Programming*, pages 345–357, 2004.

8 Ilan Reuven Cohen, Alon Eden, Amos Fiat, and Lukasz Jez. Pricing online decisions: Beyond auctions. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 73–91, 2015.

9 Shelley DuBois. Internal competition at work: Worth the trouble? *Fortune*, February 25 2012.

10 Michal Feldman, Amos Fiat, and Alan Roytman. Makespan minimization via posted prices. unpublished, 2017.

11 Amos Fiat. Makespan minimization via posted prices, November 3 2016. talk given under the auspices of the Algorithms and Uncertainty program at the Simons Institute for the Theory of Computing at the University of California at Berkeley.

12 Sungjin Im, Benjamin Moseley, and Kirk Pruhs. A tutorial on amortized local competitiveness in online scheduling. *SIGACT News*, 42(2):83–97, 2011.

13 Nicole Immorlica, Erran L. Li, Vahab S. Mirrokni, and Andreas S. Schulz. Coordination mechanisms for selfish scheduling. In *International Workshop on Internet and Network Economics*, pages 55–69, 2005.

14 Noam Nisan and Amir Ronen. Algorithmic mechanism design (extended abstract). In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*, pages 129–140, 1999.

15 Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay Vazirani. *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA, 2007.

16 Kirk Pruhs, Jirí Sgall, and Eric Torng. Online scheduling. In *Handbook of Scheduling - Algorithms, Models, and Performance Analysis.* 2004.

Finding Axis-Parallel Rectangles of Fixed Perimeter or Area Containing the Largest Number of Points*

Haim Kaplan[1], Sasanka Roy[2], and Micha Sharir[3]

1 **Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
haimk@tau.ac.il
2 **Indian Statistical Institute, Kolkata, India**
sasanka.ro@gmail.com
3 **Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
michas@tau.ac.il

Abstract

Let P be a set of n points in the plane in general position, and consider the problem of finding an axis-parallel rectangle with a given perimeter, or area, or diagonal, that encloses the maximum number of points of P. We present an exact algorithm that finds such a rectangle in $O(n^{5/2}\log n)$ time, and, for the case of a fixed perimeter or diagonal, we also obtain (i) an improved exact algorithm that runs in $O(nk^{3/2}\log k)$ time, and (ii) an approximation algorithm that finds, in $O\left(n+\frac{n}{k\varepsilon^5}\log^{5/2}\frac{n}{k}\log\left(\frac{1}{\varepsilon}\log\frac{n}{k}\right)\right)$ time, a rectangle of the given perimeter or diagonal that contains at least $(1-\varepsilon)k$ points of P, where k is the optimum value.

We then show how to turn this algorithm into one that finds, for a given k, an axis-parallel rectangle of smallest perimeter (or area, or diagonal) that contains k points of P. We obtain the first subcubic algorithms for these problems, significantly improving the current state of the art.

1998 ACM Subject Classification F.2.2 Geometrical problems and computations, E.1 Data structures

Keywords and phrases Computational geometry, geometric optimization, rectangles, perimeter, area

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.52

1 Introduction

In the basic problem studied in this paper, we are given a set P of n points in the plane in general position, and a fixed parameter $\tau > 0$, and we seek an axis-parallel rectangle of perimeter 2τ that encloses the maximum number of points of P. We denote this problem as MAX-PTS(τ). We also consider variants of the problem, involving rectangles with other fixed parameters, such as the area or the length of the diagonal. Such problems have been studied

* Work by Haim Kaplan has been supported by Grant 1161/2011 from the German-Israeli Science Foundation, by Grant 1841-14 from the Israel Science Foundation, and by the Israeli Centers for Research Excellence (I-CORE) program (center no. 4/11). Work by Micha Sharir has been supported by Grant 2012/229 from the U.S.-Israel Binational Science Foundation, by Grant 892/13 from the Israel Science Foundation, by the Israeli Centers for Research Excellence (I-CORE) program (center no. 4/11), by the Blavatnik Research Fund in Computer Science at Tel Aviv University, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

by many researchers (as we survey below) and arise in statistical clustering and pattern recognition (see [1] and the references therein).

Earlier works on these problems have mainly studied the "dual" version, where we specify k and seek an axis-parallel rectangle of minimum perimeter (or area, or diagonal) that encloses k points of P. This so-called MIN-PERIM(k) (or MIN-AREA(k), or MIN-DIAG(k)) problem has been studied in Aggarwal *et al.* [1], who gave an $O(nk^2 \log n)$-time solution. Eppstein and Erickson [7] gave an algorithm that runs in $O(n \log n + nk^2)$ time and requires $O(n \log n + nk)$ storage. They observed that the k points in the optimal rectangle are among the $O(k)$ rectilinear nearest neighbors of each other. This allowed them to partition the problem into $O(n/k)$ subproblems, each on a subset of $O(k)$ points, and to apply a naive $O(k^3)$-algorithm of Aggarwal *et al.* [1] to each subset. The partition is obtained using a data structure for planar rectilinear nearest neighbors.

Datta *et al.* [5] suggested a different scheme to break the problem into $O(n/k)$ subproblems, each of size $O(k)$. Their algorithm runs within the same time bound as the algorithm of Eppstein and Erickson but requires only linear storage.

Segal and Kedem [15] gave algorithms for MIN-PERIM(k) and MIN-AREA(k), that are linear for very large values of k. Their algorithms run in $O(n + k(n-k)^2)$ time.

A very recent work on this problem, by De Berg *et al.* [6], develops a near linear algorithm for MIN-AREA(k), for small values of k. Their algorithm runs in $O(nk^2 \log n + n \log^2 n)$ time. They also give a $(1-\varepsilon)$-approximation algorithm for the dual MIN-PTS(A) problem, which asks for an axis-parallel rectangle of a given area A that contains the largest number of points of P. Notice that all algorithms for MIN-PERIM(k) and MIN-AREA(k) are cubic in the worst-case for some values of k.

As noted in [6], the variants of the problem involving area are harder. For example, it is no longer the case that the k points in an optimal rectangle are among the $O(k)$ rectilinear nearest neighbors of each other. Our paper too, which handles all three variants, derives faster algorithms for the cases of perimeter or diagonal, and the approximation algorithms that we obtain apply only for these two cases.

Let us return to the case of perimeter. Using an algorithm for MIN-PERIM(k), one can solve the original problem, that we have denoted as MAX-PTS(τ), using binary search in a straightforward manner. The overall cost of this algorithm is $O(\log n)$ times the cost of MIN-PERIM(k).

The converse direction, to turn a given algorithm for MAX-PTS(τ) into an efficient solution of MIN-PERIM(k), is somewhat more involved, but is doable. Indeed, in this paper we first solve MAX-PTS(τ) directly, and then show how to solve MIN-PERIM(k) by a logarithmic number of calls to MAX-PTS(τ).

1.1 Our results

We first present, in Section 2, an algorithm for MAX-PTS(τ) that runs in $O(n^{5/2} \log n)$ time. The method is sufficiently general, so the algorithm also solves the variants where the area or the diagonal of the rectangle are fixed (and we want to maximize the number of points of P that it contains), within the same running time bound.

We then use, in Section 2.3, a simple grid-based construction that allows us to solve MAX-PTS(τ) in an output-sensitive manner. Specifically, the running time improves to $O(nk^{3/2} \log k)$, where k is the output size, the maximum number of points of P contained in such a rectangle. A simple modification of the same approach yields the same improvement for the case of fixed diagonal, but, unfortunately, not for the case of fixed area.

We also obtain, in Section 3, an approximation algorithm that finds, in near-linear time (see Theorem 3 for the precise bound), an axis-parallel rectangle of the given perimeter that contains at least $(1-\varepsilon)k$ points of P, for a prespecified error parameter ε. The approximation algorithm extends to the case of diagonal but not to the case of area.

Finally, we consider the dual problem MIN-PERIM(k), as defined above, and present an algorithm that solves it in $O(nk^{3/2}\log k\log n)$ time, which is only $O(\log n)$ times slower than the running time of our algorithm for MAX-PTS(τ). We obtain this result by reducing MIN-PERIM(k) to a logarithmic number of calls to MAX-PTS(τ). This bound improves (for almost all values of k) the previous best bound $O(n\log n+nk^2)$ of [5, 7]. Our reduction is fairly general, and can also be applied to the cases of area or diagonal. For the case of area, our algorithm is not output-sensitive, and runs in $O(n^{5/2}\log^2 n)$ time. These are the first subcubic algorithms for these problems (for any value of k), improving upon [1, 6, 5, 7, 15]. For the case of area, the algorithm in [6] is faster when k is small; for the cases of perimeter or diagonal, our algorithms, as already noted, are significantly faster for almost all values of k.

2 An exact algorithm for max-pts(τ)

We recall our basic problem: Let P be a set of n points in the plane in general position (in particular, no two points of P have the same x- or y-coordinate), and let τ be a given positive real number. We want to find an axis-parallel rectangle R of perimeter 2τ that contains the largest number of points of P.

Let $\mathcal{Q}=\mathcal{Q}(\tau)$ denote the collection of all axis-parallel rectangles R of perimeter 2τ. Each rectangle $R\in\mathcal{Q}$ can be parameterized by three parameters (x,y,z), where (x,y) is the bottom-left vertex of R, and z is its width (x-span); its height (y-span) is then $\tau-z$. In other words, we identify the rectangles of $\mathcal{Q}$ with the points of $\mathbb{R}^2\times[0,\tau]$.

A point $p=(p_1,p_2)$ lies in a rectangle $R\in\mathcal{Q}$, parameterized by (x,y,z), if and only if

$$x\le p_1\le x+z \qquad \text{and} \qquad y\le p_2\le y+\tau-z. \tag{1}$$

For each $p=(p_1,p_2)\in P$, let K_p denote the set of all rectangles in $\mathcal{Q}$ that contain p. In the parametric 3-space, K_p is a tetrahedron, bounded by the four halfspaces specified in (1). Our problem is now reduced to that of finding a point of maximum *depth* in the arrangement of these n isothetic tetrahedra (i.e., a point contained in the largest possible number of tetrahedra; note that these tetrahedra are indeed translates of one another).

The cross-section of a tetrahedron K_p, at any fixed z, is an axis-parallel rectangle $K_p(z)$, given by

$$p_1-z\le x\le p_1 \qquad \text{and} \qquad p_2-\tau+z\le y\le p_2. \tag{2}$$

All the rectangles $K_p(z)$, for $p\in P$, are translates of one another; they are in fact the sets $p-R_0(z)$, for $p\in P$, where $R_0(z)=[0,z]\times[0,\tau-z]$. Let $\mathcal{R}(z)$ denote the collection of these rectangles, for any $z\in[0,\tau]$, and let $\mathcal{A}(z)$ denote the planar arrangement of the rectangles of $\mathcal{R}(z)$. Our problem now is to find a z at which the maximum depth $\Delta(z)$ of a point in $\mathcal{A}(z)$ is maximized.

As z varies from 0 to τ, the rectangles of $\mathcal{R}(z)$ simultaneously deform, as their common width (x-span) increases and their common height (y-span) decreases. The arrangement $\mathcal{A}(z)$ varies continuously, but its combinatorial structure remains unchanged as long as both the left-to-right order of the y-vertical edges of the rectangles, and the bottom-to-top order of the x-horizontal edges of the rectangles, remain unchanged. Under the general position

assumption, no pair of left edges, of right edges, of top edges, or of bottom edges, can ever attain the same x- or y-coordinate. Hence, the critical values of z at which the combinatorial structure of $\mathcal{A}(z)$ changes are those at which either the lines supporting the left side of one rectangle and the right side of another coincide, or the lines supporting the bottom side of one rectangle and the top side of another coincide. The set of critical values is therefore

$$\Big\{p_1 - q_1,\ p_2 - q_2 + \tau \mid p = (p_1, p_2) \neq q = (q_1, q_2) \in P\Big\} \cap (0, \tau).$$

There are at most $n(n-1)$ such critical values (for each pair p, q, at most one of $p_1 - q_1$, $q_1 - p_1$ can lie in $(0, \tau)$, and the same holds for $p_2 - q_2 + \tau$, $q_2 - p_2 + \tau$), and we may assume them to be all distinct, by our general position assumption. At each such critical value, either the left edge of one rectangle and the right edge of another rectangle in $\mathcal{R}(z)$, that are adjacent in the left-to-right order of the vertical edges, are swapped in this order, or the bottom edge of one rectangle and the top edge of another rectangle, adjacent in the bottom-to-top order of the horizontal edges, are swapped.

In what follows, we present a data structure that maintains the arrangement $\mathcal{A}(z)$, updates it at each critical value of z, and keeps track of the maximum depth in $\mathcal{A}(z)$ after each update. The data structure requires $O\left(n^{3/2}\log n\right)$ storage, can be initialized in $O\left(n^{3/2}\log n\right)$ time, and each update takes $O\left(n^{1/2}\log n\right)$ amortized time. Using this structure, we obtain our exact algorithm.

▶ **Theorem 1.** *Given a set P of n points in the plane in general position, and a parameter $\tau > 0$, one can find, in $O\left(n^{5/2}\log n\right)$ time, an axis-parallel rectangle of perimeter 2τ that contains the maximum number of points of P. The algorithm requires $O\left(n^{3/2}\log n\right)$ storage.*

Other fixed parameters. As already noted, the approach described so far can also handle, with minor modifications, other fixed parameters, such as the area, or the length of the diagonal, or any parameter that depends on the lengths of the edges of the rectangles, so that the length of one edge uniquely determines the length of the other edge (all the parameters mentioned so far have this property). For example, in the case where our rectangles have a fixed area A, we have to replace (1) and (2) by

$$x \le p_1 \le x+z, \qquad y \le p_2 \le y+A/z, \qquad \text{and} \qquad p_1-z \le x \le p_1, \qquad p_2-A/z \le y \le p_2,$$

respectively. The latter pair of inequalities define K_p, so it is no longer a simplex. Nevertheless, the cross sections $K_p(z)$, for any $z > 0$, are all isothetic rectangles, and the critical values of z are similarly constructed – they are now of the form $|p_1 - q_1|$ or $A/|p_2 - q_2|$. This allows the rest of the analysis to proceed more or less verbatim. The case of fixed diagonal is also handled in a fully analogous manner. That is, these variants can also be solved in $O(n^{5/2}\log n)$ time.

2.1 The data structure

For any fixed value of z, one can compute the maximum depth of $\mathcal{A}(z)$ in $O(n\log n)$ time and $O(n)$ space [12, 14]. If we allow $O(n\log n)$ space (and $O(n\log n)$ time) then we can perform this computation in a straightforward manner by sweeping $\mathcal{R}(z)$ from left to right by a vertical line, while maintaining the cross sections of the rectangles in $\mathcal{R}(z)$ with the sweepline in a dynamic segment tree T [3]. To efficiently recompute the maximum depth after each update of T, we also store at each node of T the maximum depth of a leaf in its subtree. (The *depth* of a leaf is the number of rectangles containing the vertical interval

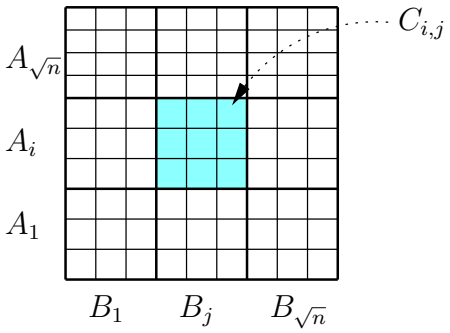


Figure 1 The grid $G(z)$, the coarser grid, its horizontal and vertical slabs, and a single highlighted cell.

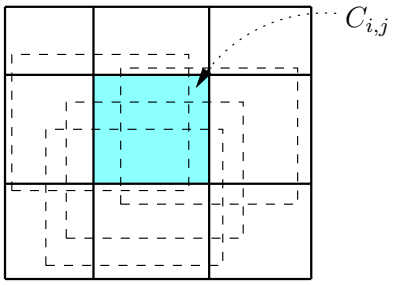


Figure 2 The sets $H_{i,j}$ and $V_{i,j}$, and the respective sets $\mathcal{R}^H_{i,j}$, $\mathcal{R}^V_{i,j}$ of the rectangles forming them, within a single highlighted cell $C_{i,j}$.

that it represents within the sweepline.) An update affects these counters only at the nodes on the two paths to the endpoints of the inserted or deleted segment, and it is therefore straightforward to update these counters when we insert or delete a segment to/from the segment tree, and to propagate them to the root, whose counter represents maximum the depth that we seek.

One could attempt at making this (static) structure dynamic, under swaps of vertical or horizontal rectangle edges, by maintaining all the versions of T, constructed during the sweep, in some persistent data structure, and by updating that structure at each swap of edges. Unfortunately, it is not clear how to perform such updates efficiently. (This is because a swap of two horizontal edges might affect arbitrarily many versions of T; vertical swaps, in contrast, affect only two consecutive versions.) Instead, we present a slower, more symmetric data structure for computing the maximum depth of $\mathcal{A}(z)$, which can be made dynamic at a reasonably low cost.

The grid. Consider the grid $G = G(z)$ formed by the horizontal and vertical lines supporting the edges of the rectangles in $\mathcal{R} = \mathcal{R}(z)$. (The other notations introduced below also depend on z, but we make this dependence implicit from now on, to simplify the notation.) Assuming that z is not critical, G is formed by $2n$ vertical lines and by $2n$ horizontal lines.

Partition G into $\sqrt{n} \times \sqrt{n}$ cells, by the $\sqrt{n} - 1$ horizontal lines and the $\sqrt{n} - 1$ vertical lines whose indices are multiples of $2\sqrt{n}$, referred to as *dividers*; the number of cells in the resulting coarser grid is n. (We ignore in what follows the insignificant rounding issues.) The horizontal dividers partition the plane into $\sqrt{n}$ horizontal slabs, denoted $A_1, \ldots, A_{\sqrt{n}}$ in this bottom-to-top order, and the vertical dividers partition the plane into $\sqrt{n}$ vertical slabs, denoted $B_1, \ldots, B_{\sqrt{n}}$ in this left-to-right order. See Figure 1. We regard each vertical (resp., horizontal) slab as closed at its left (resp., bottom) boundary, and open at its right (resp., top) boundary, so that each vertical (resp., horizontal) line is contained in exactly one vertical (resp., horizontal) slab. Accordingly, each cell in the coarser grid is closed at its left and bottom boundaries, and open at its right and top boundaries, making the cells pairwise disjoint and covering all rectangles.

For each cell $C_{i,j} = A_i \cap B_j$ of the coarser grid, let $v_{i,j}$ denote the number of corners of the rectangles in $\mathcal{R}$ that lie in $C_{i,j}$. We have $\sum_{i,j} v_{i,j} = 4n$, so, on average, each cell contains four vertices, but some cells might contain many more vertices. Still, by construction, no cell contains more than $4\sqrt{n}$ vertices. In fact, we have the stronger property that no vertical or horizontal slab contains more than $4\sqrt{n}$ vertices (At most *two* on each vertical line in a vertical slab, and similarly for horizontal slabs).

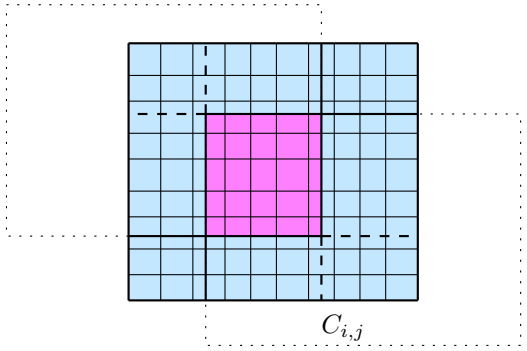


Figure 3 Two short rectangles in $C_{i,j}$, and the partition into query subcells that they induce.

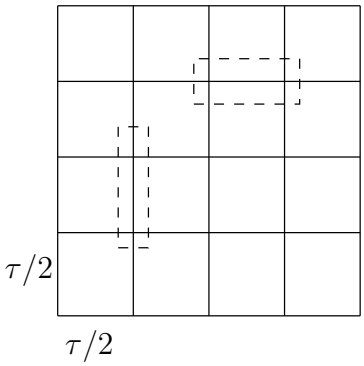


Figure 4 A rectangle R of perimeter 2τ can intersect at most six grid cells.

Computing the depth within a cell: Rectangles fully containing the cell. Consider the problem of computing for each such cell $C_{i,j}$, a counter $w_{i,j}$, equal to the number of rectangles of $\mathcal{R}$ that fully contain $C_{i,j}$. We compute the $w_{i,j}$'s by a sweep over the vertical slabs, maintaining a dynamic segment tree (see [3]) over the horizontal slabs, or rather cells, within the currently swept vertical slab. When moving from one vertical slab to the next, we find the $w_{i,j}$'s of all cells in the new slab, in the following three steps. First, we remove all the vertical segments that correspond to rectangles with a right edge in the current vertical slab (since these rectangles do not contribute to the $w_{i,j}$'s in this slab). Then, we traverse the segment tree bottom-up and compute, for each leaf, the number of segments (the y-spans of the active rectangles) containing the cell that corresponds to the leaf, thereby obtaining the desired counters $w_{i,j}$. We then add the segments that correspond to rectangles with a left edge in the current vertical slab (provided that their right edge is not in the slab). We have $O(n)$ insertions and deletions to/from this segment tree, each taking $O(\log n)$ amortized time. In addition, the number of nodes in the segment tree is $O(\sqrt{n})$ and we traverse it $\sqrt{n}$ times, to compute the counters $w_{i,j}$. It follows that the sweep takes $O(n \log n)$ total time.

Rectangles that straddle the cell. Let R be a rectangle that intersects $C_{i,j}$ (without fully containing it) and is not one of the $O(v_{i,j})$ rectangles that have a corner in $C_{i,j}$. Then R crosses $C_{i,j}$ in either a horizontal strip or a vertical strip; that is, $C_{i,j} \cap \partial R$ consists either of portions of one or two horizontal edges of R that cross $C_{i,j}$ from side to side, or of portions of one or two vertical edges that cross $C_{i,j}$ from side to side. Let $H_{i,j}$ (resp., $V_{i,j}$) denote the set of the horizontal (resp., vertical) edges of this kind, and let $\mathcal{R}^H_{i,j}$ (resp., $\mathcal{R}^V_{i,j}$) denote the set of the rectangles that contain the edges of $H_{i,j}$ (resp., of $V_{i,j}$). See Figure 2. The edges in $H_{i,j}$ partition $C_{i,j}$ into at most $2\sqrt{n}$ horizontal strips, and the depth within each strip, with respect to the rectangles in $\mathcal{R}^H_{i,j}$, is fixed, and changes by ± 1 as we move from one strip to the next. Analogous properties hold for the edges in $V_{i,j}$.

It follows that if we ignore the $O(v_{i,j})$ rectangles with vertices in $C_{i,j}$ (we refer to them as *short rectangles*) and the rectangles that fully contain $C_{i,j}$, the maximum depth in $C_{i,j}$ is the maximum depth $\delta^H_{i,j}$ of the rectangles of $\mathcal{R}^H_{i,j}$, plus the maximum depth $\delta^V_{i,j}$ of the rectangles of $\mathcal{R}^V_{i,j}$. Each of $\delta^H_{i,j}$, $\delta^V_{i,j}$ is the maximum of a sequence of depths, of length at most $2\sqrt{n}$, where consecutive elements differ by ± 1.

We maintain the vertical projections (i.e., y-spans) of the intersections of the rectangles of $\mathcal{R}^H_{i,j}$ with $C_{i,j}$ in a segment tree $T^H_{i,j}$ (which we will make dynamic in the next section).

Each leaf v of $T^H_{i,j}$ is associated with a strip $[h', h)$, where h' and h are consecutive edges of $H_{i,j}$. The depth of this strip (with respect to $\mathcal{R}^H_{i,j}$ is the sum of the numbers of segments stored at the ancestors of v).

We use this data structure to find, in $O(\log n)$ time, the strip of maximum depth in any subsequence of consecutive strips, by storing at each internal node v of $T^H_{i,j}$ the maximum depth of the leaves in its subtree (we omit the straightforward and fairly routine details of this mechanism).

We maintain $\mathcal{R}^V_{i,j}$ in an analogously defined dynamic segment tree $T^V_{i,j}$. Clearly, $T^H_{i,j}$ and $T^V_{i,j}$ allow us to answer maximum depth queries, with respect to $\mathcal{R}^H_{i,j} \cup \mathcal{R}^V_{i,j}$, where each query specifies a range of consecutive horizontal strips and a range of consecutive vertical strips, and asks for the maximum depth (with respect to $\mathcal{R}^H_{i,j} \cup \mathcal{R}^V_{i,j}$) within the rectangular Cartesian product of these ranges. Each such query takes $O(\log n)$ time.

Short rectangles and computing the maximum depth. We use the structures $T^H_{i,j}$ and $T^V_{i,j}$ to compute the real maximum depth within $C_{i,j}$, which also takes into account the $w_{i,j}$ rectangles that fully contain $C_{i,j}$, and the $O(v_{i,j})$ short rectangles. To do so, we partition $C_{i,j}$ into $O(v^2_{i,j})$ axis-parallel rectangular subcells by the horizontal and vertical lines that pass through the vertices inside $C_{i,j}$,[1] and query $T^H_{i,j}$ and $T^V_{i,j}$ for the maximum depth (in $\mathcal{R}^H_{i,j} \cup \mathcal{R}^V_{i,j}$) within each of the resulting subcells, to which we refer as *query subcells*. See Figure 3. (Note that, in general, the left and right boundary edges of each query subcell cross the interiors of two vertical strips stored in $T^V_{i,j}$, and the bottom and top boundary edges of each query subcell cross the interiors of two vertical strips stored in $T^H_{i,j}$. We expand the horizontal and the vertical ranges of the query subcell to fully include the four relevant strips.) Each query subcell, though, has an additional weight, equal to the number of short rectangles that fully contain the subcell (by construction, there is no partial overlap of any short rectangle with a query subcell). We refer to this additional weight of a query subcell as the *short weight* of the subcell. These short weights are easy to compute in $O(v^2_{i,j})$ time, by constructing the coarse grid of these subcells (within $C_{i,j}$), followed by a suitable traversal of the subcells, updating the count by 0, $+1$, or -1, as we pass from one subcell to the next. For each query subcell, we add this short weight, and the global counter $w_{i,j}$, to the depths returned by the queries to $T^H_{i,j}$ and $T^V_{i,j}$. We then output the maximum of the resulting depths, over all the $O(v^2_{i,j})$ query subcells.

Analysis. The running time of this algorithm is bounded as follows. We first sort the vertices of the rectangles by their x- and y-coordinates, and compute the global counters $w_{i,j}$. This initialization takes $O(n \log n)$ time. Then, for each cell $C_{i,j}$, we construct the trees $T^H_{i,j}$ and $T^V_{i,j}$, in $O(\sqrt{n})$ time (the relevant edges are already sorted), for an overall $O(n^{3/2})$ time. We then spend $O(v^2_{i,j})$ time for constructing the coarse grid of query subcells within $C_{i,j}$, and $O(v^2_{i,j} \log n)$ time for querying $T^H_{i,j}$ and $T^V_{i,j}$ with these subcells. Summing over all cells, and using the fact that $v_{i,j} \leq 4\sqrt{n}$ for each cell $C_{i,j}$, we get a total time of $O\left(n^{3/2} + \sqrt{n} \sum_{i,j} v_{i,j} \log n\right) = O\left(n^{3/2} \log n\right)$.

[1] When constructing this partition into query subcells, we also consider vertices on the bottom and left edges of $\partial C_{i,j}$, because, by our convention, these vertices are considered to be internal to the cell.

2.2 Updating the data structure

In this section we show how to dynamically maintain the counters $w_{i,j}$'s, the trees $T^H_{i,j}$ and $T^V_{i,j}$, the number of short rectangles in $C_{i,j}$ containing each query subcell, and the depth of each query subcell, as we increase z from 0 to τ. To retrieve the maximum depth after updating the structures at each critical z-value, we also maintain the depths of the query subcells, over all cells $C_{i,j}$, in a priority queue, and keep track of the maximum value in this queue. The overall maximum depth, i.e., the maximum number of points of P contained in an axis-parallel rectangle of perimeter 2τ, is the maximum attained by this priority queue throughout the z-sweep.

Note that at $z = 0$ each rectangle is essentially a vertical line segment (The lines supporting the left and right edges of the same rectangle are identical, and formally they are regarded as consecutive in G.). This simplifies that initialization of the data structure. For example initially $w_{i,j} = 0$ for all $1 \leq i, j \leq \sqrt{n}$.

As the value of z increases, the coordinates of the vertices of the rectangles in $\mathcal{R}(z)$ vary continuously, and so do the coordinates of the vertical and horizontal supporting lines that form the grid $G(z)$. However, discrete changes in the structure of $G(z)$ occur only when two horizontal or two vertical sides of two distinct rectangles partially overlap, or, in the looser sense that we follow, when the lines supporting two such edges coincide. The maximum depth in $\mathcal{A}(z)$ can change only at these discrete events.

Consider an event where the right side of one rectangle R_1 and the left side of another rectangle R_2 swap their vertical order; that is, the two vertical lines supporting these edges in $G(z)$ coincide and then swap their order. The event where two horizontal sides partially overlap is handled in a fully symmetric fashion This swap takes place either within a single vertical slab B_j, or across the boundary ('divider') between two adjacent slabs. In total, this affects up to $4\sqrt{n}$ cells of G (within these slabs). We describe here the case in which the swap occurs within a single vertical slab B_j; the other case is handled in a similar manner, with a few minor modifications, and will appear in the full version of this paper.

Consider first the cells of B_j that do not contain the vertices of R_1 and of R_2. Within each such cell $C_{i,j}$, the effect of the swap is that the right endpoint of the horizontal segment corresponding to R_1 and the left endpoint of the horizontal segment corresponding to R_2 swap their order in $T^V_{i,j}$, assuming they both cross $C_{i,j}$. As a result, the depth of a single vertical strip with respect to $\mathcal{R}^V_{i,j}$ increases by 2. This is because the x-spans of the rectangles in $\mathcal{R}(z)$ increase as z increases; in the symmetric case of horizontal strips, the opposite holds – the depth of a single strip decreases by 2. (As already mentioned, this holds only for cells within the common y-range of R_1 and R_2; no change occurs in the other cells.) We update the corresponding segments in the tree $T^V_{i,j}$. This takes $O(\log n)$ amortized time per cell, and $O(\sqrt{n}\log n)$ time for all cells that are affected by the swap.

We now need to reapply the rectangular depth queries for the query subcells within each affected cell $C_{i,j}$, but we note that only $O(v_{i,j})$ of the queries can change their output – these are the queries whose subcells are crossed by the vertical strip that has changed its depth. We perform these queries, as in the static case, and update the depths of these query subcells in the global priority queue accordingly ($w_{i,j}$, and the short weights of the affected subcells, do not change).

Consider next the at most four cells $C_{i,j}$ that contain vertices of R_1 or of R_2 (that is, endpoints of the swapped vertical edges). Here the swap does not affect $T^V_{i,j}$, because only one (or none, when endpoints of both edges lie in $C_{i,j}$) of the swapped vertical edges belongs to this set. We split the rest of the description of the required updates according to whether $C_{i,j}$ contains vertices only of R_1 or only of R_2, or vertices of both R_1 and R_2.

Consider first the case where $C_{i,j}$ contains only vertices of R_1, either the top-right vertex, or the bottom-right vertex, or both; $C_{i,j}$ may also contain left vertices of R_1, but they have no effect on the update procedure. The case where $C_{i,j}$ contains only the top-left vertex or the bottom-left vertex of R_2, or both (and maybe also right vertices), is handled in a fully symmetric manner. Denote by λ_1^R (resp., λ_2^L) the vertical line supporting the right side of R_1 (resp., the left side of R_2); these are the lines that swap their left-to-right order. In this case the partition of $C_{i,j}$ into query subcells essentially does not change, except that one line in $V_{i,j}$, namely λ_2^L (it is in $V_{i,j}$ because the left vertices of R_2, which it supports, are not in $C_{i,j}$) moves from one column of query subcells (just to the right of λ_1^R) to another column (just on the left of that line). We query $T_{i,j}^H$ and $T_{i,j}^V$ to get the new depth of each of the query subcells to the left and to the right of λ_1^R, with respect to the rectangles in $\mathcal{R}_{i,j}^H \cup \mathcal{R}_{i,j}^V$, add to it $w_{i,j}$ and the short weight of the subcell, and update the depths of all these subcells in the priority queue.

If $C_{i,j}$ contains at least one vertex of R_1 and at least one vertex of R_2 then, in the grid defining the partition of $C_{i,j}$ into query subcells, the corresponding vertical lines λ_1^R and λ_2^L swap (with the former moving to the right of the latter). Consequently, the short weights of some of the subcells in the column bounded these two lines (which 'closes' at the swap and 're-opens' afterwards) increases by 2; the affected cells are those that lie in the overlap between the y-spans of R_1 and R_2 (as before, the corresponding short weights decrease by 2 in the symmetric case of a horizontal swap). We locally update these short weights and the depths of these subcells accordingly, and similarly update the priority queue.

In both cases, the number of affected query subcells of $C_{i,j}$ is only $O(v_{i,j})$, so the total amortized update time is $O(v_{i,j} \log n)$.

The overall amortized time spent on maximum depth queries in $T_{i,j}^H$ and in $T_{i,j}^V$, and on updates of short weights, is $O\left(\sum_i v_{i,j} \log n\right)$, where we sum over all cells $C_{i,j}$ in the single vertical column B_j. Fortunately, $\sum_i v_{i,j} \leq 4\sqrt{n}$, so the overall (amortized) cost of an update is $O\left(\sqrt{n} \log n\right)$.

As we mentioned the case where the swap occurs across a horizontal or a vertical divider between adjacent is similar and takes the same amortized time. This completes the description and analysis of the data structure, including both correctness and performance bounds, and justifies the bounds given in Theorem 1.

2.3 An output-sensitive algorithm

Let k be the maximum number of points of P in an axis-parallel rectangle of perimeter 2τ. In this section we show how to modify our algorithm so that it runs in $O(nk^{3/2} \log k)$ time. The same modification holds for the case of fixed diagonal, but not for fixed area.

We cover the plane by a grid whose cells are of size $\tau/2 \times \tau/2$, and count the number of points of P in each nonempty cell. Using the floor function and a universal hashing scheme (see, e.g., [4]), this takes $O(n)$ expected time. Let k_0 denote the maximum number of points in any grid cell. It follows that $k_0 \leq k \leq 6k_0$, where the left inequality follows since each grid cell has perimeter 2τ, and the right inequality follows since any rectangle R of perimeter 2τ (and in particular the optimal one) can intersect at most six grid cells as is easily checked; see Figure 4.

We collect, in $O(n)$ time, all the 2×3 and 3×2 clusters C of grid cells such that C contains at least k_0 points of P, and observe that the number of such clusters is $O(n/k_0) = O(n/k)$. We apply our algorithm to each cluster separately and return the rectangle containing the largest number of points in any of the clusters. We thus obtain the following theorem.

▶ **Theorem 2.** *Given a set P of n points in the plane in general position, and a parameter $\tau > 0$, let k denote the maximum number of points of P in an axis-parallel rectangle of perimeter 2τ. One can find, in $O((n/k)k^{5/2}\log k) = O(nk^{3/2}\log k)$ time, an axis-parallel rectangle of perimeter 2τ that contains the maximum number k of points of P. The algorithm requires $O(n + k^{3/2}\log k)$ storage.*

▶ Remark.

(1) The technique in this subsection also works for finding an axis-parallel rectangle with diagonal of length d containing the maximum number of points of P. We use a $(d/\sqrt{2}) \times (d/\sqrt{2})$ grid (each of whose cells has diagonal $= d$), argue that any axis-parallel rectangle of diagonal d is contained in some small local cluster of grid cells, and obtain, as above, an algorithm that runs in $O(nk^{3/2}\log k)$ time, where k is the maximum number of points in a rectangle of diagonal d.

(2) The technique in this subsection does not extend to the case of rectangles with a fixed area, since no single grid can localize every rectangle of area A within a small cluster of its cells. (In contrast, as already noted, the main algorithm, which runs in $O(n^{5/2}\log n)$ time, does extend to the case of fixed area.)

Nevertheless, for the case of a fixed area, say A, we can use a similar idea to get an algorithm whose running time depends (albeit rather weakly) on A, as follows. Assume that $P \subset [0,1]^2$, and that the given area is $A < 1$ (the case $A \geq 1$ is clearly trivial). Without loss of generality, it suffices to consider only rectangles of width between A and 1, with the corresponding height between 1 and A. We can partition the problem into $O(\log(1/A))$ subproblems, so that in each subproblem we only consider rectangles whose widths are between z_0 and $2z_0$, and heights between A/z_0 and $A/(2z_0)$, for some fixed z_0. To each subproblem we can apply a suitable variant of the preceding grid construction, and solve the subproblem in $O(nk^{3/2}\log k)$ time, for a total cost of $O(nk^{3/2}\log k\log(1/A))$ time. We leave it as an open problem to obtain an algorithm whose running time bound is k-sensitive and independent of A, in the style of Theorem 2.

3 An approximate solution

In this section we present a randomized algorithm that computes, with high probability, a rectangle of perimeter 2τ that contains at least $(1-\varepsilon)k$ points of P, for a prescribed $0 < \varepsilon < 1$, where k is the maximum possible value, and runs in time $O\left(n + \frac{n}{k\varepsilon^5}\log^{5/2}\frac{n}{k}\log\left(\frac{1}{\varepsilon}\log\frac{n}{k}\right)\right)$.

We use the grid partitioning of Section 2.3, and obtain (i) an approximation k_0 of k, up to a factor 6, and (ii) $O(n/k_0) = O(n/k)$ clusters of points, each of size $\Theta(k)$. We apply the following procedure to each cluster separately, and return the rectangle containing the largest number of points of P, among those output for each of the clusters.

So let C be a fixed "heavy" cluster of grid cells, and let $P_C = P \cap C$ denote the set of points of P in C (of size $\Theta(k)$). We take a random sample S of size $s = \Theta\left(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\right)$, for some $0 < \delta < 1$. For a suitable sufficiently large constant of proportionality, S is an $(\varepsilon/12)$-*approximation* of P for axis-parallel rectangular ranges, with probability at least $1-\delta$. That is, with probability $\geq 1-\delta$, we have, for each axis-parallel rectangle R,

$\left|\frac{|R\cap S|}{|S|}-\frac{|R\cap P_C|}{|P_C|}\right|\le\frac{\varepsilon}{12}$ (see, e.g., [11]).[2]

We now run the exact algorithm of Section 2.3 on S, and obtain an axis-parallel rectangle R_S (of perimeter 2τ) that contains the maximum number of points of S.

Correctness. Let R be an optimum rectangle (of perimeter 2τ) that contains k points of P, and let C be a cluster that fully contains R; by the arguments in Section 2.3, such a cluster always exists. Let S be the corresponding random sample of s points of P_C, and let R_S denote, as above, the axis-parallel rectangle of perimeter 2τ that contains the maximum number of points, denoted s^*, of S. Then, with probability $\ge 1-\delta$, we have $\left|\frac{|R\cap S|}{s}-\frac{k}{t}\right|\le\frac{\varepsilon}{12}$, where $t=|P_C|=\Theta(k)$. On the other hand,

$$\left|\frac{|R_S\cap S|}{s}-\frac{|R_S\cap P_C|}{t}\right|=\left|\frac{s^*}{s}-\frac{|R_S\cap P_C|}{t}\right|\le\frac{\varepsilon}{12},$$

$$\text{that is,}\quad |R_S\cap P_C|\ge\frac{s^*t}{s}-\frac{1}{12}\varepsilon t\ge\frac{|R\cap S|t}{s}-\frac{1}{12}\varepsilon t\ge\left(k-\frac{1}{12}\varepsilon t\right)-\frac{1}{12}\varepsilon t=k-\frac{1}{6}\varepsilon t.$$

Since $t\le 6k$, this is $\ge(1-\varepsilon)k$. That is, the procedure will find, with probability at least $1-\delta$, a rectangle that contains at least $(1-\varepsilon)k$ points of P.

Running time. Finding the heavy clusters C takes a total of $O(n)$ time, and the overall cost of drawing the random samples S also takes $O(n)$ time. We take $\delta=(k/n)^c$, for some suitably large exponent c, so as to guarantee correctness with high probability in all clusters. (There are only $O(n/k)$ clusters, so the probability to fail in at least one of them is at most $O((n/k)\cdot(k/n)^c)=O((k/n)^{c-1})$.) The cost of a single application of the exact algorithm is $O\left(\frac{1}{\varepsilon^5}\log^{5/2}\frac{n}{k}\log\left(\frac{1}{\varepsilon}\log\frac{n}{k}\right)\right)$. Since the number of heavy clusters is $O(n/k)$, we obtain a total running time of $O\left(n+\frac{n}{k\varepsilon^5}\log^{5/2}\frac{n}{k}\log\left(\frac{1}{\varepsilon}\log\frac{n}{k}\right)\right)$. Assuming that k is not too small and neglecting the logarithmic terms, we can even take $\varepsilon=1/k^{1/5}$, and obtain an algorithm whose running time is $\approx O(n)$, which returns, with high probability, an axis-parallel rectangle of perimeter 2τ that contains at least $k-O(k^{4/5})$ points of P.

The same technique applies with minor modifications also to the case of a fixed diagonal. We summarize the results of this section in the following theorem.

▶ **Theorem 3.** *Let P be a set of n points in the plane, and let $\tau>0$ and $\varepsilon\in(0,1)$ be given parameters. We can compute, in time $O\left(n+\frac{n}{k\varepsilon^5}\log^{5/2}\frac{n}{k}\log\left(\frac{1}{\varepsilon}\log\frac{n}{k}\right)\right)$, an axis-parallel rectangle of perimeter 2τ that contains at least $(1-\varepsilon)k$ points of P, where k is the maximum number of points contained in such a rectangle. Within the same time bound we can compute an axis-parallel rectangle of diagonal d that contains at least $(1-\varepsilon)k$ points of P, where k is the maximum number of points contained in such a rectangle.*

We note that for the case of a fixed area, an approximation algorithm, based on a totally different approach, was recently obtained by De Berg et al. [6]; it runs in $O((n/\varepsilon^4)\log^2 n\log(1/\varepsilon))$ time.

[2] Using discrepancy based methods one can find a smaller ε-approximation of size $O(\frac{1}{\varepsilon}\log^2(\frac{1}{\varepsilon}))$ [2]. However, such an ε-approximation is less efficient to compute (although polynomial).

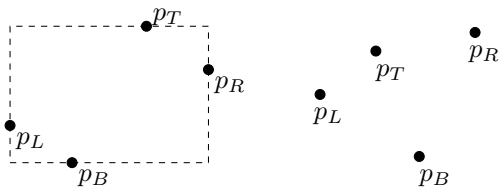


Figure 5 A quadruple (p_L, p_R, p_B, p_T) that defines a valid rectangle (left), and a quadruple that does not (right).

4 An efficient exact algorithm for min-perim(k)

In this section we present an efficient algorithm for the dual version of the problem, in which we specify k, and seek an axis-parallel rectangle of smallest perimeter that contains k points of P. The same technique also applies to the cases where the objective is to minimize the area, or the diagonal, of the enclosing rectangle. For the case of maximum diagonal, we obtain the same bound, and for the case of maximum area, we obtain a bound that is not output sensitive. In what follows we focus on the case of minimum perimeter, and only later discuss the extensions to the cases of minimum area or diagonal.

Let Q be an optimum rectangle, namely, an axis-parallel rectangle of smallest perimeter that contains k points of P. Clearly, each side of Q must contain a point of P, where these four points are not necessarily distinct (the number of distinct points is always between two and four). Denote by p_L, p_R, p_B, and p_T the points that lie on the left, right, bottom, and top sides of Q, respectively. Naively, there are $\binom{n}{2}$ candidate pairs (p_L, p_R), and $\binom{n}{2}$ candidate pairs (p_B, p_T). However, using an observation of [7], the number of candidate pairs is only $O(nk)$, because p_R is one of the $O(k)$ rectilinear nearest neighbors of p_L, and similarly for p_T and p_B. We find the $O(nk)$ left-right candidate pairs, and the $O(nk)$ bottom-top candidate pairs, in $O(n \log n + nk)$ time, as in [7]. We sort the ordered pairs (p_L, p_R) of distinct points of P, with p_L lying to the left of p_R, in increasing order of the differences between their x-coordinates, into a list X, and apply a symmetric construction with respect to the bottom-top pairs (p_B, p_T) and their y-coordinates, to obtain another sorted list Y.

Consider the matrix M whose rows are the elements of X (in sorted order) and whose columns are the elements of Y (in sorted order). For each pair of pairs $\pi_1 = (p_L, p_R)$ of X and $\pi_2 = (p_B, p_T)$ of Y, put $M(\pi_1, \pi_2) = (x(p_R) - x(p_L)) + (y(p_T) - y(p_B))$, and note that $M(\pi_1, \pi_2)$ is half the perimeter of the axis-parallel rectangle defined by the quadruple (p_L, p_R, p_B, p_T). To be precise, not every such quadruple defines a valid rectangle, but each valid candidate rectangle is defined by such a quadruple; See Figure 5.

The matrix M is a monotone matrix, that is, each of its rows and each of its columns is sorted in increasing order. Using the algorithm of Frederickson and Johnson [10] (see also [8, 9, 13]), We can find the ρ-th largest element in M in time $O(nk)$, for any rank ρ.

We thus run a binary search through the $O((nk)^2)$ critical perimeters (that is, entries of M), by making $O(\log n)$ calls to our algorithm MAX-PTS(τ), where the outcome of each call guides the continuation of the binary search. Each call incurs an overhead of $O(nk)$ time to find in M the relevant perimeter τ, and the algorithm itself takes time $O(nk_\tau^{3/2} \log k_\tau)$ where k_τ is the maximum number of points in a rectangle of perimeter 2τ (see Theorem 2). To make the overall running time bound k-sensitive, we pause the execution of the algorithm for MAX-PTS(τ) after the step where it obtains an approximation k_0 to k_τ, satisfying $k_0 \leq k_\tau \leq 6k_0$. If $k_0 > k$ we know that $k_\tau > k$, and we continue the binary search with a smaller τ. If $k > 6k_0$ we know that $k > k_\tau$, and we continue the binary search with a larger

τ. If $k_0 \leq k \leq 6k_0$, we let the algorithm run to completion, and bifurcate depending on the relation between the output k_τ and k. Altogether, we obtain the following result.

▶ **Theorem 4.** *Given a set P of n points in the plane, and a parameter $k \leq n$, we can find an axis-parallel rectangle of minimum perimeter that contains k points of P in time*

$$O\left(nk^{3/2} \log k \log n\right).$$

▶ Remark. The same procedure applies to the cases of minimum area or minimum diagonal. For the case of diagonal, we obtain the same performance bound. For the case of area, we have to put all $\binom{n}{2}$ pairs of points in X and Y, so selection in M takes $O(n^2)$ time, and the "decision procedure" MAX-PTS(A), where A is the given area, now takes $O(n^{5/2} \log n)$ time, resulting in a k-insensitive algorithm that runs in time $O(n^{5/2} \log^2 n)$. This is still a significant improvement over the recent algorithm of De Berg et al. [6] when k is not too small. It is an open problem to get an output sensitive bound, similar to the one in Theorem 4, for the case of area.

References

1. A. Aggarwal, H. Imai, N. Katoh, and S. Suri. Finding k points with minimum diameter and related problems. *J. Algorithms*, 12(1):38–56, 1991.
2. N. Bansal and S. Garg. Algorithmic discrepancy beyond partial coloring. In *Proc. of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 914–926, 2017.
3. Y-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9):1412–1434, 1992.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
5. A. Datta, H.P. Lenhof, C. Schwarz, and M. Smid. Static and dynamic algorithms for k-point clustering problems. *J. Algorithms*, 19(3):474–503, 1995.
6. M. de Berg, S. Cabello, O. Cheong, D. Eppstein, and C. Knauer. Covering many points with a small-area box. *CoRR*, abs/1612.02149, 2016.
7. D. Eppstein and J. Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discrete Comput. Geom.*, 11(3):321–350, 1994.
8. G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in X + Y and matrices with sorted columns. *J. Comput. Syst. Sci.*, 24(2):197–208, 1982.
9. G. N. Frederickson and D. B. Johnson. Finding kth paths and p-centers by generating and searching good data structures. *J. Algorithms*, 4(1):61–80, 1983.
10. G. N. Frederickson and D. B. Johnson. Generalized selection and ranking: Sorted matrices. *SIAM J. Comput.*, 13(1):14–30, 1984.
11. S. Har-peled. *Geometric Approximation Algorithms*. American Mathematical Society, Boston, MA, USA, 2011.
12. H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *J. Algorithms*, 4(4):310–323, 1983.
13. A. Mirzaian and E. Arjomandi. Selection in X + Y and matrices with sorted rows and columns. *Information Processing Letters*, 20(1):13–17, 1985.
14. S.C. Nandy and B.B. Bhattacharya. A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids. *Computers Math. Applic.*, 29(8):45–61, 1995.
15. M. Segal and K. Kedem. Enclosing k points in the smallest axis parallel rectangle. *Inform. Process. Letts.*, 65(2):95–99, 1998.

LZ-End Parsing in Linear Time

Dominik Kempa[1] **and Dmitry Kosolobov**[2]

1 **University of Helsinki, Helsinki, Finland**
`dominik.kempa@cs.helsinki.fi`
2 **University of Helsinki, Helsinki, Finland**
`dkosolobov@mail.ru`

Abstract

We present a deterministic algorithm that constructs in linear time and space the LZ-End parsing (a variation of LZ77) of a given string over an integer polynomially bounded alphabet.

1998 ACM Subject Classification E.4 Data compaction and compression

Keywords and phrases LZ-End, LZ77, construction algorithm, linear time

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.53

1 Introduction

Lempel–Ziv (LZ77) parsing [34] has been a cornerstone of data compression for the last 40 years. It lies at the heart of many common compressors such as `gzip`, `7-zip`, `rar`, and `lz4`. More recently LZ77 has crossed-over into the field of *compressed indexing of highly repetitive data* that aims to store repetitive databases (such as repositories of version control systems, Wikipedia databases [31], collections of genomes, logs, Web crawls [8], etc.) in small space while supporting fast substring retrieval and pattern matching queries [4, 7, 13, 14, 15, 23, 27]. For this kind of data, in practice, LZ77-based techniques are more efficient in terms of compression than the techniques used in the standard compressed indexes such as FM-index and compressed suffix array (see [24, 25]); moreover, often the space overhead of these standard indexes hidden in the $o(n)$ term, where n is the length of the uncompressed text, turns out to be too large for highly repetitive data [4].

One of the first and most successful indexes for highly repetitive data was proposed by Kreft and Navarro [24]. In its simplest form LZ77 greedily splits the input text into substrings (called *phrases*) such that each phrase is a first occurrence of a single letter or the longest substring that has an earlier occurrence. The index in [24] is built upon a small modification of LZ77 parsing called *LZ-End* (introduced in [22]) which assumes that the end of an earlier occurrence of each phrase aligns with the end of some previous phrase. This enables much faster retrieval of substrings of the compressed text without decompression.

While basic LZ77 parsing is solved optimally in many models [2, 9, 17, 18, 21, 26, 30], the construction of LZ-End remains a problem. Kreft and Navarro [24] presented an algorithm that constructs the LZ-End parsing of a string of length n in $O(n\ell(\log\sigma + \log\log n))$ time and $O(n)$ space, where ℓ is the length of the longest phrase in the parsing and σ is the alphabet size. They also presented a more space efficient version that works in $O(n\ell\log^{1+\epsilon} n)$ time and uses $O(n\log\sigma)$ *bits* of space, where ϵ is an arbitrary positive constant. This construction algorithm provides unsatisfactory time guarantees: it is quadratic in the worst case.

In [19] we described an algorithm that builds the LZ-End parsing of a read-only string of length n in $O(n\log\ell)$ *expected* time and $O(z+\ell)$ space, where z is the number of phrases and ℓ is the length of the longest phrase. In this paper we present an optimal-time deterministic

algorithm constructing the LZ-End parsing. We assume that the input string (of length n) is drawn from a polynomially bounded integer alphabet $\{0, 1, \ldots, n^{O(1)}\}$ and the computational model is the standard word RAM with $\Theta(\log n)$-bit machine words.

▶ **Theorem 1.** *The LZ-End parsing of a string of length n over the alphabet $\{0, 1, \ldots, n^{O(1)}\}$ can be computed in $O(n)$ time and space.*

The paper is organized as follows. In Section 2 we describe an algorithm that constructs the LZ-End parsing in $O(n \log\log n)$ time and linear space; we believe that this intermediate result is especially interesting for practice. In Section 3, we obtain an $O(n \log^2 n)$-time algorithm based on a completely different approach. Finally, we combine the two developed techniques in Section 4 and thus obtain a linear algorithm.

Preliminaries. Let s be a string of length $|s| = n$. We write $s[i]$ for the ith letter of s and $s[i..j]$ for $s[i]s[i+1]\cdots s[j]$. The *reversal of s* is the string $\overleftarrow{s} = s[n]\cdots s[2]s[1]$. A string u is a *substring* of s if $u = s[i..j]$ for some i and j; the pair (i, j) is not necessarily unique and we say that i specifies an *occurrence* of u in s. A substring $s[1..j]$ (resp., $s[i..n]$) is a *prefix* (resp. *suffix*) of s. For any i, j, the set $\{k \in \mathbb{Z} \colon i \le k \le j\}$ (possibly empty) is denoted by $[i..j]$. Our notation for arrays is similar: e.g., $a[i..j]$ denotes an array indexed by the numbers $[i..j]$.

Hereafter, s denotes the input string of length n over the integer alphabet $\{0, 1, \ldots, n^{O(1)}\}$. We extensively use a number of classical arrays built on the reversal $\overleftarrow{s}$ (the definitions slightly differ from the standard ones to avoid excessive mappings between the positions of s and $\overleftarrow{s}$): the *suffix array* $\mathsf{SA}[1..n]$ such that $\overleftarrow{s[1..\mathsf{SA}[1]]} < \overleftarrow{s[1..\mathsf{SA}[2]]} < \cdots < \overleftarrow{s[1..\mathsf{SA}[n]]}$ (lexicographically), the *inverse suffix array* $\mathsf{ISA}[1..n]$ such that $\mathsf{SA}[\mathsf{ISA}[i]] = i$ for $i \in [1..n]$, and the *longest common prefix (LCP) array* $\mathsf{LCP}[1..n-1]$ such that, for $i \in [1..n-1]$, $\mathsf{LCP}[i]$ is equal to the length of the longest common prefix of $\overleftarrow{s[1..\mathsf{SA}[i]]}$ and $\overleftarrow{s[1..\mathsf{SA}[i+1]]}$. We equip the array LCP with the *range minimum query (RMQ)* data structure [10] that, for any $i, j \in [1..n]$ such that $\mathsf{ISA}[i] < \mathsf{ISA}[j]$, allows us to compute in $O(1)$ time the value $\min\{\mathsf{LCP}[k] \colon \mathsf{ISA}[i] \le k < \mathsf{ISA}[j]\}$, which is equal to the length of the longest common suffix of $s[1..i]$ and $s[1..j]$. For brevity, this combination of LCP and RMQ is called the *LCP structure*. It is well known that all these structures can be built in $O(n)$ time (e.g., see [6]).

The *LZ-End parsing* [22, 23, 24] of a string s is a decomposition $s = f_1 f_2 \cdots f_z$ constructed by the following greedy process: if we have already processed a prefix $s[1..k] = f_1 f_2 \cdots f_{i-1}$, then $f_i[1..|f_i|-1]$ is the longest prefix of $s[k+1..|s|-1]$ that is a suffix of a string $f_1 f_2 \cdots f_j$ for some $j < i$; the substrings f_i are called *phrases*. For instance, the string *ababaaaaaac* has the LZ-End parsing *a.b.aba.aa.aaac*.

2 First Suboptimal Algorithm

Our first approach is based on two combinatorial properties of the LZ-End parsing that were observed in [19]. First, the definition of the LZ-End parsing easily implies the following lemma suggesting a way how to perform the construction of the LZ-End parsing incrementally.

▶ **Lemma 2.** *Let $f_1 f_2 \cdots f_z$ be the LZ-End parsing of a string s. If i is the maximal integer such that the string $f_{z-i} f_{z-i+1} \cdots f_z$ is a suffix of a string $f_1 f_2 \cdots f_j$ for $j < z - i$, then, for any letter a, the LZ-End parsing of the string sa is $f'_1 f'_2 \cdots f'_{z'}$, where $z' = z - i$, $f'_1 = f_1, f'_2 = f_2, \ldots, f'_{z'-1} = f_{z'-1}$, and $f'_{z'} = f_{z-i} f_{z-i+1} \cdots f_z a$.*

Secondly, it turns out that the number of phrases that might "unite" into a new phrase when a letter has been appended (as in Lemma 2) is severely restricted.

▶ **Lemma 3** (see [19]). *If $f_1 f_2 \cdots f_z$ is the LZ-End parsing of a string s, then, for any letter a, the last phrase in the LZ-End parsing of the string sa is 1) $f_{z-1}f_z a$ or 2) $f_z a$ or 3) a.*

The algorithm presented in this section builds the LZ-End parsing incrementally. When a prefix $s[1..k]$ is processed, we have the LZ-End parsing $f_1 f_2 \cdots f_z$ of $s[1..k]$ and we are to construct the parsing for the string $s[1..k{+}1]$. By Lemma 3, if $f_{z-1}f_z$ (resp., f_z) is a suffix of $f_1 f_2 \cdots f_j$ for some $j < z-1$ (resp., $j < z$), then the last phrase in the parsing of $s[1..k{+}1]$ is $f_{z-1}f_z s[k{+}1]$ (resp., $f_z s[k{+}1]$); otherwise, the last phrase is $s[k{+}1]$.

To process the cases of Lemma 3 efficiently, we maintain a bit array $M[1..n]$ that marks those prefixes in the lexicographically sorted set of all reversed prefixes of s that end at phrase boundaries: for $i \in [1..n]$, $M[i] = 1$ iff $s[1..\mathsf{SA}[i]] = f_1 f_2 \cdots f_j$ for some $j \in [1..z]$ in the LZ-End parsing $f_1 f_2 \cdots f_z$ of the current prefix $s[1..k]$. We equip M with the *van Emde Boas data structure* [33] that allows us to compute, for any given i, the maximal $j < i$ (resp., the minimal $j' > i$) (if any) such that $M[j] = 1$ (resp., $M[j'] = 1$); we use a dynamic version of this data structure that occupies $O(n)$ space and supports queries on M and modifications of the form $M[i] \leftarrow 1$ or $M[i] \leftarrow 0$ in $O(\log\log n)$ deterministic time (e.g., see [5]).

Let us describe how to check whether f_z has an *earlier* occurrence in $s[1..k] = f_1 f_2 \cdots f_z$ that ends at a phrase boundary. We first find in $O(\log\log n)$ time the maximal $j < \mathsf{ISA}[k]$ and the minimal $j' > \mathsf{ISA}[k]$ such that $M[j] = 1$ and $M[j'] = 1$. Suppose such j and j' exist (the case when either $M[1..\mathsf{ISA}[k]{-}1]$ or $M[\mathsf{ISA}[k]{+}1..n]$ consists of all zero is similar but simpler). Using the LCP structure, we compute in $O(1)$ time the length t (resp., t') of the longest common suffix of $s[1..k]$ and $s[1..\mathsf{SA}[j]]$ (resp., $s[1..k]$ and $s[1..\mathsf{SA}[j']]$). It is straightforward that f_z has an earlier occurrence in $s[1..k]$ ending at a phrase boundary iff $\max\{t, t'\} \geq |f_z|$.

Analogously, to check whether $f_{z-1}f_z$ has an earlier occurrence in $s[1..k]$ ending at a phrase boundary different from the boundary $|f_1 f_2 \cdots f_{z-1}|$, we temporarily unmark the bit $M[\mathsf{ISA}[k - |f_z|]]$ modifying the van Emde Boas data structure accordingly, then obtain the numbers t and t' in the same way as above, and restore $M[\mathsf{ISA}[k - |f_z|]]$ with the van Emde Boas data structure, all in $O(\log\log n)$ time; $f_{z-1}f_z$ has the required earlier occurrence iff $\max\{t, t'\} \geq |f_{z-1}f_z|$. Finally, by Lemmas 2 and 3, we obtain the LZ-End parsing of the prefix $s[1..k{+}1]$ by removing, based on the above computations, zero, one, or two last phrases from the list of all phrases and adding a new last phrase. The array M and the van Emde Boas data structure are modified accordingly. Thus, we have proved the following lemma.

▶ **Lemma 4.** *The LZ-End parsing of a string of length n over the alphabet $\{0, 1, \ldots, n^{O(1)}\}$ can be computed in $O(n \log\log n)$ time and $O(n)$ space.*

This algorithm provides good time guarantees (unlike the algorithms of Kreft and Navarro [24]) and seems to be of practical interest.

It is easy to see that the van Emde Boas data structure that is required to search predecessors and successors in the dynamic bit array M is the bottleneck of the described algorithm. It is known that for the insertion-only bit arrays there is an analogous data structure (the split-find data structure discussed below) that works in $O(n)$ overall time. Then, it is natural to ask whether the array M really requires a lot of deletions in the worst case or, based on the LZ77 intuition[1], only a few phrases in the LZ-End parsing of the current prefix might be removed in the future. The following example shows that a significant

[1] A similar incremental construction procedure for LZ77 would, at each step, append a letter to the end of the current string and then modify only the last phrase of the currently built parsing.

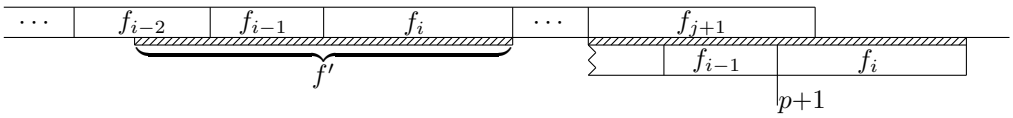


Figure 1 Case (2) in Lemma 6; f_i occurs at position $p{+}1$ located inside the phrase f_{j+1}.

amount of phrases from the LZ-End parsing of the current prefix can be removed in the final parsing and, therefore, the described approach strongly relies on the dynamic predecessor structure, which is known to require $\omega(1)$ query time [1, 28].

▶ **Example 5.** Choose an integer $k > 0$. Define $s_k = a_k$ and $s_i = a_i b_{i+1} s_{i+1}$ for $i = k{-}1, \ldots, 2, 1$, where a_i and b_i are distinct letters. Define $t_i = c_k c_{k-1} \cdots c_i$, where c_i are letters different from a_i and b_i. Our example is the string $s = s_k t_k s_{k-1} t_{k-1} \cdots s_2 t_2 s_1 t_1 b_2 s_2$ (notice the ending $b_2 s_2$). The string s is depicted below with separators "|", which are not real letters, at the end of each phrase of the LZ-End parsing of s (for readability, s is split into lines corresponding to the substrings $s_i t_i$ and the lines are aligned):

$$
\begin{array}{c}
a_k|c_k| \\
a_{k-1}|b_k|a_k c_k c_{k-1}| \\
\cdots\cdots\cdots\cdots \\
a_2|b_3|a_3 b_4 a_4 \cdots b_{k-2} a_{k-2} b_{k-1} a_{k-1} b_k a_k c_k c_{k-1} c_{k-2} \cdots c_2| \\
a_1|b_2|a_2 b_3 a_3 b_4 a_4 \cdots b_{k-2} a_{k-2} b_{k-1} a_{k-1} b_k a_k c_k c_{k-1} c_{k-2} \cdots c_2 c_1| \\
b_2 a_2|b_3 a_3|b_4 a_4| \cdots b_{k-2} a_{k-2}|b_{k-1} a_{k-1}|b_k a_k|.
\end{array}
$$

The parsing of any substring $s_i t_i$, for $i \in [2..k]$, consists of three phrases: two phrases corresponding to the letters a_i and b_{i+1} that did not occur before and the phrase $s_{i+1} t_{i+1} c_i$, where $s_{i+1} t_{i+1}$ is the previous line and c_i is a letter that did not occur before. Now consider the parsing of the last line $b_2 s_2$. For $i < k$, there is only one occurrence of a_i before the last line that is succeeded by a separator "|" and this occurrence is preceded by c_{i+1}. Analogously, for $i \leq k$, the only earlier occurrence of b_i succeeded by "|" is preceded by $c_i a_{i-1}$. This observation easily implies that the parsing of the last line consists of $k{-}1$ phrases $b_i a_i$.

Now consider the string $st_0 = s c_k c_{k-1} \cdots c_0$. The last phrase of the LZ-End parsing of st_0 is $b_2 s_2 t_0$ because $b_2 s_2 t_1$ is a suffix of the substring $s_1 t_1$ (kth line). Thus, this last phrase "absorbs" $k - 1$ last phrases of the parsing of s. It remains to notice that the length of s is $\Theta(k^2)$ and the number of phrases in the parsing of s is $\Theta(k)$.

3 Second Suboptimal Algorithm

Our second algorithm follows the definition of the LZ-End parsing constructing phrases greedily one by one from left to right. The algorithm itself is inefficient but we will show in Section 4 that its techniques can be combined with the incremental solution of Section 2 in order to obtain a linear algorithm.

Suppose that $f_1, f_2, \ldots, f_j$ are the first j phrases of the LZ-End parsing of s and f is a candidate for a new phrase, i.e., $f_1 f_2 \cdots f_j f$ is a prefix of s and $f[1..|f|{-}1]$ is a suffix of $f_1 f_2 \cdots f_k$ for $k \in [1..j]$. Our method "grows" f relying on the following lemma (see Fig. 1).

▶ **Lemma 6.** *Suppose that $f_1 f_2 \cdots f_z$ is the LZ-End parsing of a prefix of s. If, for $j \in [1..z{-}1]$, the phrase f_{j+1} is not present in the LZ-End parsing of the whole string s, then there exists $i \in [1..j]$ such that either (1) f_{j+1} is a suffix of the phrase f_i or (2) $f_1 f_2 \cdots f_i$ has a suffix f' such that $f_1 f_2 \cdots f_j f'$ is a prefix of s, f_{j+1} is a prefix of f', and $0 < |f'| - |f_i| < |f_{j+1}|$.*

Proof. Since f_{j+1} is not a phrase of the LZ-End parsing of s, it follows from Lemma 2 that there exists $j' \in [1..j]$ such that the first $j'+1$ phrases of the LZ-End parsing of s are $f_1, f_2, \ldots, f_{j'}, f$, where $f[1..|f|-1]$ contains f_{j+1} as a substring. By definition, there exists $i' \in [1..j']$ such that $f[1..|f|-1]$ is a suffix of $f_1 f_2 \cdots f_{i'}$. Suppose that the corresponding copy of f_{j+1} in $f_1 f_2 \cdots f_{i'}$ occurs at position q. Then, by construction, the suffix of $f_1 f_2 \cdots f_{i'}$ starting at position q occurs at position $|f_1 f_2 \cdots f_j|+1$. Hence, if $s[q..q+|f_{j+1}|-1] = f_{j+1}$ is a suffix of a phrase or is "intersected" by a phrase boundary, we easily obtain, respectively, (1) or (2). Otherwise, there is a phrase $\hat{f}$ such that $s[q..q+|f_{j+1}|-1] = f_{j+1}$ is a substring of $\hat{f}[1..|\hat{f}|-1]$ and the suffix of $\hat{f}$ starting at position q occurs at position $|f_1 f_2 \cdots f_j|+1$. In other words, this situation is analogous to the situation with f and we can analogously consider the "source" of $\hat{f}[1..|\hat{f}|-1]$ and the corresponding copy of f_{j+1} in this source. Then we repeat the analysis. Since this recursive procedure moves us to the left every time, it cannot continue forever and we will eventually find that either (1) or (2) holds. ◀

To extend f according to the case (1) of Lemma 6, we store all constructed phrases $f_1, f_2, \ldots, f_j$ in the lexicographically sorted order of their reversals, i.e., we store a permutation $i_1, i_2, \ldots, i_j$ of $[1..j]$ such that $\overleftarrow{f}_{i_1} \leq \overleftarrow{f}_{i_2} \leq \cdots \leq \overleftarrow{f}_{i_j}$. By the binary search in this sorted set, we find in $O(\log n)$ time, using the LCP structure, whether f is a suffix of f_i for some $i \in [1..j]$. This part is similar to the incremental approach of Section 2 (but less efficient).

To extend f according to the case (2) of Lemma 6, we process every position p of the considered occurrence of f in s and try to find a phrase f_i such that, as it is depicted in Figure 1 (assuming $f_{j+1} = f$), $i \in [1..j]$, f_i occurs at position $p+1$ embracing the last position of f (i.e., $p + |f_i| \geq |f_1 f_2 \cdots f_j f|$), and the prefix of f ending at position p is a suffix of $f_1 f_2 \cdots f_{i-1}$ (the details follow). Let us describe the data structures required to find such f_i.

Auxiliary data structures. First, we build the *suffix tree* of the string s (see the definition in, e.g., [6]); note that, unlike the suffix array SA that was built for the reversal $\overleftarrow{s}$, the suffix tree is built for the string s itself and, thus, contains the suffixes $s[1..n], s[2..n], \ldots, s[n..n]$. For simplicity, assume that $s[n]$ is a special letter that does not occur in $s[1..n-1]$ and, hence, the suffixes of s are in the one-to-one correspondence with the leaves of the tree. We build an array of pointers mapping suffixes of s to the corresponding leaves. It is well known (e.g., see [6]) that the suffix tree of s with this array can be constructed in $O(n)$ time.

Recall that the suffix tree has *explicit* and *implicit* vertices. The *string depth* of an (explicit or implicit) vertex is the length of the string written on the path connecting the root and the vertex. We augment the suffix tree with the following dynamic data structure.

▶ **Lemma 7.** *In $O(n)$ time one can build on the suffix tree of s a data structure supporting the following operations:*

1. *for a given number $w \in [1..n]$ and (explicit or implicit) vertex v, mark v and assign the weight w to v, all in $O(\log n)$ time;*
2. *for given numbers i, j, d and a leaf, find a marked vertex v that is an ancestor of this leaf, has weight $w \in [i..j]$, and has string depth at least d, all in $O(\log^2 n)$ time.*

Proof. The *heavy path decomposition* [32] is a decomposition of all vertices of the suffix tree into disjoint paths (called *heavy paths*), each of which descends from a vertex to a leaf, so that all ancestors of any given leaf belong to at most $\log n$ distinct heavy paths. It is shown in [32] that the heavy path decomposition can be constructed in $O(n)$ time.

We equip each heavy path with an (initially empty) dynamic 2-dimensional orthogonal range reporting data structure of [3]. To mark a given vertex v and assign a weight $w \in [1..n]$ to it, we simply insert in $O(\log n)$ time [3] the pair (d, w), where d is the string depth of v,

into the range reporting data structure corresponding to the heavy path containing v. In order to answer a query for given numbers i, j, d and a leaf, we consecutively process each of $O(\log n)$ ancestral heavy paths of this leaf starting from the deepest one: for each path, we perform in $O(\log n)$ time [3] the range reporting query $[d..d'] \times [i..j]$, where d' is equal to the string depth of the vertex having the "head" of the previously processed heavy path as a child and $d' = n$ for the path containing the leaf. It is easy to see that one of these range reporting queries must find (if any) the required marked ancestor whose weight is in the given range $[i..j]$ and whose string depth is at least d. Auxiliary structures organizing all fast navigation on the heavy paths can be easily constructed in $O(n)$ time. ◀

Also we utilize the following data structure, which can be viewed as a simplified version of the weighted ancestor data structures from [16, 20].

▶ **Lemma 8** (see [16, 20]). *In $O(n)$ time one can build on the suffix tree of s a data structure that, for a given leaf and a number d, allows us to find in $O(\log n)$ time the ancestor (explicit or implicit) of the leaf with the string depth d.*

To find an (explicit or implicit) vertex corresponding to a substring $s[i..j]$, one can perform the query of Lemma 8 on $d = j - i + 1$ and on the leaf corresponding to $s[i..n]$.

Algorithm. At the beginning, the algorithm builds in $O(n)$ time the LCP structure and the suffix tree of s equipped with the data structures of Lemmas 7 and 8. We maintain the following invariant: if the first j phrases $f_1, \dots, f_j$ of the LZ-End parsing of s are already constructed, then, for each $i \in [1..j]$, the vertex (implicit or explicit) of the suffix tree of s corresponding to the string f_i is marked and has weight $\mathsf{ISA}[|f_1 \cdots f_{i-1}|]$; we also store in a dynamic balanced tree a permutation $i_1, \dots, i_j$ of the set $[1..j]$ such that $\overleftarrow{f_{i_1}} \le \cdots \le \overleftarrow{f_{i_j}}$.

Suppose that we have already constructed j phrases $f_1, f_2, \dots, f_j$ and f is a candidate for the new phrase f_{j+1}, i.e., $f[1..|f|-1]$ is a suffix of $f_1 f_2 \cdots f_k$ for some $k \in [1..j]$. First, using the LCP structure and the balanced tree containing $i_1, \dots, i_j$, we perform in $O(\log n)$ time the binary search in the sorted set $\overleftarrow{f_{i_1}}, \dots, \overleftarrow{f_{i_j}}$ and find whether f is a suffix of f_i for some $i \in [1..j]$. If such f_i exists, then we "grow" f by one letter according to the case (1) of Lemma 6 and the string fa, where $a = s[|f_1 \cdots f_j f|+1]$, becomes a new candidate for f_{j+1}. Otherwise, we consecutively process from left to right each position p in the considered occurrence of f (i.e., $|f_1 f_2 \cdots f_j| < p < |f_1 f_2 \cdots f_j f|$) and check whether there is a phrase f_i, for $i \in [1..j]$, such that the prefix u of f ending at p (i.e., $u = s[|f_1 f_2 \cdots f_j|+1..p]$) is a suffix of $f_1 f_2 \cdots f_{i-1}$ and f_i occurs at position $p+1$ embracing the position $|f_1 f_2 \cdots f_j f|$ (i.e., $p + |f_i| \ge |f_1 f_2 \cdots f_j f|$); the procedure finding such f_i for a given p works in $O(\log^2 n)$ time and is described below in Lemma 9. Once such f_i is found for a position p, we "grow" f according to the case (2) of Lemma 6 so that the string $s[|f_1 f_2 \cdots f_j|+1..p+|f_i|+1]$, which contains f as a proper prefix, becomes a new candidate for f_{j+1}. Obviously, if we processed a position p in this way and could not extend f, then there is no reason to consider p in the future. Hence, the whole left to right processing of the positions of f can start not from the first position $|f_1 f_2 \cdots f_j|+1$ of f but from the last processed position of f (if any).

It follows from Lemma 6 that if we could not grow f neither by the processing of all positions p inside f nor by the processing of the case (1) of Lemma 6 described above, then f is the new phrase f_{j+1}. In this case, to maintain the invariant, we find the (explicit or implicit) vertex corresponding to the string $f = f_{j+1}$, mark this vertex, and assign the weight $\mathsf{ISA}[|f_1 f_2 \cdots f_j|]$ to it; all this is done in $O(\log n)$ time using the data structures from Lemmas 7 and 8. Further, using the binary search and the LCP structure, we insert the

string $\overleftarrow{f}_{j+1}$ in an appropriate place of the sorted set $\overleftarrow{f}_{i_1}, \ldots, \overleftarrow{f}_{i_j}$ and modify the balanced tree storing $i_1, \ldots, i_j$ accordingly, all in $O(\log n)$ time. Finally, the string $s[|f_1 \cdots f_{j+1}|{+}1]$ becomes a candidate for the next phrase f_{j+2} and we continue the construction.

Once we have performed in $O(\log^2 n)$ time the procedure finding an "extending" phrase f_i for a given position p inside f (see Lemma 9 below), we either grow the current candidate f by at least one letter or we do not grow f and this was the last processing of this position. By this observation, the overall running time of the algorithm is $O(n \log^2 n)$. The procedure itself is described in the following lemma, assuming that $h-1=j$, $f_h = f$, and $S = \{f_1, f_2, \ldots, f_j\}$; the lemma is formulated in a more general form that will be useful below in Section 4.

▶ **Lemma 9.** *Let $f_1 f_2 \cdots f_h$ be the LZ-End parsing of a prefix of s. Suppose that, for each f_i from a subset S of phrases, the vertex of the suffix tree of s corresponding to f_i is marked and has weight $\mathsf{ISA}[|f_1 f_2 \cdots f_{i-1}|]$. Then, for any position p such that $|f_1 f_2 \cdots f_{h-1}| < p < |f_1 f_2 \cdots f_h|$, one can find in $O(\log^2 n)$ time $f_i \in S$ (if any) such that $p + |f_i| \geq |f_1 f_2 \cdots f_h|$, f_i occurs at position $p{+}1$, and $s[|f_1 f_2 \cdots f_{h-1}|{+}1..p]$ is a suffix of $f_1 f_2 \cdots f_{i-1}$.*

Proof. Suppose that the required phrase $f_i \in S$ indeed exists. By assumption, the vertex v of the suffix tree that corresponds to the string f_i is marked and has weight $\mathsf{ISA}[|f_1 f_2 \cdots f_{i-1}|]$. The vertex v is an ancestor of the leaf corresponding to $s[p{+}1..n]$. Denote $u = s[|f_1 f_2 \cdots f_{h-1}|{+}1..p]$. Let $[\ell_u..r_u]$ be the maximal subrange of the range $[1..n]$ such that, for each $d \in [\ell_u..r_u]$, the string $s[1..\mathsf{SA}[d]]$ has a suffix u; the range $[\ell_u..r_u]$ can be calculated by the binary search in $O(\log n)$ time using the LCP structure. Since u is a suffix of $f_1 f_2 \cdots f_{i-1}$, the weight $\mathsf{ISA}[|f_1 f_2 \cdots f_{i-1}|]$ of the vertex v lies in the range $[\ell_u..r_u]$. Using the data structure of Lemma 7, we try to find in $O(\log^2 n)$ time a marked ancestor v of the leaf corresponding to $s[p{+}1..n]$ such that the weight of v is in the range $[\ell_u..r_u]$ and the string depth of v is at least $|f_1 f_2 \cdots f_h| - p$ (so that the string f_i corresponding to v occurs at position $p{+}1$ and embraces the last position of f_h; see Figure 1 assuming $f_{j+1} = f_h$). If such ancestor exists, we have found f_i. Otherwise, we decide that such $f_i \in S$ does not exist. It is straightforward that in this way we will necessarily find such $f_i \in S$ if it really exists. ◀

4 Linear Algorithm

Now we combine the two approaches described in Sections 2 and 3. On a high level, it is convenient to think that our algorithm is incremental as in Section 2 but it is guaranteed that only at most $\log^3 n$ last phrases from the LZ-End parsing of the currently processed prefix can be removed in the future. (In fact, any polylogarithmic threshold from $\omega(\log^2 n)$ will suffice.) After the processing of a prefix $s[1..k]$, we have the LZ-End parsing $s[1..k] = f_1 f_2 \cdots f_z$ and the phrases of this parsing are split into two groups: a set of first phrases $f_1, f_2, \ldots, f_j$ that cannot be removed from the parsing in the future (this is similar to the approach of Section 3) and at most $\log^3 n$ last phrases $f_{j+1}, f_{j+2}, \ldots, f_z$ that might be removed in the future. The phrases from the former group are called *static*. When the number of non-static phrases exceeds the threshold $\log^3 n$, the algorithm rebuilds the set of non-static phrases and, during this process, possibly marks some of them as static (see the detailed discussion below).

The algorithm maintains a bit array $M[1..n]$ defined as in Section 2 but only for the static phrases: $M[i] = 1$ iff $s[1..\mathsf{SA}[i]] = f_1 f_2 \cdots f_h$ for a static phrase f_h of the current parsing $f_1 f_2 \cdots f_z$. It follows from the above high level description that one can modify M only changing bits to ones. Therefore, the van Emde Boas data structure that answered predecessor/successor queries on M can be replaced with the following *split-find data structure* [12] (the settings of bits to ones can be viewed as splittings of continuous ranges of zeroes.)

▶ **Lemma 10** (see [12]). *There is a (split-find) data structure that, for any $i \in [1..n]$, can find (if any) the maximal $j \leq i$ (resp., minimal $j \geq i$) such that $M[j] = 1$ in $O(1)$ time and can perform (at most) n assignments $M[i] \leftarrow 1$ in overall $O(n)$ time.*

▶ **Lemma 11.** *For any $p \in [1..n]$, one can find in $O(1)$ time a static phrase f_i for which the length of the longest common suffix of $s[1..p]$ and $f_1 f_2 \cdots f_i$ is maximal (among all static f_i).*

Proof. The procedure is the same as in Section 2 but now we use the structure of Lemma 10 for predecessor/successor queries. We omit the details as they are straightforward. ◀

An analogous data structure for predecessor/successor queries on non-static phrases is organized using the so-called *fusion tree* [11].

▶ **Lemma 12** (see [11, 29]). *The fusion tree can maintain a set of at most $\log^3 n$ integers under the following operations, each of which takes $O(1)$ time:*

1. *insert an integer x with user-defined satellite information into the set;*
2. *remove an integer x from the set;*
3. *for an integer x, find (if any) in the set the maximal $y \leq x$ (resp., minimal $y \geq x$) with the corresponding satellite information.*

▶ **Lemma 13.** *Suppose that, for each phrase f_i from a set S of phrases, the fusion tree stores the number $\mathsf{ISA}[|f_1 f_2 \cdots f_i|]$ with the satellite information containing a pointer to f_i. Then, for any $p \in [1..n]$, one can find in $O(1)$ time a phrase $f_i \in S$ for which the length of the longest common suffix of $s[1..p]$ and $f_1 f_2 \cdots f_i$ is maximal (among all $f_i \in S$).*

Proof. The proof is analogous to the proof of Lemma 11. We omit the obvious details. ◀

During the incremental construction, the fusion tree stores the set of all non-static phrases as described in Lemma 13. Suppose that we have processed a prefix $s[1..k]$ and $f_1 f_2 \cdots f_z$ is the LZ-End parsing of this prefix. To check whether f_z has an earlier occurrence ending at a phrase boundary, we temporarily remove f_z from the fusion tree, apply Lemmas 11 and 13 thus obtaining, respectively, static and non-static phrases f_i and $f_{i'}$ described in these lemma, and use the LCP structure to calculate the lengths of the longest common suffixes of f_z and $f_1 f_2 \cdots f_i$, and of f_z and $f_1 f_2 \cdots f_{i'}$; then, f_z has the required occurrence iff one of these two computed lengths is greater than or equal to $|f_z|$. To check whether $f_{z-1} f_z$ has an earlier occurrence ending at a phrase boundary, we do the same but also temporarily remove f_{z-1} from the fusion tree. After this, the temporarily removed phrases f_{z-1} and f_z are restored. According to the results of the checking, we remove zero, or one (f_z), or two (f_{z-1}, f_z) phrases from the fusion tree and insert a new phrase, resp., $s[k{+}1]$, or $f_z s[k{+}1]$, or $f_{z-1} f_z s[k{+}1]$, thus constructing the parsing of $s[1..k{+}1]$. The whole procedure takes $O(1)$ time. Clearly, such incremental algorithm works in $O(n)$ overall time but sometimes we have a problem: the new non-static phrase inserted in the fusion tree can exceed the limit of $\log^3 n$ elements. Such overflows of the fusion tree are fixed in two ways described below.

Overflows of the fusion tree 1. Let the fusion tree contain the phrases $f_{j+1}, f_{j+2}, \ldots, f_z$ of the LZ-End parsing $f_1 f_2 \cdots f_z$ of $s[1..k]$. Suppose that the fusion tree overflows when the letter $s[k{+}1]$ is appended; obviously, this can happen only if $z - j = \lfloor \log^3 n \rfloor$ and the last phrase of the parsing of $s[1..k{+}1]$ is $s[k{+}1]$. We are to rebuild the current set of non-static phrases $f_{j+1}, f_{j+2}, \ldots, f_z$ (we assume that $s[k{+}1]$ is not inserted in the fusion tree yet) in order to fix the coming overflow. The algorithm maintains a variable t that contains the sum of the lengths of all non-static phrases, i.e., $t = |f_{j+1} f_{j+2} \cdots f_z|$ at the given moment.

We try to unload the fusion tree performing the following procedure consecutively for each of the t positions $k+1, k+2, \ldots, k+t$ from left to right (for simplicity, assume that $k+t < n$; the case $k+t \geq n$ is analogous): for a position p, we apply Lemmas 11, 13 and use the LCP structure in the same way as above in order to check in $O(1)$ time whether the string $s[|f_1 f_2 \cdots f_{z-1}|{+}1..p]$ is a suffix of a string $f_1 f_2 \cdots f_i$ for some $i \in [1..z{-}1]$. Suppose that this checking has succeeded and $q \in [k{+}1..k{+}t]$ is the leftmost position for which $s[|f_1 f_2 \cdots f_{z-1}|{+}1..q]$ is a suffix of $f_1 f_2 \cdots f_i$ for some $i \in [1..z{-}1]$. Then, it follows from Lemma 3 that the parsing of the string $s[1..q{+}1]$ is $f_1 f_2 \cdots f_{z-1} f$, where $f = s[|f_1 f_2 \cdots f_{z-1}|{+}1..q{+}1]$. Hence, once such position q is found, we stop the processing of the positions and modify the fusion tree in $O(1)$ time removing the phrase f_z and putting the new phrase f inside. Since the modified fusion tree contains only the phrases $f_{j+1}, f_{j+2}, \ldots, f_{z-1}, f$ (i.e., the same number $z-j$), it is not overflowed and, therefore, our incremental algorithm can continue the execution from the prefix $s[1..q{+}1]$.

Since each position is analyzed in $O(1)$ time, the processing takes $O(q-k)$ time if such position q was found (and $O(t)$ time otherwise). Therefore, if every overflow of the fusion tree during the work of the algorithm is successfully fixed by the described method, then the construction of the LZ-End parsing of the whole string s takes $O(n)$ time. It remains to consider the case when this method could not find the required position q.

Overflows of the fusion tree 2. As in Section 3, at the beginning, our algorithm builds the suffix tree of s equipped with the data structures of Lemmas 7 and 8. We maintain the following invariant: for each static phrase f_i such that $|f_i| \geq \log^3 n$, the (explicit or implicit) vertex of the suffix tree corresponding to f_i is marked and has weight $\mathsf{ISA}[|f_1 f_2 \cdots f_{i-1}|]$, i.e., the invariant is like in Section 3 but only for static and sufficiently long phrases.

Suppose that, after the fusion tree overflow occurred on the prefix $s[1..k{+}1]$ of s, we processed all $t = |f_{j+1} f_{j+2} \cdots f_z|$ positions $k+1, k+2, \ldots, k+t$ as above but could not "grow" the last phrase f_z of the parsing $f_1 f_2 \cdots f_z$ of $s[1..k]$. We say that a phrase f_h of the parsing is *extendable* if there is $q \geq |f_1 f_2 \cdots f_h|$ such that $s[|f_1 f_2 \cdots f_{h-1}|{+}1..q]$ is a suffix of $f_1 f_2 \cdots f_i$ for some $i \in [1..h{-}1]$. (Note that q cannot be equal to n since we assumed that $s[n]$ does not occur in $s[1..n{-}1]$.) Since the positions $k+1, k+2, \ldots, k+t$ all were unsuccessfully processed by the above procedure trying to "extend" f_z, q must be greater than $k+t$ and, by Lemma 6, the phrase f_i "extending" f_h can be chosen so that f_i starts inside f_h (as in Fig. 1) and has length at least $t+1 \geq \log^3 n$. For simplicity of exposition, we summarize this in the following lemma, which is an easy corollary of Lemma 6.

▶ **Lemma 14.** *Let t be a positive integer. Denote by $f_1 f_2 \cdots f_z$ the LZ-End parsing of a prefix $s[1..k]$ of s. Suppose that, for each $q \in [k..k{+}t]$, there is no $i \in [1..z{-}1]$ such that $s[|f_1 f_2 \cdots f_{z-1}|{+}1..q]$ is a suffix of $f_1 f_2 \cdots f_i$. Then, for any extendable phrase f_h with $h \in [1..z]$, there exist $i \in [1..h{-}1]$ and a position p such that $|f_1 f_2 \cdots f_{h-1}| < p < |f_1 f_2 \cdots f_h|$, $p{+}|f_i| > k{+}t$, f_i occurs at position $p{+}1$, and $s[|f_1 f_2 \cdots f_{h-1}|{+}1..p]$ is a suffix of $f_1 f_2 \cdots f_{i-1}$.*

Since $t = |f_{j+1} f_{j+2} \cdots f_z|$, at most $\log^2 n$ non-static phrases have length $\geq t/\log^2 n$ and most non-static phrases ($\geq z - j - \log^2 n = \lfloor \log^3 n \rfloor - \log^2 n$) have length $< t/\log^2 n$. (The choice of the threshold $t/\log^2 n$ is clarified below.) By a simple traversal of non-static phrases, we find in $O(z-j) \subset O(t)$ time the rightmost non-static phrase f_h such that $|f_h| < t/\log^2 n$. By Lemma 14, if f_h is extendable, then it can be "extended" by a phrase f_i of length $>t$ such that f_i occurs at a position inside f_h. Since $|f_i| > t$ and t is the sum of the lengths of all non-static phrases, f_i must be static. Therefore, by the invariant, the vertex of the suffix tree

corresponding to f_i is marked and has weight $\mathsf{ISA}[|f_1 f_2 \cdots f_{i-1}|]$. Based on this observation, the algorithm decides whether f_h is extendable processing each position of f_h in $O(\log^2 n)$ time by the procedure of Lemma 9 (assuming that the set S from Lemma 9 corresponds to the invariant) in the same way as in Section 3. The overall time of this processing is $O(|f_h| \log^2 n) = O(\frac{t}{\log^2 n} \log^2 n) = O(t)$. (That is why the threshold is $t/\log^2 n$.) The further overflow fixing procedure depends on whether the phrase f_h is extendable.

Suppose that f_h is not extendable (it is a simpler case). Then, the algorithm marks the non-static phrases $f_{j+1}, f_{j+2}, \ldots, f_h$ as static, sets $M[\mathsf{ISA}[|f_1 f_2 \cdots f_{h'}|]] \leftarrow 1$, for $h' \in [j{+}1..h]$, modifying the data structure of Lemma 10 accordingly, and removes these phrases from the fusion tree. This is correct due to the following straightforward lemma.

▶ **Lemma 15.** *Suppose that $f_1 f_2 \cdots f_z$ is the LZ-End parsing of a prefix $s[1..k]$ of s. For $h \in [1..z]$, if the phrase f_h is non-extendable, then so are all the phrases $f_1, f_2, \ldots, f_{h-1}$.*

To maintain the invariant, for each new static phrase $f_{h'}$ such that $|f_{h'}| \geq \log^3 n$, the algorithm finds in $O(\log n)$ time using the data structure of Lemma 8 the vertex of the suffix tree corresponding to the string $f_{h'}$ and marks this vertex assigning the weight $\mathsf{ISA}[|f_1 f_2 \cdots f_{h'-1}|]$ to it in $O(\log n)$ time using the data structure of Lemma 7. After this, only phrases of length $\geq t/\log^2 n$ can remain in the fusion tree. Since there are at most $\log^2 n$ such phrases, the fusion tree is not overflowed and we can continue our incremental algorithm from the prefix $s[1..k{+}1]$ whose parsing is $f_1 f_2 \cdots f_z s[k{+}1]$. (This case of non-extendable f_h makes the overall time estimation of the algorithm non-trivial; see the discussion below.)

Suppose that f_h is extendable and we found $q > k + t$ and a static phrase f_i such that $s[|f_1 f_2 \cdots f_{h-1}|{+}1..q]$ is a suffix of $f_1 f_2 \cdots f_i$ (it is important that f_i is static). We are to compute the LZ-End parsing of the string $s[1..q{+}1]$ based on the following lemma.

▶ **Lemma 16.** *Let $f_1 f_2 \cdots f_z$ be the LZ-End parsing of a prefix $s[1..k]$ of s. Suppose that, for $h \in [1..z]$ and $q > k$, $s[|f_1 f_2 \cdots f_{h-1}|{+}1..q]$ is a suffix of $f_1 f_2 \cdots f_i$ for a static phrase f_i. Then, the LZ-End parsing of $s[1..q{+}1]$ has one of the following forms: (1) $f_1 f_2 \cdots f_m f$, for $m < h$, or (2) $f_1 f_2 \cdots f_m f' f$, for $m < h - 1$, such that $f_1 f_2 \cdots f_z$ is a prefix of $f_1 f_2 \cdots f_m f'$.*

Proof. Suppose that the LZ-End parsing of $s[1..q]$ coincides with the parsing of $s[1..k]$ on the first d phrases, i.e., $s[1..q] = f_1 f_2 \cdots f_d f'_1 f'_2 \cdots f'_c$ for some $c \geq 1$. It follows from Lemma 2 that $f_1 f_2 \cdots f_z$ is a prefix of $f_1 f_2 \cdots f_d f'_1$. Since f_i is static, we have $i \leq d$, i.e., the phrases $f_1, f_2, \ldots, f_i$ are presented in the parsing of $s[1..q]$. Therefore, by Lemma 2, the LZ-End parsing of $s[1..q{+}1]$ is either $f_1 f_2 \cdots f_d f'_1 f$ (here, the new phrase f "absorbs" the phrases $f'_2, f'_3, \ldots, f'_c$; we put $m := d$ and $f' := f'_1$) or $f_1 f_2 \cdots f_m f$ for some $m \leq d$ (the new phrase f "absorbs" f'_1 and, probably, some of the phrases $f_d, f_{d-1}, \ldots$). Since f necessarily "absorbs" the phrases $f_h f_{h+1} \cdots f_z$, we have $d < h - 1$ in the former case and, hence, $m < h - 1$. ◀

The main problem is to find f and f' from Lemma 16 (and to determine whether f' really exists). For this, we perform a version of the incremental algorithm for the positions $k + t + 1, k + t + 2, \ldots, q$ from left to right; the difference is that, during this, we do not store any auxiliary phrases that appear as substrings of $s[k{+}1..q]$ because anyway, by Lemma 16, they are not present in the final parsing of $s[1..q{+}1]$. Let us discuss this in more details.

Let $Q = \{q_1, q_2, \ldots, q_c\}$ be the increasing sequence of all positions from $[k{+}t{+}1..q]$ such that the LZ-End parsing of $s[1..q_d{+}1]$, for any $d \in [1..c]$, has the form $f_1 f_2 \cdots f_{m_d} f'_d$ for some $m_d \leq z$ and some phrase f'_d. It is convenient to imagine an incremental algorithm (as in Section 2) that builds the parsing of the string $s[1..q{+}1]$ incrementally starting from the parsing of $s[1..k{+}t]$; our goal is to determine the moments when this algorithm passes

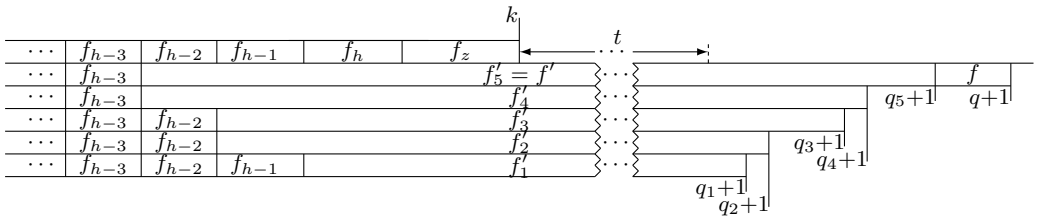


Figure 2 Construction of the parsing from Lemma 16; here $h = z - 1$, $c = 5$, $m_1 = h - 1, m_2 = m_3 = h-2, m_4 = m_5 = m = h-3$. Each line depicts the parsing of $s[1..q'+1]$ for $q' \in \{q_1, q_2, q_3, q_4, q\}$.

the positions from Q (also see Fig. 2 for clarifications). By the choice of q, the string $s[|f_1 f_2 \cdots f_{h-1}|+1..q]$ is a suffix of $f_1 f_2 \cdots f_i$, for static phrase f_i, and, therefore, if none of the positions $[k+t+1..q-1]$ belongs to Q, then q must belong to it. By definition of Q, the case (1) of Lemma 16 is realized iff $q_c = q$. Further, the phrases f', f and the number m from Lemma 16 can be determined as follows: $f = f'_c$ and $m = m_c$ if $q_c = q$ (case (1)), and $f' = f'_c$, $f = s[q_c+2..q+1]$, $m = m_c$ otherwise (case (2)). Note that the "source" of the phrase f found during the calculation of q_c and m_c below might differ from the "source" $f_1 f_2 \cdots f_i$ and might have a longer common suffix with $s[1..q]$. Thus, it remains to compute q_c and m_c through the imitation of the work of our imaginary incremental algorithm.

During the processing of the positions $[k+t+1..q]$, our real algorithm maintains a variable m that is equal to m_d for the last processed position $q_d \in Q$ (initially, $m = z$) and the fusion tree stores only the phrases $f_{j+1}, f_{j+2}, \ldots, f_m$ (so that, initially, it stores all non-static phrases). For each $p = k + t + 1, k + t + 2, \ldots, q$ from left to right, we apply Lemmas 11, 13 and use the LCP structure (in the same way as in the beginning of this section) to find in $O(1)$ time a phrase $f_{i'}$ such that $f_{i'}$ either is static or is currently in the fusion tree and the length ℓ of the longest common suffix of $s[1..p]$ and $f_1 f_2 \cdots f_{i'}$ is maximal (among all such phrases $f_{i'}$). Then, p belongs to Q iff $\ell \geq p - |f_1 f_2 \cdots f_m|$. Further, if $\ell \geq p - |f_1 f_2 \cdots f_{m-1}|$ and $i' \neq m$, we remove the phrase f_m from the fusion tree and decrease m by one; in the special case when $\ell \geq p - |f_1 f_2 \cdots f_{m-1}|$ and $i' = m$, we remove f_m from the fusion tree, repeat the processing of p, and, if m did not change in this second attempt, restore f_m. The variable m is decreased only by one since, as it follows from Lemmas 2 and 3, for any $d \in [1..c]$, we have either $m_d = m_{d-1}$ or $m_d = m_{d-1} - 1$, assuming $m_0 = z$. The described algorithm computes the numbers q_c and m_c (and, thus, f, f', and m) in $O(q - k)$ time.

We remove the phrases $f_{m+1}, \ldots, f_z$ from the fusion tree and put f and f' (if f' does exist) in it. So, by Lemma 16, the set of non-static phrases of $s[1..q+1]$ consists of either $f_{j+1}, f_{j+2}, \ldots, f_m, f$, for $m < h$, or $f_{j+1}, f_{j+2}, \ldots, f_m, f', f$, for $m < h-1$. Since $h-j \leq z-j$, there are at most $z - j = \lfloor \log^3 n \rfloor$ phrases in this set. Therefore, the fusion tree is not overflowed anymore and the algorithm can continue the execution from the prefix $s[1..q+1]$.

The correctness of the whole algorithm of this section should be clear at this point.

Time estimation. The algorithm processes each position of s in $O(1)$ time from left to right until it reaches a position $k+1$ where the fusion tree overflows when the letter $s[k+1]$ is appended. The overflow is fixed in two ways. First, the algorithm processes each of the positions $k + 1, k + 2, \ldots, k + t$ in $O(1)$ time from left to right, for an appropriate value of t, until it finds a position q such that our usual algorithm can continue the execution from the prefix $s[1..q+1]$ with the fixed non-overflowed fusion tree. It is obvious that all fixing procedures of this kind take $O(n)$ overall time. Thus, it remains to consider the time required to fix the overflows in which the processing of the corresponding positions $k + 1, k + 2, \ldots, k + t$ could not help; we refer to the overflows of this kind as *hard overflows*.

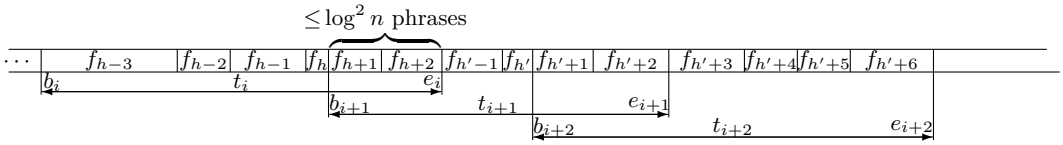


Figure 3 The case in the proof of Lemma 17 when f_h and $f_{h'}$ both are not extendable. The depicted phrases are from the LZ-End parsing of the prefix $s[1..e_{i+2}]$.

To maintain the invariant, the algorithm marks in the suffix tree the vertices corresponding to the static phrases of length at least $\log^3 n$. As there are at most $O(n/\log^3 n)$ such phrases and each marking takes $O(\log n)$ time, the overall time required for the maintenance of the invariant is $o(n)$ and, hence, we can exclude the time spent on these markings from the consideration. Denote by t_i the value of the variable t at the moment when the ith hard overflow occurs. Suppose that the ith hard overflow occurs when the algorithm reaches a prefix $s[1..k_i]$, for some k_i, and tries to process $s[1..k_i{+}1]$. The processing of this hard overflow takes $O(t_i + q_i - k_i)$ time, where $s[1..q_i{+}1]$ is a prefix from which the algorithm continues its execution after the fixing of the overflow. It is easy to see that $\sum_i (t_i + q_i - k_i) = O(n) + \sum_i t_i$ and, hence, it suffices to prove that, for any input string $s[1..n]$, we have $\sum t_i = O(n)$.

Consider the ith hard overflow. Suppose that it occurs on a prefix $s[1..k]$ with the LZ-End parsing $f_1 f_2 \cdots f_z$ and the fusion tree contains the phrases $f_{j+1}, f_{j+2}, \ldots, f_z$ at this moment. Denote $b_i = |f_1 f_2 \cdots f_j| + 1$ and $e_i = |f_1 f_2 \cdots f_z|$ ("b" and "e" are shortenings for "begin" and "end"). Since $t_i = |f_{j+1} f_{j+2} \cdots f_z|$, we have $e_i = b_i + t_i - 1$.

▶ **Lemma 17.** *Suppose that d is the number of hard overflows occurred during the processing of a string s. Then, for any $i \in [1..d{-}2]$, we have $b_{i+2} + e_{i+2} \geq b_i + e_i + t_i$.*

Proof. Note that the sequences $\{b_i\}$ and $\{e_i\}$ are non-decreasing and $b_i < e_i$ for any $i \in [1..d]$. Recall that the ith hard overflow occurs after the processing of the prefix $s[1..e_i]$ and the procedure fixing the overflow tries to "extend" a non-static phrase f_h of the LZ-End parsing of $s[1..e_i]$. Due to Lemma 14, if f_h is extendable, then the algorithm continues its execution from a prefix $s[1..q{+}1]$ for some $q > e_i + t_i$. Therefore, we obtain $e_{i+1} \geq q > e_i + t_i$ and, hence, $b_i + e_i + t_i < b_i + e_{i+1} \leq b_{i+2} + e_{i+2}$.

Suppose that f_h is not extendable. Then, the algorithm marks f_h and all phrases to the left of f_h as static and only at most $\log^2 n$ phrases (of length $\geq t_i/\log^2 n$) remain in the fusion tree (see Fig. 3 and 4). Now consider the $(i{+}1)$st hard overflow. It follows from the above discussion that only at most $\log^2 n$ first phrases of the fusion tree can contain phrases of the parsing of $s[1..e_i]$ at this moment. The procedure fixing the $(i{+}1)$st hard overflow analogously tries to "extend" a non-static phrase $f_{h'}$ of the LZ-End parsing of $s[1..e_{i+1}]$. This phrase $f_{h'}$ is the rightmost phrase of length $< t_{i+1}/\log^2 n$. Since there are at least $\lfloor \log^3 n \rfloor - \log^2 n$ phrases of length $< t_{i+1}/\log^2 n$, the phrase $f_{h'}$ cannot coincide with any of the phrases from the parsing of $s[1..e_i]$ and, therefore, it must occur at a position to the right of the position e_i (see Fig. 3 and 4 for a clarification).

Suppose that $f_{h'}$ is not extendable (see Fig. 3). Then, the algorithm marks $f_{h'}$ and all phrases to the left of $f_{h'}$ as static. Hence, during the $(i{+}2)$nd hard overflow, b_{i+2} must be greater than the rightmost position of $f_{h'}$ and, thus, $b_{i+2} > e_i$ (see Fig. 3). Since $e_i = b_i + t_i - 1$, the later implies $b_i + t_i \leq b_{i+2}$ and, hence, $b_i + e_i + t_i \leq b_{i+2} + e_{i+2}$.

Suppose that $f_{h'}$ is extendable. Since $t_i \leq (b_{i+1} - b_i) + t_{i+1}$ (see Fig. 4), we derive $b_i + e_i + t_i \leq b_i + e_i + (b_{i+1} - b_i) + t_{i+1} = b_{i+1} + e_i + t_{i+1} \leq b_{i+1} + e_{i+1} + t_{i+1}$. By Lemma 14, since $f_{h'}$ is found to be extendable, after the $(i{+}1)$st hard overflow the algorithm continues

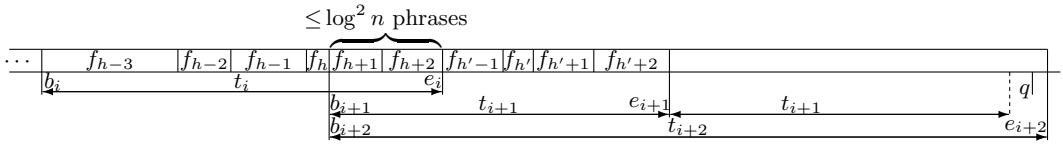


Figure 4 The case in the proof of Lemma 17 when f_h is not extendable and $f_{h'}$ is extendable. The depicted phrases are from the LZ-End parsing of the prefix $s[1..e_{i+1}]$.

its execution from a prefix $s[1..q+1]$ for some $q > e_{i+1} + t_{i+1}$. Therefore, we obtain $e_{i+2} \geq q > e_{i+1} + t_{i+1}$ (see Fig. 4), which implies $b_i + e_i + t_i \leq b_{i+1} + e_{i+1} + t_{i+1} < b_{i+2} + e_{i+2}$. ◀

It follows from Lemma 17 that $\sum_{i=1}^{d-2} t_i \leq \sum_{i=1}^{d-2}(b_{i+2} + e_{i+2} - b_i - e_i) = b_d + e_d + b_{d-1} + e_{d-1} - b_1 - e_1 - b_2 - e_2$, which is obviously $O(n)$. Therefore, since $t_d + t_{d-1} = O(n)$, we obtain $\sum_{i=1}^{d} t_i = O(n)$. This finally proves Theorem 1.

References

1. P. Beame and F. E. Fich. Optimal bounds for the predecessor problem. In *STOC 1999*, pages 295–304. ACM, 1999. doi:10.1006/jcss.2002.1822.
2. D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel–Ziv parsing. In *SODA 2016*, pages 2053–2071. SIAM, 2016. doi:10.1137/1.9781611974331.ch143.
3. G. E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *SODA 2008*, pages 894–903. SIAM, 2008.
4. F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Universal indexes for highly repetitive document collections. *Information Systems*, 61:1–23, 2016. doi:10.1016/j.is.2016.04.002.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms, third edition*. MIT press, 2009.
6. M. Crochemore and W. Rytter. *Jewels of stringology*. World Scientific Publishing Co. Pte. Ltd., 2002.
7. H. Ferrada, T. Gagie, T. Hirvola, and S. J. Puglisi. Hybrid indexes for repetitive datasets. *Phil. Trans. R. Soc. A*, 372, 2014. doi:10.1098/rsta.2013.0137.
8. P. Ferragina and G. Manzini. On compressing the textual web. In *WSDM 2010*, pages 391–400. ACM, 2010. doi:10.1145/1718487.1718536.
9. J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *ESA 2015*, volume 9294 of *LNCS*, pages 533–544. Springer, 2015. doi:10.1007/978-3-662-48350-3_45.
10. J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *CPM 2006*, volume 4009 of *LNCS*, pages 36–48. Springer, 2006. doi:10.1007/11780441_5.
11. M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.
12. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *STOC 1983*, pages 246–251. ACM, 1983. doi:10.1145/800061.808753.
13. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *LATA 2012*, volume 7183 of *LNCS*, pages 240–251. Springer, 2012. doi:10.1007/978-3-642-28332-1_21.
14. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *LATIN 2014*, volume 8392 of *LNCS*, pages 731–742. Springer, 2014. doi:10.1007/978-3-642-54423-1_63.

15 T. Gagie, P Gawrychowski, and S. J. Puglisi. Faster approximate pattern matching in compressed repetitive texts. In *ISAAC 2011*, volume 7074 of *LNCS*, pages 653–662. Springer, 2011. doi:10.1007/978-3-642-25591-5_67.

16 P. Gawrychowski, M. Lewenstein, and P. K. Nicholson. Weighted ancestors in suffix trees. In *ESA 2014*, volume 8737 of *LNCS*, pages 455–466. Springer, 2014. doi:10.1007/978-3-662-44777-2_38.

17 J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel–Ziv factorization: Simple, fast, small. In *CPM 2013*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013. doi:10.1007/978-3-642-38905-4_19.

18 J. Kärkkäinen, D. Kempa, and S. J Puglisi. Lempel-Ziv parsing in external memory. In *DCC 2014*, pages 153–162. IEEE, 2014. doi:10.1109/DCC.2014.78.

19 D. Kempa and D. Kosolobov. LZ-End parsing in compressed space. In *DCC 2017*, pages 350–359. IEEE, 2017. doi:10.1109/DCC.2017.73.

20 T. Kopelowitz and M. Lewenstein. Dynamic weighted ancestors. In *SODA 2007*, pages 565–574. SIAM, 2007.

21 D. Kosolobov. Faster lightweight Lempel–Ziv parsing. In *MFCS 2015*, volume 9235 of *LNCS*, pages 432–444, 2015. doi:10.1007/978-3-662-48054-0_36.

22 S. Kreft and G. Navarro. LZ77-like compression with fast random access. In *DCC 2010*, pages 239–248. IEEE, 2010. doi:10.1109/DCC.2010.29.

23 S. Kreft and G. Navarro. Self-indexing based on LZ77. In *CPM 2011*, volume 6661 of *LNCS*, pages 41–54. Springer, 2011. doi:10.1007/978-3-642-21458-5_6.

24 S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013. doi:10.1016/j.tcs.2012.02.006.

25 V. Mäkinen and G. Navarro. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007. doi:10.1145/1216370.1216372.

26 J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *SODA 2017*, pages 408–424. SIAM, 2017. doi:10.1137/1.9781611974782.26.

27 G. Navarro. Indexing text using the Ziv-Lempel trie. *J. Discrete Algorithms*, 2(1):87–114, 2004. doi:10.1016/S1570-8667(03)00066-2.

28 M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *STOC 2006*, pages 232–240. ACM, 2006. doi:10.1145/1132516.1132551.

29 M. Pătraşcu and M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *FOCS 2014*, pages 166–175. IEEE, 2014. doi:10.1109/FOCS.2014.26.

30 A. Policriti and N. Prezza. Computing LZ77 in run-compressed space. In *DCC 2016*, pages 23–32. IEEE, 2016. doi:10.1109/DCC.2016.30.

31 J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *SPIRE 2008*, volume 5280 of *LNCS*, pages 164–175. Springer, 2008. doi:10.1007/978-3-540-89097-3_17.

32 D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

33 P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical systems theory*, 10(1):99–127, 1976. doi:10.1007/BF01683268.

34 J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.

Combinatorial n-fold Integer Programming and Applications*†

Dušan Knop[1], Martin Koutecký[2], and Matthias Mnich[3]

1 **Charles University, Prague, Czech Republic, and Department of Informatics, University of Bergen, Bergen, Norway**
knop@kam.mff.cuni.cz

2 **Charles University, Prague, Czech Republic**
koutecky@kam.mff.cuni.cz

3 **Universität Bonn, Bonn, Germany, and Maastricht University, Maastricht, The Netherlands**
mmnich@uni-bonn.de

Abstract

Many fundamental NP-hard problems can be formulated as integer linear programs (ILPs). A famous algorithm by Lenstra allows to solve ILPs in time that is exponential only in the dimension of the program. That algorithm therefore became a ubiquitous tool in the design of fixed-parameter algorithms for NP-hard problems, where one wishes to isolate the hardness of a problem by some parameter. However, it was discovered that in many cases using Lenstra's algorithm has two drawbacks: First, the run time of the resulting algorithms is often doubly-exponential in the parameter, and second, an ILP formulation in small dimension can not easily express problems which involve many different costs.

Inspired by the work of Hemmecke, Onn and Romanchuk [Math. Prog. 2013], we develop a single-exponential algorithm for so-called *combinatorial n-fold integer programs*, which are remarkably similar to prior ILP formulations for various problems, but unlike them, also allow variable dimension. We then apply our algorithm to a few representative problems like Closest String, Swap Bribery, Weighted Set Multicover, and obtain exponential speedups in the dependence on the respective parameters, the input size, or both.

Unlike Lenstra's algorithm, which is essentially a bounded search tree algorithm, our result uses the technique of augmenting steps. At its heart is a deep result stating that in combinatorial n-fold IPs an existence of an augmenting step implies an existence of a "local" augmenting step, which can be found using dynamic programming. Our results provide an important insight into many problems by showing that they exhibit this phenomenon, and highlights the importance of augmentation techniques.

1998 ACM Subject Classification F2.2 Nonnumerical Algorithms and Problems

Keywords and phrases integer programming, closest strings, fixed-parameter algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.54

* D.K. supported by the CE-ITI grant project P202/12/G061 of GA ČR; M.K. supported by the GA UK grant project 1784214. M.M. supported by ERC Starting Grant 305465 (BeyondWorstCase).
† A full version of the paper is available at https://arxiv.org/abs/1705.08657.

1 Introduction

The Integer Linear Programming (ILP) problem is fundamental as it models many combinatorial optimization problems. Since it is NP-complete, we naturally ask about the complexity of special cases. A fundamental algorithm by Lenstra from 1983 shows that ILPs can be solved in polynomial time when their number of variables (the dimension) d is fixed [30]; that algorithm is thus a natural tool to prove that the complexity of some special cases of other NP-hard problems is also polynomial.

A systematic way to study the complexity of "special cases" of NP-hard problems has been developed in the past 25 years in the field of parameterized complexity. There, the problem input is augmented by some integer parameter k, and one then measures the problem complexity in terms of both the instance size n as well as k. Of central importance are algorithms with run times of the form $f(k)n^{O(1)}$ for some computable function f, which are called *fixed-parameter algorithms*; the key idea is that the degree of the polynomial does not grow with k. For background on parameterized complexity, we refer to the monograph [7].

Kannan's improvement [23] of Lenstra's algorithm runs in time $d^{O(d)}n$, which is thus a fixed-parameter algorithm for parameter d. Gramm et al. [17] pioneered the application of Lenstra's and Kannan's algorithm in parameterized complexity, giving a fixed-parameter algorithm for the Closest String problem [17]. This led Niedermeier [34] to propose:

> *[...] It remains to investigate further examples besides* Closest String *where the described ILP approach turns out to be applicable. More generally, it would be interesting to discover more connections between fixed-parameter algorithms and (integer) linear programming.*

Since then, many more applications of Lenstra's and Kannan's algorithm for parameterized problems have been proposed. However, essentially all of them [5, 9, 10, 21, 33, 29] share a common trait with the algorithm for Closest String: they have a doubly-exponential dependence on the parameter. Moreover, it is difficult to find ILP formulations with small dimension for problems whose input contains many objects with varying cost functions, such as in Swap Bribery [4, Challenge #2].

Our contributions. We show that a certain form of ILP, which is closely related to the previously used formulations for Closest String and other problems, can be solved in single-exponential time and in variable dimension. For example, Gramm et al.'s [17] algorithm for Closest String runs in time $2^{2^{O(k \log k)}} \log L$ and has not been improved since 2003, while our algorithm runs in time $k^{O(k^2)} \log L$. Moreover, our algorithm has a strong combinatorial flavor and is based on different notions than are typically encountered in parameterized complexity, most importantly augmenting steps.

As an example of our form of ILP, consider the following ILP formulation of the Closest String problem. We are given k strings $s_1, \ldots, s_k$ of length L that come (after some preprocessing) from alphabet $[k] := \{1, \ldots, k\}$, and an integer d. The goal is to find a string $y \in [k]^L$ such that, for each s_i, the Hamming distance $d_H(y, s_i)$ is at most d, if such y exists. For $i \in [L]$, $(s_1[i], \ldots, s_k[i])$ is the i-th column of the input. Clearly there are at most k^k different column types in the input, and we can represent the input succinctly with multiplicities $b^{\mathbf{f}}$ of each column type $\mathbf{f} \in [k]^k$. Moreover, there are k choices for the output string y in each column. Thus, we can encode the solution by, for each column type $\mathbf{f} \in [k]^k$ and each output character $e \in [k]^k$, describing how many solution columns are of type $(\mathbf{f}, e)$.

This is the basic idea behind the formulation of Gramm et al. [17], as depicted on the left:

$$\begin{array}{rcll} \sum_{e\in[k]}\sum_{\mathbf{f}\in[k]^k} d_H(e,f_j)x_{\mathbf{f},e} & \leq & d & \\ \sum_{e\in[k]} x_{\mathbf{f},e} & = & b^{\mathbf{f}} & \\ x_{\mathbf{f},e} & \geq & 0 & \end{array} \quad \left| \quad \begin{array}{rcll} \sum_{\mathbf{f}\in[k]^k}\sum_{(\mathbf{f}',e)\in[k]^{k+1}} d_H(e,f_j)x^{\mathbf{f}}_{\mathbf{f}',e} & \leq & d & \forall j\in[k] \\ \sum_{(\mathbf{f}',e)\in[k]^{k+1}} x^{\mathbf{f}}_{\mathbf{f}',e} & = & b^{\mathbf{f}} & \forall \mathbf{f}\in[k]^k \\ & & & \forall(\mathbf{f},e)\in[k]^{k+1} \\ x^{\mathbf{f}'}_{\mathbf{f},e} & = & 0 & \forall \mathbf{f}'\neq\mathbf{f}, \forall e\in[k] \\ 0\leq x^{\mathbf{f}}_{\mathbf{f},e} & \leq & b^{\mathbf{f}} & \forall \mathbf{f}\in[k]^k \end{array}$$

Let $(1\ \cdots\ 1) = \mathbf{1}^\intercal$ be a row vector of all ones. Then we can view the above as

$$\begin{array}{cccc|c} D_1 & D_2 & \cdots & D_{k^k} & \leq d \\ \mathbf{1}^\intercal & 0 & \cdots & 0 & = b^1 \\ 0 & \mathbf{1}^\intercal & \cdots & 0 & = b^2 \\ \vdots & \vdots & \ddots & \vdots & = \vdots \\ 0 & 0 & \cdots & \mathbf{1}^\intercal & = b^{k^k} \end{array} \quad \left| \quad \begin{array}{cccc|c} D & D & \cdots & D & \leq d \\ \mathbf{1}^\intercal & 0 & \cdots & 0 & = b^1 \\ 0 & \mathbf{1}^\intercal & \cdots & 0 & = b^2 \\ \vdots & \vdots & \ddots & \vdots & = \vdots \\ 0 & 0 & \cdots & \mathbf{1}^\intercal & = b^{k^k}, \end{array}\right.$$

where $D = (D_1\ D_2\ \ldots\ D_{k^k})$. The formulation on the right is clearly related to the one on the left, but contains "dummy" variables which are always zero. This makes it seem unnatural at first, but notice that it has the nice form

$$\min\left\{f(\mathbf{x}) \mid E^{(n)}\mathbf{x}=\mathbf{b},\ \mathbf{l}\leq\mathbf{x}\leq\mathbf{u},\ \mathbf{x}\in\mathbb{Z}^{nt}\right\},\ \text{where}\ E^{(n)} := \begin{pmatrix} D & D & \cdots & D \\ A & 0 & \cdots & 0 \\ 0 & A & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A \end{pmatrix}. \quad (1)$$

Here, $r, s, t, n \in \mathbb{N}$, $\mathbf{u}, \mathbf{l} \in \mathbb{Z}^{nt}$, $\mathbf{b} \in \mathbb{Z}^{r+ns}$ and $f : \mathbb{Z}^{nt} \to \mathbb{Z}$ is a separable convex function, $E^{(n)}$ is an $(r+ns)\times nt$-matrix, $D \in \mathbb{Z}^{r\times t}$ is an $r\times t$-matrix and $A \in \mathbb{Z}^{s\times t}$ is an $s\times t$-matrix. We call $E^{(n)}$ the *n-fold product of* $E = \binom{D}{A}$. This problem (1) is known as *n-fold integer programming* $(IP)_{E^{(n)},\mathbf{b},\mathbf{l},\mathbf{u},f}$. Building on a dynamic program of Hemmecke, Onn and Romanchuk [19] and a so-called proximity technique of Hemmecke, Köppe and Weismantel [18], Knop and Koutecký [25] prove that:

▶ **Proposition 1** ([25, Theorem 7]). *There is an algorithm that, given $(IP)_{E^{(n)},\mathbf{b},\mathbf{l},\mathbf{u},f}$ encoded with L bits, solves*[1] *it in time $a^{O(trs+t^2s)}\cdot n^3L$, where $a = \max\{\|D\|_\infty, \|A\|_\infty\}$.*

However, since the ILP on the bottom right of the previous page has $t = k^k$, applying Proposition 1 gives no advantage over applying Lenstra to solve the Closest String problem. We overcome this by focusing on a special case with $A = (1\ \cdots\ 1) = \mathbf{1}^\intercal \in \mathbb{Z}^{1\times t}$, $(b^1,\ldots,b^n) \geq \mathbf{0}$, $u^i_j \in \{0, \|\mathbf{b}\|_\infty\}$ for all $i \in [n]$ and $j \in [t]$[2], and $f(\mathbf{x}) = \mathbf{w}^\intercal\mathbf{x}$, i.e., the objective is linear. We denote $f^i(\mathbf{x}^i) = \mathbf{w}^{i\intercal}\mathbf{x}^i$. We call this form *combinatorial n-fold IP*[3], and achieve an exponential speed-up in t:

[1] Given an (IP), we say that to *solve* it is to either (i) declare it infeasible or unbounded or (ii) find a minimizer of it.

[2] More precisely, $\mathbf{x} \geq \mathbf{0}$ and $\mathbf{1}^\intercal\mathbf{x}^i = b^i$ imply $x^i_j \leq b^i$ and thus we just need either $u^i_j = 0$ or $u^i_j \geq b^i$.

[3] We deliberately use the term "n-fold IP" even if our objective is linear, making it an ILP, in order to be consistent with the previous literature [19, 31, 35].

▶ **Theorem 2.** *Let $(IP)_{E^{(n)},\mathbf{b},\mathbf{0},\mathbf{u},\mathbf{w}}$ be a combinatorial n-fold IP instance with $L = \langle \mathbf{b}, \mathbf{0}, \mathbf{u}, \mathbf{w} \rangle$ and $a = \|D\|_\infty$. Then it can be solved in time $t^{O(r)}(ar)^{O(r^2)}n^3L$.*

Observe that, when applicable, our algorithm is not only faster than Lenstra's, but works even if the number n is variable (not parameter).

By applying this result to a few selected problems we obtain exponential improvements in the dependence on the parameter, the length of the input, or both, as presented in Table 1. Statements whose proofs are omitted due to space constraints are marked with $\star$.

Stringology. A typical problem from stringology is to find a string y satisfying certain distance properties with respect to k strings $s_1, \ldots, s_k$. All previous fixed-parameter algorithms for such problems we are aware of for parameter k rely on Lenstra's algorithm, or their complexity status was complexity open (e.g., the complexity of Optimal Consensus [1] was unknown for all $k \geq 4$). Interestingly, Boucher and Wilkie [3] show the counterintuitive fact that Closest String is easier to solve when k is large, which makes the parameterization by k even more significant. Finding an algorithm with run time only single-exponential in k was a repeatedly posed open problem, e.g. [6, Challenge #1] and [2, Problem 7.1]. By applying our result, we close this gap for a wide range of problems.

▶ **Theorem 3** ($\star$). *The problems*

- Closest String, Farthest String, Distinguishing String Selection, Neighbor String, Closest String with Wildcards, Closest to Most Strings, c-HRC *and* Optimal Consensus *are solvable in time $k^{O(k^2)} \log L$, and,*
- d-Mismatch *is solvable in time $k^{O(k^2)}L^2 \log L$,*

where k is the number of input strings, L is their length, and we are assuming that the input is presented succinctly by multiplicities of identical columns.

Computational Social Choice. A typical problem in computational social choice involves an election with voters (V) and candidates (C). A natural and much studied parameter is the number of candidates $|C|$. For a long time, only algorithms double-exponential in $|C|$ were known, and improving upon them was posed as a challenge [4, Challenge #1]. Recently, Knop et al. [27] solved the challenge using Proposition 1. However, Knop et al.'s result has a cubic dependence $O(|V|^3)$ on the number of voters, and the dependence on the number of candidates is still quite large, namely $|C|^{O(|C|^6)}$. We improve their result as follows:

▶ **Theorem 4** ($\star$). $\mathcal{R}$-Swap Bribery *can be solved in time*

- $|C|^{O(|C|^2)}T^3(\log |V| + \log \sigma_{\max})$ *for $\mathcal{R}$ any natural scoring protocol, and,*
- $|C|^{O(|C|^4)}T^3(\log |V| + \log \sigma_{\max})$ *for $\mathcal{R}$ any C1 rule,*

where $T \leq |V|$ is the number of voter types and $\sigma_{\max}$ is the maximum cost of a swap.

Weighted Set Multicover. Bredereck et al. [5] define the Weighted Set Multicover (WSM) problem, which is a significant generalization of the classical Set Cover problem. Their motivation to study WSM was that it captures several problems from computational social choice and optimization problems on graphs [11, 13, 29, implicit in]. Bredereck et al. [5] design an algorithm for WSM that runs in time $2^{2^{O(k \log k)}} \log n$, using Lenstra's algorithm.

Again, applying our result yields an exponential improvement over that of Bredereck et al. [4] both in the dependence on the parameter and the size of the instance:

▶ **Theorem 5.** *There is an algorithm that solves the* Weighted Set Multicover *problem and runs in time $k^{O(k^2)}W^3(\log n + \log w_{\max})$, where k is the size of the universe, n denotes the number of sets, W is the number of different weights and $w_{\max}$ is the maximum weight.*

Table 1 Complexity improvements for a few representative problems.

Problem	Previous best runtime	Our result
Closest String	$2^{2^{O(k \log k)}} \log L$ [17]	$k^{O(k^2)} \log L$
Optimal Consensus	FPT for $k \leq 3$, open for $k \geq 4$ [1]	$k^{O(k^2)} \log L$
Score-Swap Bribery	$2^{2^{O(\lvert C\rvert \log \lvert C\rvert)}} \log \lvert V\rvert$ [9] / $\lvert C\rvert^{O(\lvert C\rvert^6)} \lvert V\rvert^3$ [27]	$\lvert C\rvert^{O(\lvert C\rvert^2)} T^3 \log \lvert V\rvert$, with $T \leq \lvert V\rvert$
C1-Swap Bribery	$2^{2^{O(\lvert C\rvert \log \lvert C\rvert)}} \log \lvert V\rvert$ [9] / $\lvert C\rvert^{O(\lvert C\rvert^6)} \lvert V\rvert^3$ [27]	$\lvert C\rvert^{O(\lvert C\rvert^4)} T^3 \log \lvert V\rvert$, with $T \leq \lvert V\rvert$
Weighted Set Multicover	$2^{2^{O(k \log k)}} n$ [5]	$k^{O(k^2)} \log n$
Huge n-fold IP	FPT with $D = I$ and A totally unimodular	FPT with parameter-sized domains

Huge n-fold IP. Onn [36] introduces a high-multiplicity version of the standard n-fold IP problem (1). It is significant because of its connection to the Bin Packing problem in the case of few item sizes, as studied by Goemans and Rothvoss [16]. Previously, Huge n-fold IP was shown to be fixed-parameter tractable when $D = I$ and A is totally unimodular; using our result, we show that it is also fixed-parameter tractable when D and A are arbitrary, but the size of variable domains is bounded by a parameter.

A summary of our results is given in Table 1; this list is not meant to be exhaustive. In fact, we believe that for any Lenstra-based result in the literature which only achieves double-exponential run times, there is a good chance that it can be sped up using our algorithm. The only significant obstacle seem to be large coefficients in the constraint matrix. We provide further insights and discussion in the full version of the paper [26].

Related work. Our main inspiration are augmentation methods based on Graver bases, especially a fixed-parameter algorithm for n-fold IP of Hemmecke, Onn and Romanchuk [19]. Our result improves the runtime of their algorithm for a special case. All the following related work is orthogonal to ours in either the achieved result, or the parameters used for it.

In fixed dimension, Lenstra's algorithm [30] was generalized for arbitrary convex sets and quasiconvex objectives by Khachiyan and Porkolab [24]. The currently fastest algorithm of this kind is due to Dadush et al. [8]. The first notable fixed-parameter algorithm for a non-convex objective is due to Lokshtanov [32], who shows that optimizing a quadratic function over the integers of a polytope is fixed-parameter tractable if all coefficients are small. Ganian and Ordyniak [14] and Ganian et al. [15] study the complexity of ILP with respect to structural parameters such as treewidth and treedepth, and introduce a new parameter called *torso-width*.

Besides fixed-parameter tractability, there is interest in the (non)existence of kernels of ILPs, which formalize the (im)possibility of various preprocessing procedures. Jansen and Kratsch [22] show that ILPs containing parts with simultaneously bounded treewidth and bounded domains are amenable to kernelization, unlike ILPs containing totally unimodular parts. Kratsch [28] studies the kernelizability of sparse ILPs with small coefficients.

2 Preliminaries

For positive integers m, n we set $[m : n] = \{m, \ldots, n\}$ and $[n] = [1 : n]$. For a graph G we denote by $V(G)$ the set of its vertices. We write vectors in boldface (e.g., $\mathbf{x}, \mathbf{y}$ etc.) and their entries in normal font (e.g., the i-th entry of $\mathbf{x}$ is x_i). Given an matrix $A \in \mathbb{Z}^{m \times n}$, vectors

$\mathbf{b} \in \mathbb{Z}^m$, $\mathbf{l}, \mathbf{u} \in \mathbb{Z}^n$ and a function $f : \mathbb{Z}^n \to \mathbb{Z}$, we denote by $(IP)_{A,\mathbf{b},\mathbf{l},\mathbf{u},f}$ the problem

$$\min \{f(\mathbf{x}) \mid A\mathbf{x} = \mathbf{b}\,,\ \mathbf{l} \le \mathbf{x} \le \mathbf{u}\,,\ \mathbf{x} \in \mathbb{Z}^n\}\,.$$

We say that $\mathbf{x}$ is feasible for $(IP)_{A,\mathbf{b},\mathbf{l},\mathbf{u},f}$ if $A\mathbf{x} = \mathbf{b}$ and $\mathbf{l} \le \mathbf{x} \le \mathbf{u}$. If we want to talk about any such IP, we simply denote it as (IP).

Graver Bases and Augmentation. Let us now introduce Graver bases, how they can be used for optimization, and also the special case of n-fold IPs. For background, we refer to the books of Onn [35] and De Loera et al. [31].

Given two n-dimensional integer vectors $\mathbf{x}$ and $\mathbf{y}$, we say they are *sign-compatible* if they lie in the same orthant, or equivalently, if for each $i \in [n]$, the sign of x_i and y_i is the same. We say $\sum_i \mathbf{g}^i$ is a *sign-compatible sum* if all $\mathbf{g}^i$ are pair-wise sign-compatible. Moreover, we write $\mathbf{y} \sqsubseteq \mathbf{x}$ if $\mathbf{x}$ and $\mathbf{y}$ are sign-compatible and $|y_i| \le |x_i|$ for each $i \in [n]$, and write $\mathbf{y} \sqsubset \mathbf{x}$ if at least one of the inequalities is strict. Clearly, $\sqsubseteq$ imposes a partial order called "conformal order" on n-dimensional vectors. For an integer matrix $A \in \mathbb{Z}^{m\times n}$, its *Graver basis* $\mathcal{G}(A)$ is the set of $\sqsubseteq$-minimal non-zero elements of the *lattice* of A, $\ker_{\mathbb{Z}}(A) = \{\mathbf{z} \in \mathbb{Z}^n \mid A\mathbf{z} = \mathbf{0}\}$. An important property of $\mathcal{G}(A)$ is the following.

▶ **Proposition 6** ([35, Lemma 3.2]). *Every integer vector $\mathbf{x} \neq \mathbf{0}$ with $A\mathbf{x} = \mathbf{0}$ is a sign-compatible sum $\mathbf{x} = \sum_i \mathbf{g}^i$ of Graver basis elements $\mathbf{g}^i \in \mathcal{G}(A)$, with some elements possibly appearing with repetitions.*

Given a feasible solution $\mathbf{x}$ to an (IP), we call $\mathbf{g}$ a *feasible step* if $\mathbf{x} + \mathbf{g}$ is feasible in (IP). Moreover, we call a feasible step $\mathbf{g}$ *augmenting* if $f(\mathbf{x} + \mathbf{g}) < f(\mathbf{x})$. Given a feasible solution $\mathbf{x}$ to (IP), we call a tuple $(\mathbf{g}, \alpha)$ with $\alpha \in \mathbb{Z}$ a *Graver-best step* if $\mathbf{g}$ is an augmenting step and $\forall \tilde{\mathbf{g}} \in \mathcal{G}(A)$ and $\forall \alpha' \in \mathbb{Z}$, $f(\mathbf{x} + \alpha\mathbf{g}) \le f(\mathbf{x} + \alpha'\tilde{\mathbf{g}})$. We call α the *step length*. The *Graver-best augmentation procedure* for an (IP) and a given feasible solution $\mathbf{x}_0$ works as follows:

1. If there is no Graver-best step for $\mathbf{x}_0$, return it as optimal.
2. If a Graver-best step $(\alpha, \mathbf{g})$ for $\mathbf{x}_0$ exists, set $\mathbf{x}_0 := \mathbf{x}_0 + \alpha\mathbf{g}$ and go to 1.

▶ **Proposition 7** ([31, implicit in Theorem 3.4.1]). *Given a feasible solution $\mathbf{x}_0$ and a separable convex function f, the Graver-best augmentation procedure finds an optimum in at most $2n - 2\log M$ steps, where $M = f(\mathbf{x}_0) - f(\mathbf{x}^*)$ and $\mathbf{x}^*$ is any minimizer.*

***n*-fold IP.** The structure of $E^{(n)}$ (in problem (1)) allows us to divide the nt variables of $\mathbf{x}$ into n *bricks* of size t. We use subscripts to index within a brick and superscripts to denote the index of the brick, i.e., x^i_j is the j-th variable of the i-th brick with $j \in [t]$ and $i \in [n]$.

3 Combinatorial n-fold IPs

This section is dedicated to proving Theorem 2. We fix an instance of combinatorial n-fold IP, that is, a tuple $(n, D, \mathbf{b}, \mathbf{u}, \mathbf{w})$.

3.1 Graver complexity of combinatorial n-fold IP

The key property of the n-fold product $E^{(n)}$ is that, for any $n \in \mathbb{N}$, the number of nonzero bricks of any $\mathbf{g} \in \mathcal{G}(E^{(n)})$ is bounded by some constant $g(E)$ called the *Graver complexity of E*. A proof is given for example by Onn [35, Lemma 4.3]; it goes roughly as follows. Consider any $\mathbf{g} \in \mathcal{G}(E^{(n)})$ and take its restriction to its nonzero bricks $\bar{\mathbf{g}}$. By Proposition 6,

each brick $\bar{\mathbf{g}}^j$ can be decomposed into elements from $\mathcal{G}(A)$, giving a vector $\mathbf{h}$ whose bricks are elements of $\mathcal{G}(A)$. Then, consider a compact representation $\mathbf{v}$ of $\mathbf{h}$ by counting how many times each element from $\mathcal{G}(A)$ appears. Since $\mathbf{g} \in \mathcal{G}\big(E^{(n)}\big)$ and $\mathbf{h}$ is a decomposition of its nonzero bricks, we have that $\sum_j D\mathbf{h}^j = 0$. Let G be a matrix with the elements of $\mathcal{G}(A)$ as columns. It is not difficult to show that $\mathbf{v} \in \mathcal{G}(DG)$. Since $\|v\|_1$ is an upper bound on the number of bricks of $\mathbf{h}$ and thus of nonzero bricks of $\mathbf{g}$ and clearly does not depend on n, $g(E) = \max_{v \in \mathcal{G}(DG)} \|v\|_1$ is finite. Let us make precise two observations from this proof.

▶ **Lemma 8** ([20, Lemma 3.1], [35, implicit in proof of Lemma 4.3])**.** *Let* $(\mathbf{g}^1, \ldots, \mathbf{g}^n) \in \mathcal{G}\big(E^{(n)}\big)$. *Then, for all* $i \in [n]$ *there exist vectors* $\mathbf{h}^{i,1}, \ldots, \mathbf{h}^{i,n_i} \in \mathcal{G}(A)$ *such that* $\mathbf{g}^i = \sum_{k=1}^{n_i} \mathbf{h}^{i,k}$, *and* $\sum_{i=1}^{n} n_i \leq g(E)$.

▶ **Lemma 9** ([20, Lemma 6.1], [35, implicit in proof of Lemma 4.3])**.** *Let* $D \in \mathbb{Z}^{r \times t}$, $A \in \mathbb{Z}^{s \times t}$, $G \in \mathbb{Z}^{t \times p}$ *be the matrix whose columns are elements of* $\mathcal{G}(A)$ *and* $p = |\mathcal{G}(A)| \leq \|A\|_\infty^{st}$, *and let* $E = \binom{D}{A}$. *Then* $g(E) \leq \max_{v \in \mathcal{G}(DG)} \|v\|_1 \leq \|A\|_\infty^{st} \cdot (r\|DG\|_\infty)^r$.

Notice that this bound on $g(E)$ is exponential in t. Our goal now is to exploit the fact that the matrix A in a combinatorial n-fold IP is very simple and thus get a better bound.

▶ **Lemma 10.** *Let* $D \in \mathbb{Z}^{r \times t}$, $E = \binom{D}{\mathbf{1}^\intercal}$, *and* $a = \|D\|_\infty$. *Then,* $g(E) \leq t^2(2ra)^r$.

To see this, we will need to understand the structure of $\mathcal{G}(\mathbf{1}^\intercal)$:

▶ **Lemma 11.** *It holds that* $\mathcal{G}(\mathbf{1}^\intercal) = \{\mathbf{g} \mid \mathbf{g}$ *has one* 1 *and one* -1 *and* 0 *otherwise*$\} \subseteq \mathbb{Z}^t$, $|\mathcal{G}(\mathbf{1}^\intercal)| = t(t-1)$, *and for all* $\mathbf{g} \in \mathcal{G}(\mathbf{1}^\intercal)$, $\|\mathbf{g}\|_1 = 2$.

Proof. Observe that the claimed set of vectors is clearly $\sqsubseteq$-minimal in $\ker_{\mathbb{Z}}(\mathbf{1}^\intercal)$. We are left with proving there is no other non-zero $\sqsubseteq$-minimal vector in $\ker_{\mathbb{Z}}(\mathbf{1}^\intercal)$. For contradiction assume there is such a vector $\mathbf{h}$. Since it is non-zero, it must have a positive entry h_i. On the other hand, since $\mathbf{1}^\intercal\mathbf{h} = \mathbf{0}$, it must also have a negative entry h_j. But then $\mathbf{g}$ with $g_i = 1$, $g_j = -1$ and $g_k = 0$ for all $k \notin \{i, j\}$ is $\mathbf{g} \sqsubset \mathbf{h}$, a contradiction. The rest follows. ◀

Proof of Lemma 10. We simply plug into the bound of Lemma 9. By Lemma 11, $p = t(t-1) \leq t^2$. Also, $\|DG\|_\infty \leq \max_{\mathbf{g} \in \mathcal{G}(\mathbf{1}^\intercal)} \{\|D\|_\infty \cdot \|\mathbf{g}\|_1\} \leq 2a$ where the last inequality follows from $\|\mathbf{g}\|_1 = 2$ for all $\mathbf{g} \in \mathcal{G}(\mathbf{1}^\intercal)$, again by Lemma 11. ◀

3.2 Dynamic programming

Hemmecke, Onn and Romanchuk [19] devise a clever dynamic programming algorithm to find augmenting steps for a feasible solution of an n-fold IP. Lemma 8 is key in their approach, as they continue by building a set $Z(E)$ of all sums of at most $g(E)$ elements of $\mathcal{G}(A)$ and then use it to construct the dynamic program. However, such a set $Z(E)$ would clearly be of size exponential in t, which we cannot afford. Our insight here is to build a different dynamic program. In [20], the layers of the dynamic program correspond to partial sums of elements of $\mathcal{G}(A)$; in our dynamic program, the layers will correspond directly to elements of $\mathcal{G}(A)$. This makes it impossible to enforce feasibility with respect to lower and upper bounds in the same way as done in [20]; however, we work around this by exploiting the special structure of $\mathcal{G}(A) = \mathcal{G}(\mathbf{1}^\intercal)$ and simpler lower and upper bounds and enforce them by varying the number of layers of given types. Additionally, we also differ in how we enforce feasibility with respect to the upper rows $(D\ D\ \cdots\ D)$.

Given a brick $i \in [n]$ and $j \in [t]$, let $\mathcal{H}_j^i = \big\{\mathbf{h} \in \mathcal{G}(\mathbf{1}^\intercal) \mid h_j = -1, \mathbf{h} \leq \mathbf{u}^i\big\} \cup \{\mathbf{0} \in \mathbb{Z}^t\}$; here $\mathcal{H}_j^i$ represents the steps which can decrease coordinate x_j^i. Observe that $|\mathcal{H}_j^i| \leq t$. Let $\Sigma(E) = \prod_{j=1}^{r} [-2g(E)a : 2g(E)a]$ be the *signature set of* E whose elements are *signatures*.

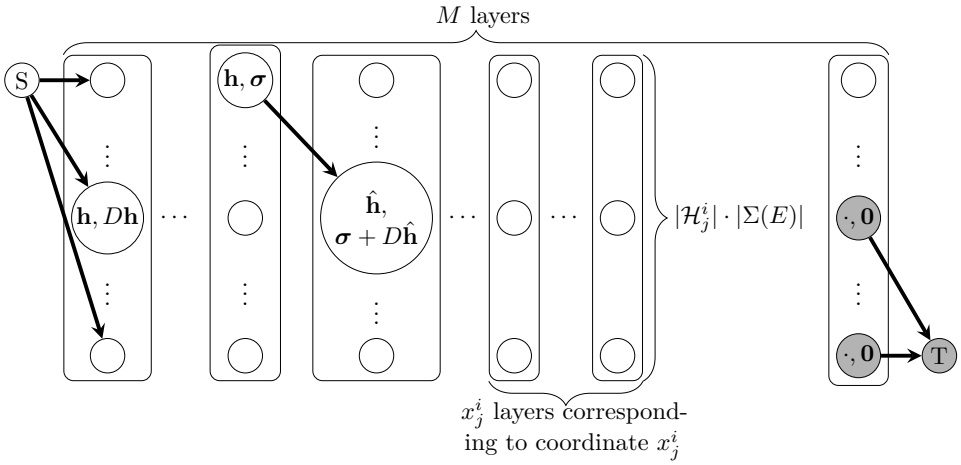


Figure 1 A schema of the augmentation graph $DP(\mathbf{x})$.

Essentially, we will use the signature set to keep track of partial sums of selected elements from $\mathbf{h} \in \mathcal{G}(\mathbf{1}^\intercal)$ to ensure that a resulting vector $\mathbf{g}$ satisfies $D\mathbf{g} = \mathbf{0}$. However, we notice that to ensure $D\mathbf{g} = \mathbf{0}$, it is sufficient to remember the partial sum of elements $D\mathbf{h}$ for $\mathbf{h} \in \mathcal{G}(\mathbf{1}^\intercal)$, thus shrinking them to dimension r. This is another insight which allows us to avoid the exponential dependence on t. Note that $|\Sigma(E)| \leq (1 + 4g(E)a)^r$. Given $\mathbf{x}$ with $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$, we define an index function μ: for $i \in [n]$, $j \in [t]$ and $\ell \in [x_j^i]$ let $\mu(i,j,\ell) := \left(\sum_{k=1}^{i-1} \|\mathbf{x}^k\|_1\right) + \left(\sum_{j'=1}^{j-1} x_{j'}^i\right) + \ell$. In the following text, we consider any vector $\mathbf{x}$ satisfying $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$ even though it would be natural to consider a feasible solution. This is deliberate, as we will later show that we need these claims to hold also for vectors $\mathbf{x}$ derived from feasible solutions which, however, need not be feasible solutions themselves.

▶ **Definition 12** (Augmentation Graph). Given a vector $\mathbf{x}$ with $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$, we define the *augmentation graph* $DP(\mathbf{x})$ to be the following vertex weighted directed layered graph.

There are two distinguished vertices S and T in $DP(\mathbf{x})$, called the *source* and the *sink*. We split the remaining vertices of $DP(\mathbf{x})$ into $M = \|\mathbf{x}\|_1$ layers, denoted $\mathcal{L}(1), \ldots, \mathcal{L}(M)$. With $i \in [n], j \in [t]$ and $\ell \in [\mathbf{x}_j^i]$ we associate the layer $\mathcal{L}(1) = \{(1, \mathbf{h}, D\mathbf{h}) \mid \mathbf{h} \in \mathcal{H}_1^1\}$ if $\mu(i,j,\ell) = 1$ and $\mathcal{L}\big(\mu(i,j,\ell)\big) = \big\{\mu(i,j,\ell)\big\} \times \mathcal{H}_j^i \times \Sigma(E)$ otherwise. Let $L = \max_{\ell=1,\ldots,M} |\mathcal{L}(\ell)|$. A vertex $\big(\mu(i,j,\ell), \mathbf{h}, \boldsymbol{\sigma}\big)$ has weight $f^i(\mathbf{h} + \mathbf{x}^i) - f^i(\mathbf{x}^i)$.

There are the following edges in $DP(\mathbf{x})$. From S to every vertex in the first layer $\mathcal{L}(1)$. Let $u \in \mathcal{L}(\ell)$ and $v \in \mathcal{L}(\ell+1)$ be vertices in consecutive layers with $u = (\ell, \mathbf{h}^\ell, \boldsymbol{\sigma}^\ell)$ and $v = (\ell+1, \mathbf{h}^{\ell+1}, \boldsymbol{\sigma}^{\ell+1})$. If $\boldsymbol{\sigma}^{\ell+1} = \boldsymbol{\sigma}^\ell + D\mathbf{h}^{\ell+1}$, then there is an edge oriented from u to v. Finally, there is an edge from every vertex $u \in \mathcal{L}(M)$ to T if $u = (M, \mathbf{h}, \mathbf{0})$.

Note that by the bounds on $|\mathcal{G}(\mathbf{1}^\intercal)|$ (Lemma 11) and $g(E)$ (Lemma 10), there are at most $L \leq t\left(t^2(2ra)^r\right)^r$ vertices in each layer of $DP(\mathbf{x})$. For an overview of the augmentation graph refer to Fig. 1.

Let P be an S–T path in $DP(\mathbf{x})$ and let $\mathbf{h}^\ell \in \mathcal{G}(\mathbf{1}^\intercal)$ be such that $(\ell, \mathbf{h}^\ell, \boldsymbol{\sigma})$ is its $(\ell+1)$-st vertex. For each $i \in [n]$, let $\mathbf{g}^i = \sum_{j=1}^t \sum_{\ell=1}^{x_j^i} \mathbf{h}^{\mu(i,j,\ell)}$. We say that $\mathbf{h} = (\mathbf{h}^1, \ldots, \mathbf{h}^M)$ is the *P-augmentation vector* and that $\mathbf{g} = (\mathbf{g}^1, \ldots, \mathbf{g}^n)$ is the *compression of* $\mathbf{h}$ (denoted by $\mathbf{g} = \mathbf{h}^\downarrow$). Conversely let $\mathbf{g} \in \mathcal{G}\big(E^{(n)}\big)$ and recall that M is the number of layers of $DP(\mathbf{x})$. By Lemma 8, for all $i \in [n]$ there exist vectors $\mathbf{h}^{i,1}, \ldots, \mathbf{h}^{i,n_i} \in \mathcal{G}(\mathbf{1}^\intercal)$ such that $\mathbf{g}^i = \sum_{k=1}^{n_i} \mathbf{h}^{i,k}$, and $\sum_{i=1}^n n_i \leq g(E)$. For each $i \in [n]$ and $j \in [t]$, let m_j^i be the number of $\mathbf{h}^{i,k}$ with $h_j^{i,k} = -1$. The *expansion* of $\mathbf{g}$ is $\mathbf{h} = (\mathbf{h}^1, \ldots, \mathbf{h}^M)$ defined as follows (we denote this as

$\mathbf{h} = \mathbf{g}^{\uparrow}$). Fix $i \in [n]$ and $j \in [t]$. Assign the distinct m^i_j vectors $\mathbf{h}^{i,k}$ with $h^{i,k}_j = -1$ to $\mathbf{h}^{\mu(i,j,\ell)}$ for $\ell \in [m^i_j]$, and let $\mathbf{h}^{\mu(i,j,\ell)} = \mathbf{0}$ for $\ell \in [m^i_j + 1 : x^i_j]$. Essentially, we pad the vector $\mathbf{h}$ obtained by Lemma 8 with $\mathbf{0}$ bricks to construct an $\mathbf{h} = \mathbf{g}^{\uparrow}$. Also notice that an S–T path P such that $\mathbf{h}$ is a P-augmentation vector can be constructed by choosing appropriate $\boldsymbol{\sigma} \in \Sigma(E)$ for each brick of $\mathbf{h}$.

Let $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$. We say that $\mathbf{g}$ is a *solution of* $DP(\mathbf{x})$ if $\mathbf{0} \leq \mathbf{x} + \mathbf{g} \leq \mathbf{u}$ and there exists an S–T path P with P-augmentation vector $\mathbf{h}$ and $\mathbf{g} = \mathbf{h}^{\downarrow}$; the weight $w(\mathbf{g})$ is then defined as the weight of the path P; note that $w(\mathbf{g}) = f(\mathbf{x} + \mathbf{g}) - f(\mathbf{x})$. A solution $\mathbf{g}$ is called a *minimal solution of* $DP(\mathbf{x})$ if it is a solution of minimal weight. The following lemma relates solutions of $DP(\mathbf{x})$ to potential feasible steps in $\mathcal{G}\big(E^{(n)}\big)$.

▶ **Lemma 13.** *Let $\mathbf{x} \in \mathbb{Z}^{nt}$ satisfy $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$ and let $\mathbf{g}$ be a solution of $DP(\mathbf{x})$. It holds that $\mathbf{0} \leq \mathbf{x} + \mathbf{g} \leq \mathbf{u}$ and $E^{(n)}\mathbf{g} = \mathbf{0}$.*

Proof. It follows from the definition that there are exactly x^i_j layers in which it is possible to select a vector $\mathbf{h}$ such that $h^i_j = -1$. Observe further that all other layers that are derived from the i-th brick can only increase the value of g^i_j. It follows that $\mathbf{x} + \mathbf{g} \geq \mathbf{0}$.

Recall that $u^i_j \in \{0, \|\mathbf{b}\|_\infty\}$. If $u^i_j = 0$, then we have excluded all vectors $\mathbf{h}$ with $h^i_j = 1$ from $\mathcal{H}^i_k$ for all $k \in [t]$. Thus $x^i_j + g^i_j = x^i_j = 0 \leq 0$ as claimed. On the other hand, if $u^i_j = \|\mathbf{b}\|_\infty$, then observe that $\sum_{k=1}^{t} g^i_k = 0$ and because $\mathbf{x} + \mathbf{g} \geq \mathbf{0}$, we conclude that $x^i_j + g^i_j \leq b^i \leq \|\mathbf{b}\|_\infty = u^i_j$.

Let $(\mathbf{h}_k, \boldsymbol{\sigma}_k)$, for each $k \in [M]$, be the vertex from the k-th layer of path P corresponding to $\mathbf{g}^{\uparrow}$. Note that $\boldsymbol{\sigma}_M = \mathbf{0}$. If follows that $\boldsymbol{\sigma}_{\ell+1} = \left(\sum_{k=1}^{\ell} D\mathbf{h}_k\right) + D\mathbf{h}_{\ell+1}$ for all $1 \leq \ell \leq M-1$. Thus $\sum_{k=1}^{M} D\mathbf{h}_k = \mathbf{0}$ and because we have $A\mathbf{h}_k = \mathbf{0}$ from the definition of $\mathcal{H}^i_j$ we conclude that $E^{(n)}\mathbf{g} = \mathbf{0}$. ◀

▶ **Lemma 14.** (⋆) *Let $\mathbf{x} \in \mathbb{Z}^{nt}$ satisfy $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$. Every $\tilde{\mathbf{g}} \in \mathcal{G}\big(E^{(n)}\big)$ with $\mathbf{0} \leq \mathbf{x} + \tilde{\mathbf{g}} \leq \mathbf{u}$ is a solution of $DP(\mathbf{x})$.*

We define the $g(E)$-*truncation* of $\mathbf{x}$ as the vector $\overline{\mathbf{x}}$ given by $\overline{x}^i_j = \min\{x^i_j, g(E)\}$.

▶ **Lemma 15.** (⋆) *Let $\mathbf{x} \in \mathbb{Z}^{nt}$ satisfy $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$. Every $\tilde{\mathbf{g}} \in \mathcal{G}\big(E^{(n)}\big)$ with $\mathbf{0} \leq \mathbf{x} + \tilde{\mathbf{g}} \leq \mathbf{u}$ is a solution of $DP(\overline{\mathbf{x}})$.*

Clearly our goal is then to find the lightest S–T path in the graph $DP(\overline{\mathbf{x}})$. However, there will be edges with negative weights. Still, finding the lightest path can be done in a layer by layer manner (see e.g. [19, Lemma 3.4]) in time $O(|V(DP(\overline{\mathbf{x}}))| \cdot L) = O(\|\overline{\mathbf{x}}\|_1 \cdot L^2)$. The following lemma is then an immediate consequence of Lemmas 14 and 15.

▶ **Lemma 16** (Optimality certification). *Given $\mathbf{x} \in \mathbb{Z}^{nt}$ with $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$, it is possible to find a vector $\mathbf{g}$ such that $E^{(n)}\mathbf{g} = \mathbf{0}$, $\mathbf{0} \leq \mathbf{x} + \mathbf{g} \leq \mathbf{u}$, and $f(\mathbf{x} + \mathbf{g}) < f(\mathbf{x})$, or decide there is none such $\mathbf{g}$, in time $\|\overline{\mathbf{x}}\|_1 \cdot L^2 \leq t^{O(r)}(ar)^{O(r^2)}n$.*

Proof. It follows from Lemma 13 that all solutions of $DP(\mathbf{x})$ fulfill the first two conditions. Observe that if we take $\mathbf{g}$ to be a minimal solution of $DP(\overline{\mathbf{x}})$, then either $f(\mathbf{x}) = f(\mathbf{x} + \mathbf{g})$ or $f(\mathbf{x}) < f(\mathbf{x} + \mathbf{g})$. Due to Lemma 15 the set of solutions of $DP(\overline{\mathbf{x}})$ contains all $\tilde{\mathbf{g}} \in \mathcal{G}\big(E^{(n)}\big)$ with $\mathbf{0} \leq \tilde{\mathbf{g}} \leq \mathbf{u}$. Thus, by Proposition 7, if $f(\mathbf{x}) = f(\mathbf{x} + \mathbf{g})$, no $\mathbf{g}$ satisfying all three conditions exist.

Now simply plug in our bounds on $\|\overline{\mathbf{x}}\|_1$ and L and compute a minimal S–T path:

$$L \leq |\mathcal{G}(\mathbf{1}^{\intercal})| \cdot |\Sigma(E)| \leq t^2 \cdot \left(1 + 4t^2 a(2ra)^r\right)^r \leq t^{O(r)}(ar)^{O(r^2)}$$

is the maximum size of a layer and $\|\overline{\mathbf{x}}\|_1 \leq nt \cdot g(E) \leq nt \cdot t^2(2ra)^r \leq O(t^2)(ar)^{O(r)}n$ is the number of layers. ◀

3.3 Long steps

So far, we are able to *find* an augmenting step in time independent of M; however, each step might only bring an improvement of $O(1)$ and thus possibly many improving steps would be needed. Now, given a step length $\alpha \in \mathbb{N}$, we will show how to find a feasible step $\mathbf{g}$ such that $f(\mathbf{x}+\alpha\mathbf{g}) \leq f(\mathbf{x}+\alpha\tilde{\mathbf{g}})$ for any $\tilde{\mathbf{g}} \in \mathcal{G}\big(E^{(n)}\big)$. Moreover, we will show that there are not too many step lengths that need to be considered in order to find a Graver-best step which, by Proposition 7, leads to a good bound on the required number of steps.

Let $\alpha \in \mathbb{N}$ and let $\mathbf{x}$ with $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$. We define $\mathbf{x}_\alpha$ to be the *α-reduction* of $\mathbf{x}$, $\mathbf{x}_\alpha = \left\lfloor \frac{\mathbf{x}}{\alpha} \right\rfloor$. This operation takes priority over the truncation operation, that is, by $\overline{\mathbf{x}_\alpha}$ we mean the $g(E)$-truncation of vector $\mathbf{x}_\alpha$ (i.e., $\overline{\mathbf{x}_\alpha} = \overline{(\mathbf{x}_\alpha)}$). Note that for large enough α, $DP(\mathbf{x}_\alpha)$ contains only two vertices S and T and no arcs and thus there is no S–T path and no solutions.

▶ **Lemma 17.** (⋆) *Let $\alpha \in \mathbb{N}$ and let $\mathbf{x}$ with $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$. Every $\tilde{\mathbf{g}} \in \mathcal{G}\big(E^{(n)}\big)$ with $\mathbf{0} \leq \mathbf{x}+\alpha\tilde{\mathbf{g}} \leq \mathbf{u}$ is a solution of $DP(\overline{\mathbf{x}_\alpha})$.*

However, a Graver-best step might still be such that its step length α is large and thus we cannot afford to find a minimal solution of $DP(\mathbf{x}_\alpha)$ for all possible step lengths. We need another observation to see that many step lengths need not be considered. Let the *state of* $\mathbf{x}_\alpha$, $\psi(\mathbf{x}_\alpha) \in \{0,1,2\}^{[n]\times[t]}$, be defined by:

- $\psi(\mathbf{x}_\alpha)^i_j = 0$ if $(\mathbf{x}_\alpha)^i_j = 0$,
- $\psi(\mathbf{x}_\alpha)^i_j = 1$ if $1 \leq (\mathbf{x}_\alpha)^i_j < g(E)$, and,
- $\psi(\mathbf{x}_\alpha)^i_j = 2$ if $(\mathbf{x}_\alpha)^i_j \geq g(E)$.

Given a feasible solution $\mathbf{x}$, we call a step length α *interesting* if $\overline{\mathbf{x}_\alpha} \neq \overline{\mathbf{x}_{\alpha+1}}$ and *boring* otherwise. Moreover, α is *irrelevant* if there is no Graver-best step with step length α.

▶ **Lemma 18.** (⋆) *If α is boring, then it is irrelevant.*

▶ **Definition 19** (Candidate step lengths Γ)**.** Let Γ be a set of candidate step lengths constructed iteratively as follows:

Input: vector $\mathbf{x}$ with $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$ and $g(E)$
Computes: set of candidate steps Γ
$\Gamma \leftarrow \{1\}$ and $\gamma \leftarrow 2$
while $\mathbf{x}_\gamma > \mathbf{0}$ **do**
  foreach i,j *with* $\psi(\mathbf{x}_\gamma)^i_j = 1$ **do**
    $\Gamma^i_j \leftarrow \left\{\left(k, \lfloor x^i_j/k \rfloor\right) \mid k \in \mathbb{N}, 0 < \lfloor x^i_j/k \rfloor < (\mathbf{x}_{\gamma-1})^i_j\right\}$
    $\gamma^i_j \leftarrow k$ such that $(k,q) \in \Gamma^i_j$, q is maximal, and secondary to this k also maximal
  $\tilde{\gamma}_1 \leftarrow \min\left\{\gamma^i_j \mid \psi(\mathbf{x}_\gamma)^i_j = 1\right\}$
  $\tilde{\gamma}_2 \leftarrow \min\left\{\left\lfloor \frac{x^i_j}{g(E)} \right\rfloor \mid \psi(\mathbf{x}_\gamma)^i_j = 2\right\}$
  add $\min\{\tilde{\gamma}_1, \tilde{\gamma}_2\}$ to Γ
  $\gamma \leftarrow \max\Gamma + 1$
return Γ

▶ **Lemma 20.** *If α is the step length of a Graver-best step, then $\alpha \in \Gamma$.*

Proof. We will prove that Γ contains all interesting step lengths. Consider an $\alpha \notin \Gamma$. Either $\mathbf{x}_\alpha = \mathbf{0}$ and clearly in that case $DP(\mathbf{x}_\alpha)$ does not yield an augmenting step since it has no layers and thus no weighted vertices, and thus α is irrelevant.

Otherwise, take $\gamma := \min\{\gamma' \mid \gamma' \in \Gamma, \gamma' \geq \alpha\}$. Because of the minimality of γ with respect to all of the $\min\{\cdot\}$ clauses of the algorithm of Definition 19, we have that $\overline{\mathbf{x}_\alpha} = \overline{\mathbf{x}_\gamma}$ and thus α is boring and by Lemma 18 irrelevant.

Since Γ contains all remaining step lengths, it also contains all interesting steps and must contain the step length for any Graver-best step. ◀

▶ **Lemma 21.** *$|\Gamma| \leq O(nt \cdot g(E))$ and Γ can be constructed in time $O(|\Gamma| \cdot \log \|\mathbf{x}\|_\infty)$.*

Proof. Fix a coordinate x^i_j of $\mathbf{x}$ and consider a run of the algorithm of Definition 19. If $x^i_j > g(E)$, $\tilde{\gamma}_2 := \left\lfloor \frac{x^i_j}{g(E)} \right\rfloor$ is added to Γ at some point. For every $\gamma > \tilde{\gamma}_2$ we have that $(\mathbf{x}_\gamma)^i_j < g(E)$ and thus we need not consider the $\min\{\cdot\}$ clause for $\psi(\mathbf{x}_\gamma)^i_j = 2$.

Consider a step of the algorithm which adds $\tilde{\gamma}_1$, and observe that $\tilde{\gamma}_1$ is chosen such that $(\mathbf{x}_{\tilde{\gamma}_1})^i_j > (\mathbf{x}_{\tilde{\gamma}_1+1})^i_j$. But since $(\mathbf{x}_{\tilde{\gamma}_2+1})^i_j < g(E)$, such situation can occur at most $g(E)$ times.

Thus we have added at most $O(g(E))$ different step lengths to Γ per coordinate, $O(nt \cdot g(E))$ step lengths in total.

Regarding the time it takes to construct Γ, we perform $O(|\Gamma|)$ arithmetic operations, and since we are dealing with numbers of size at most $\|\mathbf{x}\|_\infty$, each operation takes time $O(\log \|\mathbf{x}\|_\infty)$, concluding the proof. ◀

▶ **Lemma 22** (Graver-best computation)**.** *Given a feasible solution $\mathbf{x}$ of a combinatorial n-fold IP, in time $t^{O(r)}(ar)^{O(r^2)}n^2$ one can either find a Graver-best step $(\alpha, \mathbf{g})$ or decide that none exists.*

Proof. For $\gamma \in \Gamma$ let $\mathbf{g}^\gamma$ be a minimal solution of $DP(\overline{\mathbf{x}_\gamma})$ and let $\alpha := \arg\min_{\gamma \in \Gamma} \left(f(\mathbf{x} + \gamma \mathbf{g}^\gamma)\right)$. Finally, let $\mathbf{g} := \mathbf{g}^\alpha$. Then we claim that $(\alpha, \mathbf{g})$ is a Graver-best step.

By Lemma 17 for all $\tilde{\mathbf{g}} \in \mathcal{G}\left(E^{(n)}\right)$ it holds that $f(\mathbf{x} + \alpha \mathbf{g}) \leq f(\mathbf{x} + \alpha \tilde{\mathbf{g}})$. Moreover, by Lemma 20, if there exists a Graver-best step with step length γ, then $\gamma \in \Gamma$, and thus by the construction of α, $(\alpha, \mathbf{g})$ is Graver-best step.

Regarding the time complexity, to obtain $\mathbf{g}$ we need to solve $DP(\overline{\mathbf{x}_\gamma})$ for each $\gamma \in \Gamma$ by Lemma 16, requiring time $|\Gamma| \cdot t^{O(r)}(ar)^{O(r^2)}n \leq t^{O(r)}(ar)^{O(r^2)}n^2$. ◀

3.4 Finishing the proof

Proof of Theorem 2. In order to prove Theorem 2 we need to put the pieces together. First, let us assume that we have an initial feasible solution $\mathbf{x}_0$. In order to reach the optimum, by Proposition 7 we need to make at most $(2nt - 2) \cdot O(L)$ Graver-best steps, where $L = \langle \mathbf{b}, \mathbf{0}, \mathbf{u}, \mathbf{w} \rangle$; this is because $O(L)$ is an upper bound on $f(\mathbf{x}^0) - f(\mathbf{x}^*)$ for some minimum $\mathbf{x}^*$. By Lemma 22, it takes time $t^{O(r)}(ar)^{O(r^2)}n^2$ to find a Graver-best step.

Now we are left with the task of finding a feasible solution. We follow along the lines of [19, Lemma 3.8] and solve an auxiliary combinatorial n-fold IP given by the bimatrix $\hat{E} = \begin{pmatrix} \hat{D} \\ \hat{A} \end{pmatrix}$ with $\hat{D} := (D\ I_r\ -I_r\ \mathbf{0})$ and $\hat{A} := (A\ \mathbf{1}_{2r+1}) = \mathbf{1}^\intercal \in \mathbb{Z}^{t+2r+1}$, where I_r is the identity matrix of dimension r, $\mathbf{0}$ is a column vector of length r and $\mathbf{1}_{2r+1}$ is the vector of all 1s of length $2r + 1$.

The variables $\hat{\mathbf{x}}$ of this problem have a natural partition into nt variables $\mathbf{x}$ corresponding to the original problem and $n(2r + 1)$ new auxiliary variables $\tilde{\mathbf{x}}$. Keep the original lower and upper bounds on $\mathbf{x}$ and introduce a lower bound 0 and upper bound $\|\mathbf{b}\|_\infty$ on each auxiliary variable. Finally, let the new linear objective $\hat{\mathbf{w}}^\intercal \hat{\mathbf{x}}$ be the sum of the auxiliary

variables. Observe that it is easy to construct an initial feasible solution by setting $\mathbf{x} = \mathbf{0}$ and computing $\tilde{\mathbf{x}}$ accordingly, as $\tilde{\mathbf{x}}$ serve the role of slack variables.

Then, applying the algorithm described previously either finds a solution with objective value 0, implying $\tilde{\mathbf{x}} = \mathbf{0}$, and thus $\mathbf{x}$ is feasible for the original problem, or no such solution exists, meaning that the original problem is infeasible. ◀

4 An Application to Weighted Set Multicover

In applications, it is practical to use combinatorial n-fold IP formulations which contain inequalities. Given an n-fold IP (in particular a combinatorial n-fold IP), we call the upper rows $(D\ D \cdots\ D)\mathbf{x} = \mathbf{b}^0$ *globally uniform constraints*, and the lower rows $A\mathbf{x}^i = \mathbf{b}^i$, for all $i \in [n]$, *locally uniform constraints*. In the full version [26] we show that introducing inequalities into a combinatorial n-fold IP is possible, however we need a slightly different approach than in a standard n-fold IP to keep the rigid format of a combinatorial n-fold IP.

Weighted Set Multicover. We demonstrate Theorem 2 on the following problem:

Weighted Set Multicover

Input: A universe of size k, $U = [k]$, a set system represented by a multiset $\mathcal{F} = \{F_1, \ldots, F_n\} \subseteq 2^U$, weights $w_1, \ldots, w_n \in \mathbb{N}$, demands $d_1, \ldots, d_k \in \mathbb{N}$.

Find: A multisubset $\mathcal{F}' \subseteq \mathcal{F}$ minimizing $\sum_{F_i \in \mathcal{F}'} w_i$ and satisfying $\left|\{i \mid F_i \in \mathcal{F}', j \in F_i\}\right| \geq d_j$ for all $j \in [k]$.

Proof of Theorem 5. Observe that since W is the number of different weights, and there can be at most 2^k different sets $F \in 2^U$, each pair (F, w) on the input is of one of $T \leq W2^k$ different types; let $n_1, \ldots, n_T \in \mathbb{N}$ be a succinct representation of the instance.

We shall construct a combinatorial n-fold IP to solve the problem. Let $x^\tau_{\mathbf{f}}$ for each $\mathbf{f} \in 2^U$ and each $\tau \in [T]$ be a variable. Let $u^\tau_{\mathbf{f}} = 0$ for each $\mathbf{f} \in 2^U$ such that $\mathbf{f} \neq F^\tau$, and let $u^\tau_{\mathbf{f}} = \max n_\tau$ for $\mathbf{f} = F^\tau$. The variable $x^\tau_{\mathbf{f}}$ with $\mathbf{f} = F^\tau$ represents the number of sets of type τ in the solution. The formulation is straightforward and reads

$$\begin{aligned}
\min \quad & \sum_{\tau=1}^{T} \sum_{\mathbf{f} \in 2^U} w^\tau x^\tau_{\mathbf{f}} \\
\text{s.t.} \quad & \sum_{\tau=1}^{T} \sum_{\mathbf{f} \in 2^U} f_i x^\tau_{\mathbf{f}} \geq d_i, \qquad \text{for all } i \in [k] \\
& \sum_{\mathbf{f} \in 2^U} x^\tau_{\mathbf{f}} \leq n_\tau \qquad \text{for all } \tau \in [T];
\end{aligned}$$

note that f_i is 1 if $i \in \mathbf{f}$ and 0 otherwise. Let us determine the parameters $\hat{a}, \hat{r}, \hat{t}, \hat{n}$ and $\hat{L}$ of this combinatorial n-fold IP instance. Clearly, the largest coefficient $\hat{a}$ is 1, the number of globally uniform constraints $\hat{r}$ is k, the number of variables per brick $\hat{t}$ is 2^k, the number of bricks $\hat{n}$ is T, and the length of the input $\hat{L}$ is at most $\log n + \log w_{\max}$. ◀

5 Open problems

Can our result be extended to minimizing a separable convex function f? Is Huge n-fold IP fixed-parameter tractable for parameters r, s, t and a? It is not difficult to see that optimality certification is fixed-parameter tractable using ideas similar to Onn [36]; however, one possibly needs exponentially (in the input size) many augmenting steps.

For most of our applications, complexity lower bounds are not known to us. Our algorithms yield complexity upper bounds of $k^{O(k^2)}$ on the dependence on parameter k for various problems, such as Closest String, Weighted Set Multicover, Score-Swap Bribery or even Makespan Minimization [25]. Is this just a common feature of our algorithm, or are there hidden connections between some of these problems? And what are their actual complexities? All we know so far is a trivial ETH-based $2^{o(k)}$ lower bound for Closest String based on its reduction from Satisfiability [12].

References

1. Amihood Amir, Gad M. Landau, Joong Chae Na, Heejin Park, Kunsoo Park, and Jeong Seop Sim. Efficient algorithms for consensus string problems minimizing both distance sum and radius. *Theoret. Comput. Sci.*, 412(39):5239–5246, 2011.
2. Liliana Félix Avila, Alina Garcıa, Marıa José Serna, and Dimitrios M Thilikos. A list of parameterized problems in bioinformatics. Technical report, Technical University of Catalonia, 2006. Technical report LSI-06-24-R.
3. Christina Boucher and Kathleen Wilkie. Why large closest string instances are easy to solve in practice. In *Proc. SPIRE 2010*, volume 6393 of *Lecture Notes Comput. Sci.*, pages 106–117, 2010.
4. Robert Bredereck, Jiehua Chen, Piotr Faliszewski, Jiong Guo, Rolf Niedermeier, and Gerhard J. Woeginger. Parameterized algorithmics for computational social choice: Nine research challenges. *Tsinghua Sci. Tech.*, 19(4):358–373, 2014.
5. Robert Bredereck, Piotr Faliszewski, Rolf Niedermeier, Piotr Skowron, and Nimrod Talmon. Elections with few candidates: Prices, weights, and covering problems. In *Proc. ADT 2015*, volume 9346 of *Lecture Notes Comput. Sci.*, pages 414–431, 2015.
6. Laurent Bulteau, Falk Hüffner, Christian Komusiewicz, and Rolf Niedermeier. Multivariate algorithmics for NP-hard string problems. *Bulletin of the EATCS*, 114, 2014.
7. Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer, 2015.
8. Daniel Dadush, Chris Peikert, and Santosh Vempala. Enumerative lattice algorithms in any norm via M-ellipsoid coverings. In *Proc. FOCS 2011*, pages 580–589. 2011.
9. Britta Dorn and Ildikó Schlotter. Multivariate complexity analysis of swap bribery. *Algorithmica*, 64(1):126–151, 2012.
10. Michael R. Fellows, Daniel Lokshtanov, Neeldhara Misra, Frances A. Rosamond, and Saket Saurabh. Graph layout problems parameterized by vertex cover. In *Proc. ISAAC 2008*, volume 5369 of *Lecture Notes Comput. Sci.*, pages 294–305. Springer, Berlin, 2008.
11. Jiří Fiala, Tomáš Gavenčiak, Dušan Knop, Martin Koutecký, and Jan Kratochvíl. Parameterized complexity of distance labeling and uniform channel assignment problems. *Discrete Appl. Math.*, 2017. to appear.
12. M. Frances and A. Litman. On covering problems of codes. *Theory Comput. Syst.*, 30(2):113–119, 1997.
13. Jakub Gajarský, Michael Lampis, and Sebastian Ordyniak. Parameterized algorithms for modular-width. In *Proc. IPEC 2013*, volume 8246 of *Lecture Notes Comput. Sci.*, pages 163–176. 2013.
14. Robert Ganian and Sebastian Ordyniak. The complexity landscape of decompositional parameters for ILP. In *Proc. AAAI 2016*, pages 710–716, 2016.
15. Robert Ganian, Sebastian Ordyniak, and M. S. Ramanujan. Going beyond primal treewidth for (M)ILP. In *Proc. AAAI 2017*, pages 815–821, 2017.
16. Michel X. Goemans and Thomas Rothvoß. Polynomiality for bin packing with a constant number of item types. In *Proc. SODA 2014*, pages 830–839. ACM, New York, 2014.

17 Jens Gramm, Rolf Niedermeier, and Peter Rossmanith. Fixed-parameter algorithms for closest string and related problems. *Algorithmica*, 37(1):25–42, 2003.

18 Raymond Hemmecke, Matthias Köppe, and Robert Weismantel. Graver basis and proximity techniques for block-structured separable convex integer minimization problems. *Math. Program.*, 145(1-2, Ser. A):1–18, 2014.

19 Raymond Hemmecke, Shmuel Onn, and Lyubov Romanchuk. n-fold integer programming in cubic time. *Math. Program.*, 137(1-2, Ser. A):325–341, 2013.

20 Raymond Hemmecke, Shmuel Onn, and Robert Weismantel. A polynomial oracle-time algorithm for convex integer minimization. *Math. Program.*, 126(1, Ser. A):97–117, 2011.

21 Danny Hermelin and Liat Rozenberg. Parameterized complexity analysis for the closest string with wildcards problem. *Theoret. Comput. Sci.*, 600:11–18, 2015.

22 Bart M. P. Jansen and Stefan Kratsch. A structural approach to kernels for ILPs: Treewidth and total unimodularity. In *Proc. ESA 2015*, volume 9294 of *Lecture Notes Comput. Sci.*, pages 779–791, 2015.

23 Ravi Kannan. Minkowski's convex body theorem and integer programming. *Math. Oper. Res.*, 12(3):415–440, 1987.

24 Leonid Khachiyan and Lorant Porkolab. Integer optimization on convex semialgebraic sets. *Discrete Comput. Geom.*, 23(2):207–224, 2000.

25 Dušan Knop and Martin Koutecký. Scheduling meets n-fold integer programming. *Journal of Scheduling*, page to appear, 2017.

26 Dušan Knop, Martin Koutecký, and Matthias Mnich. Combinatorial n-fold integer programming and applications. Technical report, 2017. URL: `https://arxiv.org/abs/1705.08657`.

27 Dušan Knop, Martin Koutecký, and Matthias Mnich. Voting and bribing in single-exponential time. In *Proc. STACS 2017*, volume 66 of *Leibniz Int. Proc. Informatics*, pages 46:1–46:14, 2017.

28 Stefan Kratsch. On polynomial kernels for sparse integer linear programs. *J. Comput. System Sci.*, 82(5):758–766, 2016.

29 Michael Lampis. Algorithmic meta-theorems for restrictions of treewidth. *Algorithmica*, 64(1):19–37, 2012.

30 Hendrik W. Lenstra, Jr. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

31 Jesus A. De Loera, Raymond Hemmecke, and Matthias Köppe. *Algebraic and Geometric Ideas in the Theory of Discrete Optimization*, volume 14 of *MOS-SIAM Series on Optimization*. SIAM, 2013.

32 Daniel Lokshtanov. Parameterized integer quadratic programming: Variables and coefficients. Technical report, 2015. `http://arxiv.org/abs/1511.00310`.

33 Matthias Mnich and Andreas Wiese. Scheduling and fixed-parameter tractability. *Math. Program.*, 154(1-2, Ser. B):533–562, 2015.

34 Rolf Niedermeier. Ubiquitous parameterization—invitation to fixed-parameter algorithms. In *Proc. MFCS 2004*, volume 3153 of *Lecture Notes in Comput. Sci.*, pages 84–103. Springer, Berlin, 2004.

35 Shmuel Onn. Nonlinear discrete optimization. *Zurich Lectures in Advanced Mathematics, European Mathematical Society*, 2010.

36 Shmuel Onn. Huge multiway table problems. *Discrete Optim.*, 14:72–77, 2014.

Local Search Algorithms for the Maximum Carpool Matching Problem

Gilad Kutiel[1] **and Dror Rawitz**[*2]

1 **Department of Computer Science, Technion, Haifa, Israel**
gkutiel@cs.technion.ac.il
2 **Faculty of Engineering, Bar Ilan University, Ramat Gan, Israel**
dror.rawitz@biu.ac.il

Abstract

The Maximum Carpool Matching problem is a *star packing* problem in directed graphs. Formally, given a directed graph $G = (V, A)$, a capacity function $c : V \to \mathbb{N}$, and a weight function $w : A \to \mathbb{R}^+$, a *carpool matching* is a subset of arcs, $M \subseteq A$, such that every $v \in V$ satisfies:

(i) $d_M^{\text{in}}(v) \cdot d_M^{\text{out}}(v) = 0$,
(ii) $d_M^{\text{in}}(v) \leq c(v)$, and
(iii) $d_M^{\text{out}}(v) \leq 1$.

A vertex v for which $d_M^{\text{out}}(v) = 1$ is a *passenger*, and a vertex for which $d_M^{\text{out}}(v) = 0$ is a *driver* who has $d_M^{\text{in}}(v)$ passengers. In the Maximum Carpool Matching problem the goal is to find a carpool matching M of maximum total weight. The problem arises when designing an online carpool service, such as Zimride [4], which tries to connect between users based on a similarity function. The problem is known to be NP-hard, even in the unweighted and uncapacitated case. The Maximum Group Carpool Matching problem, is an extension of Maximum Carpool Matching where each vertex represents an unsplittable group of passengers. Formally, each vertex $u \in V$ has a size $s(u) \in \mathbb{N}$, and the constraint $d_M^{\text{in}}(v) \leq c(v)$ is replaced with $\sum_{u:(u,v)\in M} s(u) \leq c(v)$.

We show that Maximum Carpool Matching can be formulated as an unconstrained submodular maximization problem, thus it admits a $\frac{1}{2}$-approximation algorithm. We show that the same formulation does not work for Maximum Group Carpool Matching, nevertheless, we present a local search $(\frac{1}{2} - \varepsilon)$-approximation algorithm for Maximum Group Carpool Matching. For the unweighted variant of both problems when the maximum possible capacity, $c_{\max}$, is bounded by a constant, we provide a local search $(\frac{1}{2} + \frac{1}{2c_{\max}} - \varepsilon)$-approximation algorithm. We also show that the problem is APX-hard, even if the maximum degree and $c_{\max}$ are at most 3.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases approximation algorithms, local search, star packing, submodular maximization

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.55

1 Introduction

As traveling costs become higher and parking becomes sparse it is only natural to share rides or to *carpool*. Originally, carpooling was an arrangement among a group of people by which they take turns driving the others to and from a designated location. However,

* Supported in part by the Israel Science Foundation (grant no. 497/14).

taking turns is not essential, instead passengers can share the cost of the ride with the driver. Carpooling has social advantages other than reducing the costs: it reduces fuel consumption and road congestion and frees parking space. While in the past carpooling was usually a fixed arrangement between friends or neighbors, the emergence of social networks has made carpooling more dynamic and wide scale. These days applications like Zimride [4], BlaBlaCar [1], Moovit [2] and even Waze [3] are matching passengers to drivers.

The matching process of passengers to drivers entails more than matching the route. Passenger satisfaction also needs to be taken into account. Given several riding options (including taking their own car), passengers have preferences. For example, a passenger may prefer to ride with a co-worker or a friend. She may have an opinion on a driver that she rode with in the past. She may prefer a non-smoker, someone who shares her taste in music, or someone who is recommended by others. Moreover, the matching process may take into account driver preferences. For instance, we would like to minimize the extra distance that a driver has to take. Preferences may also be computed using past information. Knapen et al. [16] described an automatic service to match commuting trips. Users of the service register their personal profile and a set of periodically recurring trips, and the service advises registered candidates on how to combine their commuting trips by carpooling. The service estimates the probability that a person a traveling in person's b car will be satisfied by the trip. This is done based on personal information and feedback from users on past rides.

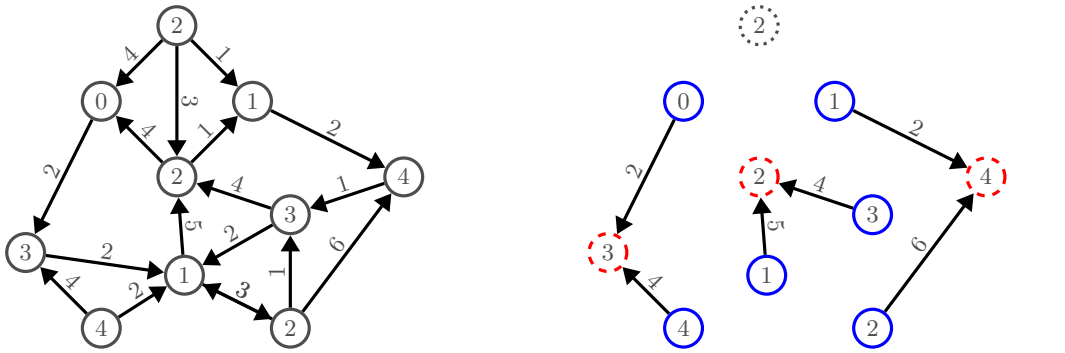
In this paper we assume that potential passenger-driver satisfactions are given as input and the goal is to compute an assignment of passengers to drivers so as to maximize the global satisfaction. More formally, we are given a directed graph $G = (V, A)$, where each vertex $v \in V$ corresponds to a user of the service, and an arc (u, v) exists if the user corresponding to vertex u is willing to commute with the user corresponding to vertex v. We are given a capacity function $c : V \to \mathbb{N}$ which bounds the number of passengers each user can drive if she is selected as a driver. A non-negative weight function $w : A \to \mathbb{R}^+$ is used to model the amount of satisfaction $w(u, v)$ of assigning u to v. If $(u, v) \in A$ implies that $(v, u) \in A$ and $w(u, v) = w(v, u)$, the instance is *undirected*. If $w(v, u) = 1$, for every $(v, u) \in A$, then the instance is *unweighted*. If $c(v) = \deg(v)$, for every v, then the instance is *uncapacitated*.

Given a directed graph G and a subset $M \subseteq A$, define $d_M^{\text{in}}(v) \triangleq |\{u : (u, v) \in M\}|$ and $d_M^{\text{out}}(v) \triangleq |\{u : (v, u) \in M\}|$. A feasible *carpool matching* is a subset of arcs, $M \subseteq A$, such that every $v \in V$ satisfies: (i) $d_M^{\text{in}}(v) \cdot d_M^{\text{out}}(v) = 0$, (ii) $d_M^{\text{in}}(v) \leq c(v)$, and (iii) $d_M^{\text{out}}(v) \leq 1$. A feasible carpool matching M partitions V as follows:

$$P_M \triangleq \left\{v : d_M^{\text{out}}(v) = 1\right\} \quad D_M \triangleq \left\{v : d_M^{\text{in}}(v) \geq 1\right\} \quad Z_M \triangleq \left\{v : d_M^{\text{out}}(v) = d_M^{\text{in}}(c) = 0\right\}$$

where P_M is the set of *passengers*, D_M is the set of *active drivers*, and Z_M is the set of *solo drivers*. In the Maximum Carpool Matching problem the goal is to find a matching M of maximum total weight, namely to maximize $w(M) \triangleq \sum_{(v,u)\in M} w(v, u)$. In other words, the Maximum Carpool Matching problem is about finding a set of (directed toward the center) vertex disjoint stars that maximizes the total weight of the arcs. Figure 1 contains an example of a Maximum Carpool Matching instance. Note that in the unweighted case the goal is to find a carpool matching M that maximizes $|P_M|$. Moreover, observe that if G is undirected, $D_M \cup Z_M$ is a *dominating set*. Hence, in this case, an optimal carpool matching induces an optimal dominating set and vice versa. Since Minimum Dominating Set is NP-hard, it follows that Maximum Carpool Matching is NP-hard even if the instance is undirected, unweigthed, and uncapacitated.

We also consider an extension of Maximum Carpool Matching, called Maximum Group Carpool Matching, in which each vertex represents a group of passengers, and each group may have a different size. Such a group may represent a family or two friends



(a) An instance containing a directed graph with capacities on the vertices and weights on the arcs.

(b) A feasible carpool matching with total weight of 23. P_M is the set of blue vertices, and D_M is the set of red, dashed vertices, and Z_M contains only the dotted, black vertex.

Figure 1 A Maximum Carpool Matching example.

traveling together. Formally, each vertex $u \in V$ has a size $s(u) \in \mathbb{N}$, and the constraint $d_M^{\text{in}}(v) \leq c(v)$ is replaced with the constraint $\sum_{u:(u,v)\in M} s(u) \leq c(v)$. Notice that Knapsack is the special case where only arcs directed at a single vertex have non-zero (integral) weights.

Related work. Agatz et al. [5] outlined the optimization challenges that arise when developing technology to support ride-sharing and survey the related operations research models in the academic literature. Hartman et al. [15] designed several heuristic algorithms for the Maximum Carpool Matching problem and compared their performance on real data. Other heuristic algorithms were developed by Knapen et al. [17]. Hartman [14] proved that the Maximum Carpool Matching problem is NP-hard even in the case where the weight function is binary and $c(v) \leq 2$ for every $v \in V$. In addition, Hartman presented a natural integer linear program and showed that if the set of drivers is known, then an optimal assignment of passengers to drivers can be found in polynomial time using a reduction to Network Flow (see also [18].) Kutiel [18] presented a $\frac{1}{3}$-approximation algorithm for Maximum Carpool Matching that is based on a Minimum Cost Flow computation and a local search $\frac{1}{2}$-approximation algorithm for the unweighted variant of Maximum Carpool Matching. The latter starts with an empty matching and tries to improve the matching by turning a single passenger into a driver.

Nguyen et al. [19] considered the Spanning Star Forest problem. A *star forest* is a graph consisting of vertex-disjoint star graphs. In the Spanning Star Forest problem, we are given an undirected graph G, and the goal is to find a spanning subgraph which is a star forest that maximizes the weight of edges that are covered by the star forest. Notice that this problem is equivalent to Maximum Carpool Matching on undirected and uncapacitated instances. We also note that if all weights leaving a vertex are the same, then the instance is referred to as vertex-weighted. Nguyen et al. [19] provided a PTAS for unweighted planner graphs and a polynomial-time $\frac{3}{5}$-approximation algorithm for unweighted graphs. They gave an exact optimization algorithm for weighted trees, and used it on a maximum spanning tree of the input graph to obtain a $\frac{1}{2}$-approximation algorithm for weighted graphs. They also shows that it is NP-hard to approximate unweighted Spanning Star Forest within a ratio of $\frac{259}{260} + \varepsilon$, for any $\varepsilon > 0$. Chen et al. [13] improved the approximation ratio for unweighted graphs from $\frac{3}{5}$ to 0.71 and gave a 0.64-approximation algorithm for vertex weighted graphs.

They also showed that the edge- and vertex-weighted problem cannot be approximated to within a factor of $\frac{19}{20}+\varepsilon$, and $\frac{31}{32}+\varepsilon$, resp., for any $\varepsilon > 0$, assuming that P $\neq$ NP. Chakrabarty and Goel [11] improved the lower bounds to $\frac{10}{11}+\varepsilon$ and $\frac{13}{14}$.

Athanassopoulos et al. [7] improved the ratio for the unweighted case to $\frac{193}{240} \approx 0.804$. They considered a natural family of *local search* algorithms for Spanning Star Forest. Such an algorithm starts with the solution where all vertices are star centers. Then, it repeatedly tries to turn $t \leq k$ from leaves to centers and $t+1$ centers to leaves. A change is made if it results in a feasible solution, namely if each leave is adjacent to at least one center. The algorithm terminates when such changes are no longer possible. Athanassopoulos et al. [7] showed that, for any k and $\varepsilon \in (0, \frac{1}{2(k+2)}]$, there exists an instance G and a local optima whose size is smaller than $(\frac{1}{2}+\varepsilon)$OPT, where OPT is the size of the optimal spanning star forest. We note that, for a given k, the construction of the above result requires that the maximum degree of G is at least $2(k+2)$. Hence, this result does not hold in graphs with maximum degree Δ.

Arkin et al. [6] considered the Maximum Capacitated Star Packing problem. In this problem the input consists of a complete undirected graph with non-negative edge weights and a capacity vector $c = \{c_1, \ldots, c_p\}$, where $\sum_{i=1}^{p} c_i = |V| - p$. The goal is to find a set of vertex-disjoint stars in G of size $c_1, \ldots, c_p$ of maximum total weight. Arkin et al. [6] provided a local search algorithm whose approximation ratio is $\frac{1}{3}$, and a matching-based $\frac{1}{2}$-approximation algorithm for the case where edge weights satisfy the triangle inequality.

Bar-Noy et al. [8] considered the Minimum 2-Path Partition problem. In this problem the input is a complete graph on $3k$ vertices with non-negative edge weights, and the goal is to partition the graph into disjoint paths of length 2. This problem is the special case of the undirected carpool matching where $c(v) = 2$, for every $v \in V$. They presented two approximation algorithms, one for the weighted case whose ratio is 0.5833, and another for the unweighted case whose ratio is $\frac{3}{4}$.

Another related problem is k-Set Packing, where one is given a collection of weighted sets, each containing at most k elements, and the goal is to find a maximum weight subcollection of disjoint sets. Chandra and Halldórsson [12] presented a $\frac{3}{2(k+1)}$-approximation algorithm for this problem. Maximum Carpool Matching can be seen as a special case of k-Set Packing with $k = c_{\max} + 1$. Consider a subset of vertice U of size at most k. Observe that each subset of vertices has an optimal internal assignment of passenger to drivers. Let the weight of this assignment be the profit of U, denoted by $p(U)$. If $k = O(1)$, $p(U)$ can be computed for every U of size at most k in polynomial time. The outcome is a k-Set Packing instance. This leads to a $\frac{3}{2(c_{\max}+2)}$-approximation algorithm when $c_{\max} = O(1)$.

Our contribution. Section 2 contains approximation algorithms for Maximum Carpool Matching. First, in Section 2.1 we show that Maximum Carpool Matching can be formulated as an unconstrained submodular maximization problem, thus it has a $\frac{1}{2}$-approximation algorithm due to [10, 9]. We present a local search algorithm for Maximum Carpool Matching which repeatedly checks whether the current carpool matching can be improved by means of a star centered at a vertex, and it terminates when such a step is not possible. The approximation ratio of this algorithm is $\frac{1}{2}$ if weights are polynomially bounded, and its ratio is $\frac{1}{2}-\varepsilon$ in general.

In Section 3 we consider Maximum Carpool Matching with bounded maximum capacity. In Section 3.1 we show that Maximum Carpool Matching is APX-hard even for undirected and unweighted instances with $\Delta \leq b$, for any $b \geq 3$. In Section 3.2 we provide another local search algorithm, whose approximation ratio is $\frac{1}{2}+\frac{1}{2c_{\max}}-\varepsilon$, for any

$\varepsilon > 0$, for unweighted Maximum Carpool Matching, where $c_{\max} \triangleq \max_{v \in V} c(v)$. Given a parameter k, our algorithm starts with the empty carpool matching. Then, it repeatedly tries to find a better matching by replacing $t \leq k$ arcs in the current solution by $t+1$ arcs that are not in the solution. We show that our analysis is tight. We also note that our algorithm falls within the local search family defined in [7]. However, on undirected and uncapaciated instances we have that $c_{\max} = \Delta$, and as mentioned above the result from [7] does not hold in bounded degree graphs.

Finally, Section 4 discusses Maximum Group Carpool Matching. We show that the unconstrained submodular maximization formulation for Maximum Carpool Matching does not work for Maximum Group Carpool Matching. We show, however, that this problem still admits a $(\frac{1}{2} - \varepsilon)$-approximation algorithm by extending our first local search algorithm. In addition, we show that the second local search algorithm generalizes to unweighted Maximum Group Carpool Matching with the same approximation ratio.

2 Approximation Algorithms

We present two algorithms for Maximum Carpool Matching: a $\frac{1}{2}$-approximation algorithm that is based on formulating the problem as an unconstrained submodular maximization problem and a local search $(\frac{1}{2} - \varepsilon)$-approximation algorithm. While the latter does not improve upon the former, it will be shown (in Section 4) that it can be generalized to Maximum Group Carpool Matching without decreasing the approximation ratio.

2.1 Submodular Maximization

In this section we show that the Maximum Carpool Matching problem can be formulated as an unconstrained submodular maximization problem, and thus it has a $\frac{1}{2}$-approximation algorithm due to Buchbinder et al. [10, 9].

Given a Maximum Carpool Matching instance $(G = (V, A), c, w)$, consider a subset $S \subseteq V$. Let $M(S)$ be a maximum weight carpool matching satisfying $D_{M(S)} \subseteq S \subseteq V \setminus P_{M(S)}$, namely $M(S)$ is the best carpool matching whose drivers belong to S and whose passengers belong to $V \setminus S$. In other words, $M(S)$ is the maximum weight carpool matching that is a subset of $A \cap (V \setminus S) \times S$. Given S, the carpool matching $M(S)$ can be computed in polynomial time by computing a maximum b-matching in the bipartite graph $B = (V \setminus S, S, A \cap (V \setminus S) \times S)$ which can be done using an algorithm for Minimum Cost Flow as shown in [18].

Consider the function $\bar{w} : 2^V \to \mathbb{R}$, where $\bar{w}(S) \triangleq w(M(S)) = \sum_{e \in M(S)} w(e)$. Observe that $\bar{w}(\emptyset) = \bar{w}(V) = 0$, and that $\bar{w}$ is not monotone. In the next lemma we prove that $\bar{w}$ is a *submodular set function*. Recall that a function f is submodular if $f(S) + f(T) \geq f(S \cup T) + f(S \cap T)$ for every two sets S and T in the domain of f.

▶ **Lemma 1.** *$\bar{w}$ is submodular.*

Proof. Consider any two subsets $S, T \subseteq V$. We show that $\bar{w}(S) + \bar{w}(T) \geq \bar{w}(S \cup T) + \bar{w}(S \cap T)$. Let $M(S \cup T)$ and $M(S \cap T)$ be optimal carpool matchings with respect to $S \cup T$ and $S \cap T$. To prove the lemma we construct two feasible carpool matchings M_S and M_T such that $M_S \subseteq (V \setminus S) \times S$, $M_T \subseteq (V \setminus T) \times T$, and $M_S \cup M_T = M(S \cup T) \cup M(S \cap T)$. The lemma follows, since $\bar{w}(S) \geq w(M_S)$ and $\bar{w}(T) \geq w(M_T)$.

First, add all the edges in $M(S \cup T)$ entering $S \setminus T$ to M_S. Similarly, add all the edges in $M(S \cup T)$ entering $T \setminus S$ to M_T. Observe that $d^{\text{in}}_{M_S}(v) = d^{\text{in}}_{M(S \cup T)}(v) \leq c(v)$, for every $v \in S \setminus T$ and that $d^{\text{in}}_{M_T}(v) = d^{\text{in}}_{M(S \cup T)}(v) \leq c(v)$, for every $v \in T \setminus S$. Next, add the edges in $M(S \cap T)$ leaving $T \setminus S$ to S and add the edges in $M(S \cap T)$ leaving $S \setminus T$ to T. It

remains to distribute the edges leaving $V \setminus (S \cup T)$ and entering $S \cap T$ in both $M(S \cup T)$ and $M(S \cap T)$. Note that there may exist edges (v, u), where $v \notin S \cup T$, and $u \in S \cap T$ such that $(v, u) \in M(S \cup T)$ and $M(S \cap T)$. We refer to this edges as *duplicate* edges. We add all edges leaving $V \setminus (S \cup T)$ and entering $S \cap T$ in $M(S \cap T)$ to M_S. Notice that this is possible, since after this addition we have that $d^{\text{in}}_{M_S}(v) \leq d^{\text{in}}_{M(S \cap T)}(v) \leq c(v)$, for every vertex $v \in S \cap T$. Then we add all duplicate edges in $M(S \cup T)$ to M_T. The remaining edges are distributed between M_S and M_T without violating capacities. This can be done, since $d^{\text{in}}_{M(S \cup T)}(v) + d^{\text{in}}_{M(S \cap T)}(v) \leq 2c(v)$, for every $v \in S \cap T$. ◀

Buchbinder et al. [10, 9] presented a general $\frac{1}{2}$-approximation algorithm for unconstrained submodular maximization, thus we have the following theorem.

▶ **Theorem 2.** *There exists a polynomial time $\frac{1}{2}$-approximation algorithm for* Maximum Carpool Matching.

2.2 A Star Improvement Algorithm

In this section we give a local search $(\frac{1}{2} - \varepsilon)$-approximation algorithm for Maximum Carpool Matching. This algorithm repeatedly checks whether the current carpool matching M can be improved by means of a star centered at a vertex v. The profit from this star is the total weight of the arcs in the star, and the cost is the total weight of lost arcs (e.g., arcs from passengers to drivers that became passengers of v). If the profit is larger than the cost, then an improvement step is performed. The algorithm terminates when such a step is not possible. We remind the reader that this algorithm will be extended to Maximum Group Carpool Matching in Section 4.

We need a few definitions before presenting our algorithm. Given a directed graphs $G = (V, A)$, define $N^{\text{in}} \triangleq \{u : (u, v) \in A\}$ and $N^{\text{out}} \triangleq \{u : (v, u) \in A\}$. Let M be a feasible carpool matching. The weight $w_M(v)$ of a vertex v with respect to M is the sum of the weights of the arcs in M that are incident on v, namely

$$w_M(v) \triangleq w(M \cap N^{\text{in}}) + w(M \cap N^{\text{out}}) = \textstyle\sum_{(u,v) \in M} w(u, v) + \sum_{(v,u) \in M} w(v, u) \ .$$

For a subset of vertices $U \subseteq V$ we define $w_M(U) \triangleq \sum_{v \in U} w_M(v)$.

We now argue that, with respect to any carpool matching M, the total weight of all the vertices is equal to twice the weight of the matching.

▶ **Observation 3.** $w_M(V) = 2w(M)$.

Proof. $\displaystyle\sum_{v \in V} w_M(v) = \sum_{v \in V} \sum_{(u,v) \in M} w(u, v) + \sum_{v \in V} \sum_{(v,u) \in M} w(v, u) = 2 \sum_{e \in M} w(e).$ ◀

Denote by $\delta(u, v)$ the difference between the weight of the arc and the weight of its source vertex, that is: $\delta_M(u, v) \triangleq w(u, v) - w_M(u)$. For a subset $S \subseteq A$ of arcs define $\delta(S) \triangleq \sum_{(u,v) \in S} \delta(u, v)$.

A subset S_v of arcs entering a vertex v, whose size is not greater than the capacity of v, is called an *improvement* to vertex v if $\delta(S_v)$ is greater than the value of v. More formally,

▶ **Definition 4.** A subset $S_v \subseteq A \cap (V \times \{v\})$ is an *improvement* with respect to a carpool matching M, if $|S_v| \leq c(v)$ and $\delta_M(S_v) > w_M(v)$. Furthermore, if there exists an improvement for a vertex v, we say that vertex v can be *improved*.

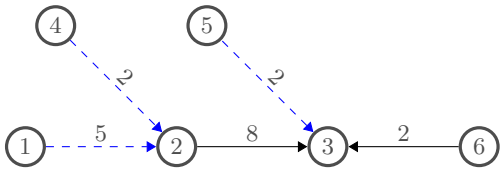


Figure 2 In this example M is the set of the blue, dashed arcs. In this case $w_M(2) = 7$, $w_M(5) = 2$, and $w_M(6) = 0$. Also, $\delta_M(2,3) = 1$ and $\delta_M(6,3) = 2$. The set $\{(2,3),(6,3)\}$ is an *improvement* to vertex 3 and $\Gamma(2,3) = \{(1,2),(4,2),(3,5),(6,3)\}$.

Algorithm 1: StarImprove(G, c)

```
1 M ← ∅
2 repeat
3 |   done ← TRUE
4 |   for v ∈ V do
5 |   |   if there exists an improvement S_v then
6 |   |   |   M ← M \ Γ(S_v) ∪ S_v
7 |   |   |   done ← FALSE
8 until done;
```

Given an arc $(u,v) \in A$, let $\Gamma(u,v)$ be the set of arcs that incident (u,v), namely define $\Gamma(u,v) \triangleq (N^{\text{in}}(u) \times \{u\}) \cup (\{v\} \times N^{\text{out}}(v))$. If S is a set of arcs, then $\Gamma(S) \triangleq \bigcup_{(u,v)\in S} \Gamma(u,v)$. Figure 2 depicts all the above definitions.

We are now ready to describe our local search algorithm, which is called **StarImprove** (Algorithm 1). It starts with an empty carpool matching M, and in every iteration it looks for a vertex that can be improved. If there exists such a vertex v, then the algorithm removes the arcs that are incident on it from M, and adds the arcs in S_v. The algorithm terminates when no vertex can be improved. Figure 3 depicts an improvement step.

We proceed to bound the approximation ratio of the algorithm, assuming termination.

For a vertex v and a set S of edges entering v, let $N_S^{\text{in}}(v) = \{u : (u,v) \in S\}$ be the set in-neighbors corresponding to S.

▶ **Lemma 5.** *Let M be a matching computed by **StarImprove**. Let v be a vertex with no improvement, and let $S \subseteq N_M^{in}(v)$, such that $|S| \leq c(v)$, then $w(S) \leq w_M(v) + w_M(N_S^{in}(v))$.*

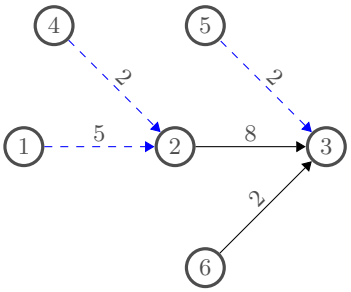
Proof. If no improvement exists, then we have that
$w(S) - w_M(N_S^{\text{in}}(v)) = \sum_{(u,v)\in S}(w(u,v) - w_M(u)) = \delta_M(S) \leq w_M(v)$. ◀

To bound the approximation ratio of the algorithm, we use a charging scheme argument.

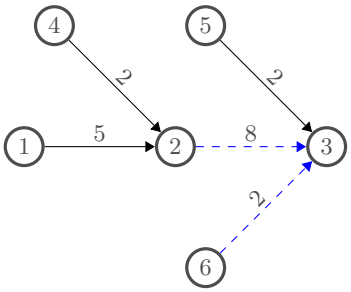
▶ **Lemma 6.** *If **StarImprove** terminates, then the computed solution is $\frac{1}{2}$-approximate.*

Proof. Let M be the matching produced by the algorithm, and let M^* be an optimal matching. We load every vertex v with an amount of money equal to $w_M(v)$, and then we show that this is enough to pay for every arc in the optimal matching. Due to Observation 3 the total amount of money that we use is exactly twice the weight of M.

Consider a driver $v \in D_{M^*}$, and let $S = (V \times \{v\}) \cap M^*$. By lemma 5 we know that $w(S) \leq w_M(v) + w_M(N_S^{\text{in}}(v))$, thus we can pay for S, using the money on v and on $N_S^-(v)$. Clearly, these vertices will not be charged again. ◀



(a) A matching that can be improved.



(b) The matching after improving vertex 3.

Figure 3 An improvement example.

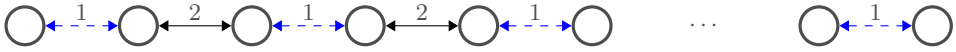


Figure 4 Consider a path with $2n+1$ arcs, and alternating arc weights (2 and 1), if **StarImprove** selects all arcs of weight 1, then no further improvement can be done and the value of the matching is $n+1$, while the optimal matching has value of $2n$.

We show that our analysis is tight using in Figure 4.

It remains to consider the running time of the algorithm.

▶ **Theorem 7.** *Algorithm* ***StarImprove*** *is a* $\frac{1}{2}$*-approximation algorithm for* Maximum Carpool Matching*, if edge weights are integral and polynomially bounded.*

Proof. First, observe that determining if a vertex v can be improved can be done efficiently by considering the incoming arcs to v in a non-increasing order of their δ_Ms, and only ones with positive values. A vertex v can be improved, then, if the δs of the first $c(v)$ (or less) arcs sum up to more than $w_M(v)$. It follows that the running time of an iteration of the for-loop is polynomial. Since the edge weights are integral and polynomially bounded, the weight of an optimal carpool matching is polynomially bounded. The algorithm runs in polynomial time, because in each iteration the algorithm improves the weight of the matching by at least one or otherwise it terminates. ◀

It remains to consider the case of general weights. It can be shown that one can use standard scaling and rounding to ensure a polynomial running time in the cost of a $(1+\varepsilon)$ factor in the approximation ratio. The proof is omitted for lack of space.

▶ **Theorem 8.** *There exists a* $(\frac{1}{2}-\varepsilon)$*-approximation algorithm for* Maximum Carpool Matching*, for every* $\varepsilon \in (0, \frac{1}{2})$*.*

3 Constant Maximum Capacity

In this section we study the Maximum Carpool Matching problem when the maximum capacity is constant, i.e., when $c_{\max} = O(1)$. We show that this variant of the problem is APX-hard even for unweighted and undirected instances. We also describe and analyze a local search algorithm for the unweighted variant of the problem, and show that the algorithm achieves a $\frac{1}{2} + \frac{1}{2c_{\max}} - \varepsilon$ approximation ratio, for any $\varepsilon > 0$.

3.1 Hardness

As we mentioned earlier, Spanning Star Forest has a lower bound of $\frac{10}{11}+\varepsilon$ for any $\varepsilon > 0$, unless P=NP [11], and this bound applies to Maximum Carpool Matching. The result, however, does not hold for the case where $\Delta = O(1)$ (and $c_{\max} = O(1)$). In this section we show that the problem remains APX-hard even in this case.

Formally, the (unweighted) Minimum Dominating Set problem is defined as follows. The input is an undirected graph $G = (V, E)$, and a feasible solution, or a *dominating set*, is a subset $D \subseteq V$ that dominates V namely such that $D \cup \bigcup_{v \in D} N(v) = V$, where $N(v)$ is the neighborhood of v. The goal is to find a minimum cardinality dominating set. Minimum Dominating Set-b is the special case of Minimum Dominating Set in which the maximum degree of a vertex in the input graph G is bounded by b. The problem was shown to be APX-hard, for $b \geq 3$, by Papadimitriou and Yannakakis [20].

We now consider the unweighted and undirected special case of the Maximum Carpool Matching problem. In this case, the input consists of an undirected graph G and a capacity function c, and the goal is to find a carpool matching M that maximizes $|P_M|$.

Given an undirected graph G, let D^* be a minimum cardinality dominating set, and let M^* be an optimal carpool matching with respect to G and the capacity function: $c(v) = \deg(v)$, for every $v \in V$.

▶ **Observation 9.** $|P_{M^*}| + |D^*| = |V|$

Proof. Given a carpool matching M, observe that $D_M \cup Z_M$ is a dominating set. In the other direction, a dominating set D induces a carpool matching of size $|V \setminus D|$. ◀

We use this duality to obtain a hardness result for Maximum Carpool Matching.

▶ **Theorem 10.** *The* Maximum Carpool Matching *problem is APX-hard, even for undirected and unweighted instances with maximum degree bounded by b, for $b \geq 3$.*

Proof. We prove the theorem by presenting an L-reduction from Minimum Dominating Set-b. (For details on L-reductions the reader is referred to [20].) We define a function f from Minimum Dominating Set-b instances to Maximum Carpool Matching instances as follows: $f(G) = (G, c)$, where $c(v) = \deg(v)$, for every $v \in V$. Next, we define a function g that given a carpool matching computes a dominating set as follows: $g(M) = V \setminus P_M$. Both f and g can be computed in polynomial time.

Let D^* be an optimal dominating set with respect to G, and let M^* be an optimal carpool matching with respect to G and c. Since $|D^*| \geq \frac{|V|}{b+1}$, it follows that $|P_{M^*}| \leq b|D^*|$, In addition, if M is a carpool matching, we have that $|D_M \cup Z_M| - |D^*| = (|V| - |P_M|) - |D^*| = |P_{M^*}| - |P_M|$. Hence, there is an L-reduction from Minimum Dominating Set-b to unweighted and undirected Maximum Carpool Matching with bounded capacity b. ◀

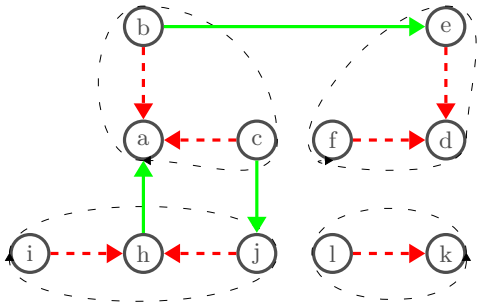
3.2 Local Search

In this section we present a local search $(\frac{1}{2} + \frac{1}{2c_{\max}} - \varepsilon)$-approximation algoithm for unweighted Maximum Carpool Matching whose running time is polynomial if $c_{\max} = O(1)$.

Let k be a constant integer to be determined later. Algorithm **EdgeSwap** (Algorithm 2) maintains a feasible matching M throughout its execution and operates in iterative manner where in each iteration it tries to find a better solution by replacing a subset of at most k edges in the current solution with another (larger) subset of edges not in the solution. The algorithm halts when no improvement can be done.

Algorithm 2: **EdgeSwap**(G, c, k)

```
1  M ← ∅
2  repeat
3      done ← true
4      forall M' ⊆ M : |M'| ≤ k do
5          forall A' ⊆ A \ M : |A'| = |M'| + 1 do
6              if M \ M' ∪ A' is feasible then
7                  M ← M \ M' ∪ A'
8                  done ← false
9  until done;
10 return M
```



(a) M^* is depicted by the dashed red edges, and M is depicted by the solid green edges. The optimal stars are outlined.

(b) The star graph: each vertex corresponds to a star in G. The upper left vertex corresponds to the star that contains the vertices a,b,c.

Figure 5 An example of a star graph.

Algorithm **EdgeSwap** terminates in polynomial time, since in every non-final iteration it improves the value of the solution by one. Thus, after at most n iterations the algorithm terminates. In every iteration the algorithm examines all subsets of edges of a fixed size and tests for feasibility, both these operations can be done in polynomial time.

Observe that a vertex $v \in D_M \cup Z_M$ is the center of a *directed star* whose leaves are the passengers in the set $P_M(v) = \{u : (u, v) \in M\}$ ($P_M(v) = \emptyset$, for $v \in Z_M$). Given a carpool matching M, we define $\mathcal{S}(M)$ to be the set of stars that are induced by M. Denote by $V(S)$ the set of vertices of a star, i.e., if v is the center of S, then $V(S) = \{v\} \cup P_M(v)$. Also, let $A(S)$ be the arcs of S. For $\mathcal{T} \subseteq \mathcal{S}(M)$, define $V(\mathcal{T}) \triangleq \bigcup_{S \in \mathcal{T}} V(S)$ and $A(\mathcal{T}) \triangleq \bigcup_{S \in \mathcal{T}} A(S)$.

It remains to analyze the approximation ratio of **EdgeSwap**. Let M^* be an optimal matching, and let M be the matching computed by **EdgeSwap**. Given both matchings we build the *star graph* in which each vertex represents a star from the optimal solution, namely from $\mathcal{S}(M^*)$, and an edge exists between two vertices if there is a star in $\mathcal{S}(M)$ that intersects the two corresponding stars of the optimal solution. Formally $H = (\mathcal{S}(M^*), E)$ where $E = \left\{(S_i^*, S_j^*) : \exists S \in \mathcal{S}(M), V(S) \cap V(S_i^*) \neq \emptyset \wedge V(S) \cap V(S_j^*) \neq \emptyset\right\}$. Figure 5 depicts a star graph.

▶ **Lemma 11.** *The maximum degree of H is $c_{\max}(c_{\max} + 1)$.*

Proof. Each star in $\mathcal{S}(M^*)$ contains at most $c_{\max} + 1$ vertices and each such vertex can belong to a star in $\mathcal{S}(M)$ containing additional $c_{\max}$ vertices, each of which is located in a different star in $\mathcal{S}(M^*)$. ◀

In what follows we compare $|M|$ and $|M^*|$ in maximal connected components of the star graph H. Intuitively, we show that M is optimal on small maximal components, and that the approximation ratio on medium (non-necessarily) components can be bounded due to the termination condition of **EdgeSwap**. Large maximal components will be partitioned into medium components.

We first show that large connected graphs (or maximal connected components) can be partitioned into medium size components. The proof is omitted for lack of space.

▶ **Lemma 12.** *An undirected connected graph $G = (V, E)$ with maximum degree Δ, can be decomposed into connected components of size at least ℓ and at most $\Delta\ell$, if $\ell \leq |V|$.*

Define $\deg_M(v) \triangleq d_M^{\text{in}}(v) + d_M^{\text{out}}(v)$. For a subset $U \subseteq V$ of vertices define $\deg_M(U) \triangleq \sum_{v \in U} \deg_M(u)$. Observe that $|M| = \frac{1}{2}\deg_M(V)$.

In the next lemma we bound the degree ratio in a component that contains stars with at most k arcs.

▶ **Lemma 13.** *Let $\mathcal{T} \subseteq \mathcal{S}(M^*)$ that induces a connected subgraph of H. If $|A(\mathcal{T})| \leq k$, then*

$$\frac{\deg_M(V(\mathcal{T}))}{\deg_{M^*}(V(\mathcal{T}))} \geq \frac{1}{2} + \frac{1}{2c_{\max}} - \frac{1}{2c_{\max}|\mathcal{T}|} .$$

Proof. Consider the solution M' obtained from M by removing all the edges from M that intersect $V(\mathcal{T})$ and adding all the edges from M^* that intersect $V(\mathcal{T})$. Observe that if an edge (u, v) in M^* intersects $V(\mathcal{T})$, then $\{u, v\} \in V(\mathcal{T})$ by the definition of the graph H. Hence, M' is feasible carpool matching.

Since $\mathcal{T}$ induces a connected subgraph of H, the removal of edges in M that intersect $V(\mathcal{T})$ decreased $|M|$ by at most $\deg_M(V(\mathcal{T})) - |\mathcal{T}| + 1$. On the other hand, the increase in size is exactly $\frac{1}{2}\deg_{M^*}(V(\mathcal{T})) \leq c_{\max}|\mathcal{T}|$. Since $|A(\mathcal{T})| \leq k$, we know that this difference can not be positive, or else, EdgeSwap would not have terminated. Thus $\frac{1}{2}\deg_{M^*}(V(\mathcal{T})) \leq \deg_M(V(\mathcal{T})) - |\mathcal{T}| + 1$, and so

$$\frac{\deg_M(V(\mathcal{T}))}{\deg_{M^*}(V(\mathcal{T}))} \geq \frac{1}{2} + \frac{|\mathcal{T}| - 1}{\deg_{M^*}(V(\mathcal{T}))} \geq \frac{1}{2} + \frac{|\mathcal{T}| - 1}{2c_{\max}|\mathcal{T}|} = \frac{1}{2} + \frac{1}{2c_{\max}} - \frac{1}{2c_{\max}|\mathcal{T}|} ,$$

as required. ◀

It remains to bound the approximation ratio of **EdgeSwap**.

▶ **Lemma 14.** *If $k \geq c_{\max}$, then $|M| \geq (\frac{1}{2} + \frac{1}{2c_{\max}} - \frac{c_{\max}(c_{\max}+1)}{2k}) \cdot |M^*|$.*

Proof. Consider a maximal (with respect to set inclusion) connected component of H induced by the vertices in $\mathcal{T} \subseteq \mathcal{S}(M^*)$. If $|M \cap A(\mathcal{T})| \leq k$, then it must be that $|M \cap A(\mathcal{T})| = |M^* \cap A(\mathcal{T})|$, since otherwise $M \cap A(\mathcal{T})$ could be improved.

It remains to consider a maximal component $\mathcal{T}$ such that If $|M \cap A(\mathcal{T})| > k$. Since the number of edges in $S \in \mathcal{S}(M^*)$ is at most $c_{\max}$, it must be that $|V(\mathcal{T})| > \frac{k}{c_{\max}}$. Due to Lemma 12 (with $\ell = \frac{k}{c_{\max}^2(c_{\max}+1)}$) we can partition $\mathcal{T}$ into connected components each of which contains between $\frac{k}{c_{\max}^2(c_{\max}+1)}$ and $\frac{k}{c_{\max}}$ vertices. Since each such vertex set $\mathcal{X}$ is connected and contains at most $\frac{k}{c_{\max}}$ stars, it follows that $|A(\mathcal{X})| \leq k$. Due to Lemma 13 we have that $\frac{\deg_M(V(\mathcal{X}))}{\deg_{M^*}(V(\mathcal{X}))} \geq \frac{1}{2} + \frac{1}{2c_{\max}} - \frac{c_{\max}(c_{\max}+1)}{2k}$. Since $\deg_{M^*}(V(\mathcal{T})) = \sum_{\mathcal{X}} \deg_{M^*}(V(\mathcal{X}))$ and $\deg_M(V(\mathcal{T})) = \sum_{\mathcal{X}} \deg_M(V(\mathcal{X}))$, it follows that $\frac{\deg_M(V(\mathcal{T}))}{\deg_{M^*}(V(\mathcal{T}))} \geq \frac{1}{2} + \frac{1}{2c_{\max}} - \frac{c_{\max}(c_{\max}+1)}{2k}$, and thus $|M \cap A(\mathcal{T})| \geq \left(\frac{1}{2} + \frac{1}{2c_{\max}} - \frac{c_{\max}(c_{\max}+1)}{2k}\right)|M^* \cap A(\mathcal{T})|$. ◀

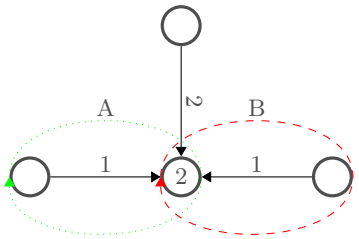


Figure 6 An unweighted Maximum Carpool Matching instance. Capacities are written inside vertices, and arcs are labeled with their size. We have that $\bar{w}(A) + \bar{w}(B) = 2 < 3 = \bar{w}(A \cup B) + \bar{w}(A \cap B)$.

By setting $k = \lceil c_{\max}(c_{\max}+1)/2\varepsilon \rceil$, we get the following result.

▶ **Corollary 15.** *There exists a* $(\frac{1}{2} + \frac{1}{2c_{\max}} - \varepsilon)$*-approximation algorithm for unweighted* Maximum Carpool Matching*, for every* $\varepsilon > 0$.

4 Group Carpool

We now consider Maximum Group Carpool Matching which a variant of Maximum Carpool Matching in which we are given a size function $s : V \to \mathbb{N}$, and the constraint $d_M^{\text{in}}(v) \leq c(v)$ is replaced with the constraint $\sum_{u:(u,v)\in M} s(u) \leq c(v)$.

We start by showing that this variant of the problem does not fit the submodular maximization formulation as defined for Maximum Carpool Matching. Recall the submodular maximization formulation given in Section 2.1, namely $\bar{w} : 2^V \to \mathbb{R}$, where $\bar{w}(S) \triangleq w(M(S))$ and $M(S)$ is the maximum weight carpool matching that satisfies $D_{M(S)} \subseteq S \subseteq V \setminus P_{M(S)}$. Figure 6 contains an instance that shows that the function $\bar{w}$ is not submodular anymore.

We show that Maximum Group Carpool Matching has a $(\frac{1}{2} - \varepsilon)$-approximation algorithm by extending the algorithm from Section 2.2. The main concern when trying to adopt the algorithm to Maximum Group Carpool Matching is how to determine if a vertex can be improved. With Maximum Carpool Matching, if weights are polynomially-bounded, it was enough to consider the incoming arcs to a vertex v in a non-increasing order of δ_M (see proof of Theorem 7). This does not work anymore, since in the Maximum Group Carpool Matching we have sizes. In fact, given v, finding the best star with respect to δ_M is a Knapsack instance where the size of the knapsack is $c(v)$. If weights are polynomially-bounded, then $\delta_M(e)$ is bounded for every arc $e \in A$, and therefore this instance of Knapsack can be solved in polynomial time using dynamic programming.

▶ **Theorem 16.** *Algorithm* ***StarImprove*** *is a* $\frac{1}{2}$*-approximation algorithm for* Maximum Group Carpool Matching*, if edge weights are integral and polynomially bounded.*

Using standard scaling and rounding we obtain the following result.

▶ **Theorem 17.** *There exists a* $(\frac{1}{2} - \varepsilon)$*-approximation algorithm for* Maximum Group Carpool Matching*, for every* $\varepsilon \in (0, \frac{1}{2})$.

Finally, we show that a variant of Algorithm **EdgeSwap** from Section 3.2 can be used to solve Maximum Group Carpool Matching while keeping the same approximation guarantees. The only difference is that when checking feasibility of a set of arcs we do not compare the number of passengers to the capacity of a driver, but rather compare the total size of the passengers to the capacity.

▶ **Theorem 18.** *There exists a* $(\frac{1}{2} + \frac{1}{2c_{\max}} - \varepsilon)$*-approximation algorithm for unweighted* Maximum Group Carpool Matching*, for every* $\varepsilon > 0$.

Acknowledgements. We thank David Adjiashvili and Reuven Bar-Yehuda for helpful discussions.

References

1. Blablacar. `https://www.blablacar.com`.
2. Moovit carpool. `https://moovitapp.com/`.
3. Waze. `https://www.waze.com/`.
4. Zimride by enterprise. `https://zimride.com/`.
5. Niels A. H. Agatz, Alan L. Erera, Martin W. P. Savelsbergh, and Xing Wang. Optimization for dynamic ride-sharing: A review. *European Journal of Operational Research*, 223(2):295–303, 2012.
6. Esther M. Arkin, Refael Hassin, Shlomi Rubinstein, and Maxim Sviridenko. Approximations for maximum transportation with permutable supply vector and other capacitated star packing problems. *Algorithmica*, 39(2):175–187, 2004.
7. Stavros Athanassopoulos, Ioannis Caragiannis, Christos Kaklamanis, and Maria Kyropoulou. An improved approximation bound for spanning star forest and color saving. In *34th International Symposium on Mathematical Foundations of Computer Science*, pages 90–101, 2009.
8. Amotz Bar-Noy, David Peleg, George Rabanca, and Ivo Vigan. Improved approximation algorithms for weighted 2-path partitions. In *23rd Annual European Symposium on Algorithms*, volume 9294 of *LNCS*, pages 953–964, 2015.
9. Niv Buchbinder and Moran Feldman. Deterministic algorithms for submodular maximization problems. In *27th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 392–403, 2016.
10. Niv Buchbinder, Moran Feldman, Joseph Naor, and Roy Schwartz. A tight linear time (1/2)-approximation for unconstrained submodular maximization. *SIAM J. Comput.*, 44(5):1384–1402, 2015.
11. Deeparnab Chakrabarty and Gagan Goel. On the approximability of budgeted allocations and improved lower bounds for submodular welfare maximization and GAP. *SIAM J. Comput.*, 39(6):2189–2211, 2010.
12. Barun Chandra and Magnús M. Halldórsson. Greedy local improvement and weighted set packing approximation. *J. Algorithms*, 39(2):223–240, 2001.
13. Ning Chen, Roee Engelberg, C. Thach Nguyen, Prasad Raghavendra, Atri Rudra, and Gyanit Singh. Improved approximation algorithms for the spanning star forest problem. *Algorithmica*, 65(3):498–516, 2013.
14. Irith Ben-Arroyo Hartman. Optimal assignment for carpooling. submitted.
15. Irith Ben-Arroyo Hartman, Daniel Keren, Abed Abu Dbai, Elad Cohen, Luk Knapen, Ansar-Ul-Haque Yasar, and Davy Janssens. Theory and practice in large carpooling problems. In *5th International Conference on Ambient Systems, Networks and Technologies*, pages 339–347, 2014.
16. Luk Knapen, Daniel Keren, Ansar-Ul-Haque Yasar, Sungjin Cho, Tom Bellemans, Davy Janssens, and Geert Wets. Estimating scalability issues while finding an optimal assignment for carpooling. In *4th International Conference on Ambient Systems, Networks and Technologies*, pages 372–379, 2013.
17. Luk Knapen, Ansar-Ul-Haque Yasar, Sungjin Cho, Daniel Keren, Abed Abu Dbai, Tom Bellemans, Davy Janssens, Geert Wets, Assaf Schuster, Izchak Sharfman, and Kanishka

Bhaduri. Exploiting graph-theoretic tools for matching in carpooling applications. *J. Ambient Intelligence and Humanized Computing*, 5(3):393–407, 2014.

18 Gilad Kutiel. Approximation algorithms for the maximum carpool matching problem. In *12th International Computer Science Symposium in Russia*, volume 10304 of *LNCS*, pages 206–216, 2017.

19 C. Thach Nguyen, Jian Shen, Minmei Hou, Li Sheng, Webb Miller, and Louxin Zhang. Approximating the spanning star forest problem and its application to genomic sequence alignment. *SIAM J. Comput.*, 38(3):946–962, 2008.

20 Christos Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. In *12th Annual ACM Symposium on Theory of Computing*, pages 229–234, 1988.

Computing Maximum Agreement Forests without Cluster Partitioning is Folly*

Zhijiang Li[1] and Norbert Zeh[2]

1 **Microsoft Canada, Vancouver, BC, Canada**
zhijiang.li@dal.ca
2 **Faculty of Computer Science, Dalhousie University, Halifax, NS, Canada**
nzeh@cs.dal.ca

Abstract

Computing a maximum (acyclic) agreement forest (M(A)AF) of a pair of phylogenetic trees is known to be fixed-parameter tractable; the two main techniques are kernelization and depth-bounded search. In theory, kernelization-based algorithms for this problem are not competitive, but they perform remarkably well in practice. We shed light on why this is the case. Our results show that, probably unsurprisingly, the kernel is often much smaller in practice than the theoretical worst case, but not small enough to fully explain the good performance of these algorithms. The key to performance is *cluster partitioning*, a technique used in almost all fast M(A)AF algorithms. In theory, cluster partitioning does not help: some instances are highly clusterable, others not at all. However, our experiments show that cluster partitioning leads to substantial performance improvements for kernelization-based M(A)AF algorithms. In contrast, kernelizing the individual clusters before solving them using exponential search yields only very modest performance improvements or even hurts performance; for the vast majority of inputs, kernelization leads to no reduction in the maximal cluster size at all. The choice of the algorithm applied to solve individual clusters also significantly impacts performance, even though our limited experiment to evaluate this produced no clear winner; depth-bounded search, exponential search *interleaved* with kernelization, and an ILP-based algorithm all achieved competitive performance.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory, G.2.3 Applications

Keywords and phrases Fixed-parameter tractability, agreement forests, hybridization, subtree prune-and-regraft

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.56

1 Introduction

Phylogenetic trees are the classical model of evolution. All extant taxa are assumed to descend from the same common ancestor and diverge in a tree-like fashion through speciation events. While this is still the accepted model for the evolution of individual genes, the evolution particularly of microbial organisms and plants is complicated by *reticulation events*, such as *lateral gene transfer* (LGT) and *hybridization*, which are known to play an important role, for example, in the development of antibiotic resistance of bacteria [20]. LGT allows an organism to acquire genetic material from an unrelated species in the same habitat. Hybridization allows an organism to inherit genetic material from more than one ancestor. Different genes shared by a group of taxa then have different tree-like evolutionary histories, which we

* This work was supported in part by NSERC and the Canada Research Chairs programme.

call *gene trees*. The differences between these trees provide the basis for discovering likely reticulation events in the evolution of this set of taxa.

A single LGT has the effect that, in the tree representing the transferred gene, the descendants of the recipient taxon appear genetically most similar to the descendants of the donor taxon, while all other relationships between taxa are preserved. Thus, the "true tree" can be transformed into the gene tree for the transferred gene by cutting off a subtree and grafting it onto the donor edge, a *subtree prune-and-regraft* (SPR) operation [11]. A series of LGTs translates into a sequence of SPR operations that transforms one input tree into the other. A set of hybridizations yields a network that *displays* each gene tree, that is, each tree can be obtained from the network by deleting edges and suppressing degree-2 nodes [2].

Reticulation events are assumed to be rare. Thus, it is common to assume that the smallest set of reticulations consistent with the input trees is the most likely scenario, and we aim to construct a hybridization network of the input trees with as few hybridizations as possible (its *hybridization number*) or a minimum-length sequence of SPR operations transforming one input tree into the other. The length of this sequence is the *SPR distance* between the two trees. Both problems are NP-hard [5, 7] and fixed-parameter tractable [7, 8, 14, 15, 18] when parameterized by the hybridization number or SPR distance. Despite these FPT results, solving either problem for more than two trees is challenging in practice. (It is unclear how to even extend the SPR distance to more than two trees. SPR supertrees [27] offer one possible approach.) For *two* input trees, very fast solutions exist [16, 17, 22, 23, 24, 25, 26]. Almost all of them use kernelization or depth-bounded search and compute the SPR distance or hybridization number via *maximum (acyclic) agreement forests* (M(A)AFs). The best known kernel sizes for MAF and MAAF of binary trees are $28k$ [7] and $14k$ [8], where k is the SPR distance or hybridization number. Combined with an $O(3^n n)$-time M(A)AF algorithm [1], a MAF or MAAF can thus be found in $O(3^{28k}n)$ or $O(3^{14k}n)$ time. For multifurcating trees, the best known kernel sizes are $28k$ and $89k$ [18], and the exact M(A)AF algorithm takes $O(4^n n)$ time. Thus, a MAF or MAAF can be found in $O(4^{28k}n)$ or $O(4^{89k}n)$ time. In contrast, the best depth-bounded search algorithms for M(A)AF take between $O(2^k n)$ and $O(5.08^k n)$ time [16, 17, 23, 24, 25, 26] depending on whether a MAF or MAAF is to be computed and whether the input trees are binary or multifurcating. This leaves an astronomical gap between the theoretical running times of kernelization-based and depth-bounded search algorithms for finding M(A)AFs of all but the simplest inputs. Yet, in practice, kernelization-based algorithms perform remarkably well [1, 9].

In this paper, we try to answer two questions: (1) Why do kernelization-based algorithms perform much better in practice than predicted in theory? (2) Which is the "ultimate" algorithm for computing agreement forests of two trees? Part of the answer to the first question is that the kernel is often much smaller than predicted, less than $4k$. This reduces the difference between the running times of kernelization-based and depth-bounded search algorithms significantly but still leaves a gap of more than 8^k even after porting the improvements from the depth-bounded search algorithms back to exponential search. This gap is massive, since values of $k \geq 50$ are not uncommon. The key to fast running times for almost all existing M(A)AF algorithms is *cluster partitioning* [3, 19], which allows us to break many non-trivial instances into smaller pieces that can be solved independently and whose total SPR distance or hybridization number (roughly) equals the SPR distance or hybridization number of the original input. This has the potential to lead to an exponential speed-up and often does in practice. We verified experimentally that cluster partitioning is the real reason why kernelization-based M(A)AF algorithms are fast: it significantly improves the performance of kernelization-based M(A)AF algorithms (and also of depth-bounded search

algorithms [27]), while kernelization leads to only modest performance gains of algorithms using only cluster partitioning and exponential search and often even hurts performance. A recent theoretical result [6] shows that agreement-based phylogenetic distances are fixed-parameter tractable in the *level* of the optimal hybridization network, which is in fact exactly the maximum hybridization number of the clusters in a cluster partition. This sheds light on the effectiveness of cluster partitioning when it is applicable. Our results suggest that many real-world inputs are highly clusterable, that is, their optimal networks have small level. To answer question (2), we investigated which algorithm, used to solve the subproblems in a cluster partition, results in the fastest running time overall. In our somewhat limited experiments for this question, three winners emerged: depth-bounded search, integer linear programming, and *interleaving* of kernelization and exponential search.

Section 2 formally defines SPR distance, hybridization number, agreement forests, and related concepts. Section 3 gives an overview of the techniques used to compute agreement forests. Section 4 presents our experimental results. Section 5 offers conclusions.

2 Subtree Prune-and-Regraft, Hybridization, and Agreement Forests

A *(rooted phylogenetic) X-tree* is a tree T with a root labelled ρ and with $|X|$ leaves labelled bijectively with the elements in X; ρ has degree 1; all internal nodes have at least two children. Edges are directed away from the root. If all internal nodes have out-degree exactly two, T is *binary*; otherwise it is *multifurcating*. T is a *resolution* of another tree S if S can be obtained from T by contracting edges. Figures 1a,c illustrate these definitions.

A *(rooted phylogenetic) X-network* is a directed acyclic graph (DAG) with a single source ρ and $|X|$ sinks labelled bijectively with the elements in X. The nodes with in-degree at least two are called *hybrid nodes*. An X-network N *displays* an X-tree T if T can be obtained from N by deleting edges and suppressing unlabelled out-degree-1 nodes. Since we always suppress unlabelled out-degree-1 nodes, we do not state this explicitly from here on. N is a *hybridization network* of a pair of X-trees (S,T) if it displays both S and T. The *hybridization number* of N is the number of edges we need to delete to obtain a tree. The hybridization number $hyb(S,T)$ of a pair of X-trees (S,T) is the minimum hybridization number of all hybridization networks of (S,T). Figure 1e illustrates these definitions.

A *subtree prune-and-regraft (SPR) operation* on a binary X-tree T deletes the parent edge of some node v, splits some edge by introducing a new node u, and makes v a child of u. The *SPR distance* $d_{\mathrm{SPR}}(S,T)$ between two binary X-trees S and T is the minimum number of SPR operations needed to transform S into T; see Figure 1d.

A *(rooted binary) X-forest* is a forest F that can be obtained from a (binary) X-tree by deleting edges. An X-forest F_1 *refines* another X-forest F_2 if F_1 can be obtained from F_2 by deleting edges. An X-forest F is an *agreement forest* (AF) of a pair of binary X-forests (F_S,F_T) if it refines both F_S and F_T. A *maximum agreement forest* (MAF) of (F_S,F_T) is an AF of (F_S,F_T) with the minimum number of components; see Figure 1f. For a component C of a forest F that refines an X-tree T, let $\mathrm{LCA}_T(C)$ be the lowest common ancestor in T of all leaves of C. A component C_1 of F is an *ancestor* of another component C_2 of F in T if $\mathrm{LCA}_T(C_1)$ is an ancestor of $\mathrm{LCA}_T(C_2)$. The *ancestry graph* $G_T(F)$ of F w.r.t. T has the components of F as its nodes and contains a directed edge (C_1,C_2) if the ancestors of C_1 are exactly the proper ancestors of C_2. For an AF F of a pair of X-trees (S,T), let $G_{S,T}(F) = G_S(F) \cup G_T(F)$. We call F an *acyclic agreement forest* (AAF) of (S,T) if $G_{S,T}(F)$ is a DAG. A *maximum acyclic agreement forest* (MAAF) of (S,T) is an AAF of (S,T) with the minimum number of components. Figures 1f–i illustrate these definitions.

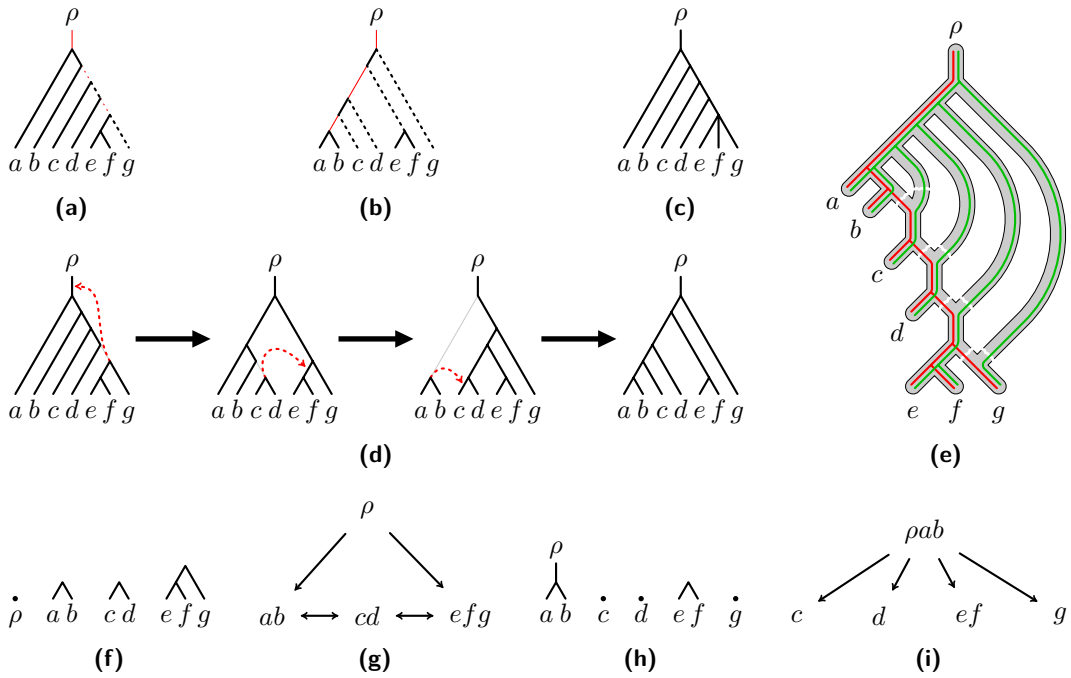


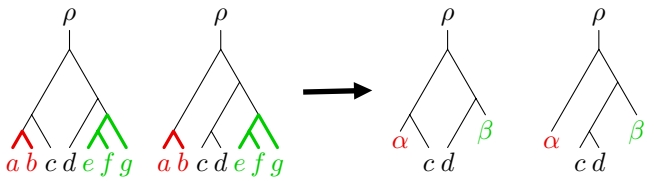
Figure 1 (a,b) Two binary X-trees S and T. **(c)** A multifurcating X-tree that has S as a resolution. **(d)** A sequence of SPR operations that turns S into T. **(e)** A hybridization network of (S,T). **(f)** A MAF of (S,T) that can be obtained by deleting the thin red (dashed or solid) edges in S and T. These are exactly the parent edges of the subtrees of S that are moved by SPR operations in (d). This is not an AAF, since its ancestry graph shown in **(g)** has a cycle. **(h)** A MAAF of (S,T) that can be obtained by deleting the dotted edges in S and T. Its ancestry graph shown in **(i)** is acyclic. We can also obtain this MAAF by deleting the parent edges of all hybrid nodes of the network in (e), which in turn correspond to the dotted edges in S and T.

Let $m(S,T)$ be the size (number of components) of a MAF of (S,T) and let $\tilde{m}(S,T)$ be the size of a MAAF of (S,T). As shown in [2, 7], we have $m(S,T) = 1 + d_{\mathrm{SPR}}(S,T)$ and $\tilde{m}(S,T) = 1 + hyb(S,T)$. In fact, it is easy to convert back and forth between any (A)AF and a corresponding SPR sequence or hybridization network, as illustrated in Figures 1d,e,f,h.

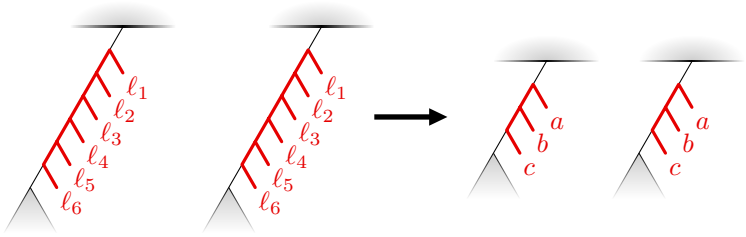
Multifurcating trees usually arise due to lack of confidence in the order of speciation events derived using statistical inference methods. In order to avoid the inference of spurious reticulation events necessary only to reconcile differences in the ordering of these events in different gene trees, low-confidence edges are contracted, resulting in nodes with more than two children. Given this source of multifurcations, it is common to define the SPR distance or hybridization number of two multifurcating trees S and T to be the minimum SPR distance or hybridization number of all pairs of binary resolutions of S and T; a M(A)AF of (S,T) is the smallest M(A)AF over all pairs of binary resolutions of S and T.

3 Techniques for Computing Agreement Forests

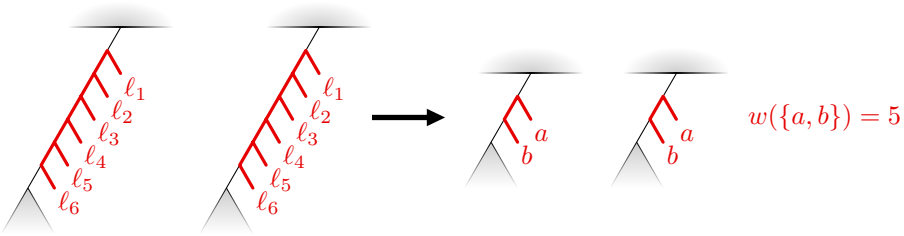
Almost every existing algorithm for computing a M(A)AF of two trees uses a combination of four techniques: *kernelization*, *exponential search*, *depth-bounded search*, and *cluster partitioning*. In this section, we review these techniques. We discuss only binary trees here. The techniques for multifurcating trees are similar albeit more complicated.



(a) Subtree reduction.



(b) MAF chain reduction.



(c) MAAF chain reduction.

Figure 2 Kerneliza tion rules for binary trees.

3.1 Kernelization

A *pendant subtree* of a binary X-tree T is a subtree induced by the descendants of a node in T. An *m-chain* of T is a sequence of leaves $\langle \ell_1, \ell_2, \ldots, \ell_m \rangle$ of T whose parents $p_1, p_2, \ldots, p_m$ form a directed path from p_1 to p_m in T. The kernelization algorithm for M(A)AF of binary trees uses two reduction rules, with separate chain reductions for MAF and MAAF:

Subtree reduction: Let P be a maximal common pendant subtree of S and T. Remove all nodes of P except the root from both S and T. This turns the root of P into a common leaf of S and T. Give both these leaves the same label, distinct from all labels already in X. See Figure 2a. This preserves $m(S,T)$ and $\tilde{m}(S,T)$.

Chain reduction (MAF): Replace every maximal common m-chain of S and T with $m > 3$ with a 3-chain $\langle a, b, c \rangle$ in both trees, where a, b, c are three new leaves currently not in X. See Figure 2b. This preserves $m(S,T)$.

Chain reduction (MAAF): Replace every maximal common m-chain of S and T with $m > 2$ with a 2-chain $\langle a, b \rangle$. This does *not* preserve $\tilde{m}(S,T)$, but $\tilde{m}(S,T)$ (along with a corresponding MAAF) can still be computed from the *weight* $w(F')$ of an appropriate AAF F' of the kernel (S', T'). As a basis for defining $w(F')$ below, add $\{a, b\}$ to a collection W of subsets of X and define $w(\{a, b\}) = m - 1$. See Figure 2c.

Bordewich and Semple [7] proved that subtree reduction and MAF chain reduction preserve $m(S,T)$ and produce a kernel (S', T') of size at most $28m(S,T)$. In the case of MAAF, a *legitimate* AAF F' of the kernel (S', T') is an AAF where, for every pair $\{a, b\} \in W$,

either a and b are singletons (i.e., are each in their own component) or belong to the same component. Let $W_s \subseteq W$ be the subset of pairs $\{a, b\} \in W$ such that a and b are singletons in F' and let $w(F') = |F'| + \sum_{\{a,b\} \in W_s} w(\{a, b\})$. Bordewich and Semple [8] proved that subtree reduction and MAAF chain reduction produce a kernel (S', T') of size at most $14\tilde{m}(S, T)$, every legitimate AAF F' of (S', T') corresponds to an AAF of (S, T) of size $w(F')$, and one of these AAFs of (S, T) is in fact a MAAF of (S, T). Thus, it suffices to find a minimum-weight legitimate AAF of (S', T'), which is easily done by augmenting the exponential search algorithm in Section 3.2 so it ignores non-legitimate AFs.

3.2 Exponential Search

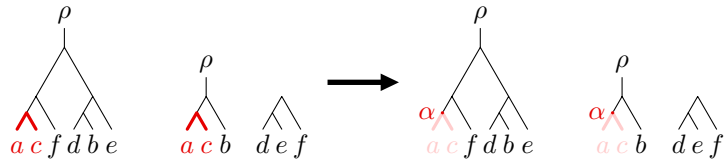
The exponential search algorithm for finding a M(A)AF of (S, T) [1] uses a recursive procedure M(A)AF(F_S, F_T, F), where F_S refines S; F_T refines T, F is an AF of (F_S, F_T), and every component of F is a pendant subtree of both F_S and F_T. MAF(F_S, F_T, F) computes a MAF of (F_S, F_T). MAAF(F_S, F_T, F) computes the smallest AAF of (S, T) that refines F_T and is refined by F; if no such AAF exists, MAAF(F_S, F_T, F) reports failure. Thus, the top-level invocation M(A)AF(S, T, F_X), where F_X has one singleton component per element in X, finds a M(A)AF of (S, T). Since M(A)AF(F_S, F_T, F) treats components of F as indivisible units, we describe the algorithm as if each of these components were replaced by a single leaf in both F_S and F_T. M(A)AF(F_S, F_T, F) first applies the following two rules:

Cherry reduction (MAF only): Let a and c be two sibling leaves of F_S, a *cherry*. If a and c are siblings also in T, then merge a and c, that is, contract them into their common parent in both F_S and F_T, and replace them with a single node in F. See Figure 3a. F remains an AF of (F_S, F_T).

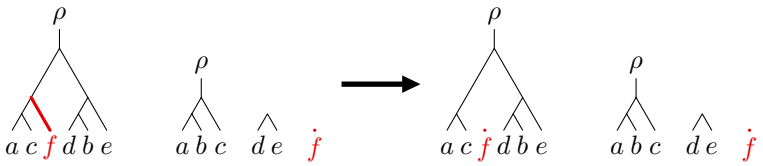
Singleton reduction: If F_T has a singleton leaf that is not a singleton in F_S, then cut its parent edge in F_S. F remains an AF of (F_S, F_T). See Figure 3b.

Let F'_S, F'_T, and F' be the forests obtained once neither rule is applicable. If $F' = F'_T$ (and hence $F' = F'_S$), then F' is a MAF of (F'_S, F'_T) and, after undoing all cherry reductions, of (F_S, F_T), so MAF(F_S, F_T, F) returns F' in this case. Since MAAF(F_S, F_T, F) does not apply cherry reduction, we have $F' = F$ and $F'_T = F_T$ in MAAF(F_S, F_T, F). Thus, if $F' = F'_T$, F is the only forest that refines F_T and is refined by F. MAAF(F_S, F_T, F) checks whether F is an AAF of (S, T) and either returns F or reports failure. If $F' \neq F'_T$, then there exists a cherry (a, c) in F'_S. If (a, c) is not a cherry of F'_T and w.l.o.g. a's depth in F_T is no less than c's, then a has a sibling b in F'_T that is not an ancestor of c and any M(A)AF of (F'_S, F'_T) is a M(A)AF of $(F'_S, F'_T /\!\!/ \{a\})$, $(F'_S, F'_T /\!\!/ \{b\})$ or $(F'_S, F'_T /\!\!/ \{c\})$, where $F /\!\!/ V$ is the forest obtained from F by cutting the parent edges of all nodes in V (see e.g. [1]). See Figure 3c. Thus, M(A)AF(F_S, F_T, F) makes three recursive calls M(A)AF(F'_S, F''_T, F') where $F''_T \in \{F'_T /\!\!/ \{a\}, F'_T /\!\!/ \{b\}, F'_T /\!\!/ \{c\}\}$. If (a, c) *is* a cherry of F'_T, which is possible only for MAAF(F_S, F_T, F) because MAF(F_S, F_T, F) applies cherry reduction, then any AAF F'' of (S, T) that refines F'_T either refines $F_T /\!\!/ \{a\}$ or $F_T /\!\!/ \{c\}$ or (a, c) is a cherry of F''. Thus, MAAF(F_S, F_T, F) makes three recursive calls MAAF(F'_S, F''_T, F''), where either $F'' = F'$ and $F''_T = F'_T /\!\!/ \{a\}$ or $F''_T = F'_T /\!\!/ \{c\}$, or $F''_T = F'_T$ and F'' is obtained from F' by applying cherry reduction to (a, c). See Figure 3d. Since each invocation makes three recursive calls and the recursion depth can be shown to be at most n, the running time is $O(3^n n)$.

Similar ideas give an $O(4^n n)$-time algorithm for multifurcating trees (see [13, 25]). Faster algorithms for both binary and multifurcating M(A)AF are possible, either by using dynamic programming [12] or by porting some of the ideas from the currently fastest depth-bounded search algorithms [23, 25, 26] back to exponential search.



(a) Cherry reduction.



(b) Singleton reduction.

(c) If (a, c) is not a cherry of F_T, M(A)AF(F_S, F_T) makes three recursive calls, cutting the parent edges of a, b, and c in F_T in turn; b may not be a leaf of F_T and may have no corresponding node in F_S.

(d) If (a, c) is a cherry of F_T, MAAF(F_S, F_T) makes three recursive calls, two cutting the parent edges of a and c and one that applies the cherry reduction shown in (a).

Figure 3 Reduction and branching rules in the exponential search algorithm for binary trees.

3.3 Depth-Bounded Search

The depth-bounded search algorithm for MAF is practically identical to the exponential search algorithm. The invocation MAF(F_S, F_T, F, k) now takes an additional parameter k and decides whether there exists an AF of (F_S, F_T) of size at most k. If $|F_T| > k$, MAF(F_S, F_T, k) can immediately report failure, which limits the search depth to k because $|F_T|$ increases by at least one from one level of recursion to the next. Thus, the running time of MAF(S, T, F_X, k) is $O(3^k n)$ for binary trees and $O(4^k n)$ for multifurcating trees. A MAF can be found in $O(3^{m(S,T)} n)$ or $O(4^{m(S,T)} n)$ time, by running MAF(S, T, F_X, k) with parameter $k = 1, 2, \ldots$ until we find the first AF. This approach combined with improved branching rules and other techniques results in the currently fastest MAF algorithms, with running time $O(2^k n)$ for binary trees [23, 26] and $O(2.42^k n)$ for multifurcating trees [25].

The exponential search algorithm for MAAF cannot be translated directly into a depth-bounded search algorithm because, when (a, c) is a common cherry of F'_S and F'_T, the algorithm makes three recursive calls and the branch that applies cherry reduction cuts no edges. To obtain a depth-bounded search algorithm for MAAF, we apply the MAF algorithm (including cherry reduction!) to find a collection of AFs. It turns out that, given a parameter $k \geq \tilde{m}(S, T)$, MAF(S, T, X, k) finds an AF F that can be refined to a MAAF of (S, T) by cutting more edges [24]. Thus, to find an AAF of (S, T) of size at most k, if it exists, we run MAF(S, T, X, k) and, for each AF F it finds, check whether an AAF of (S, T) of size at most k can be obtained by cutting more edges in F. This takes $O(n) \cdot \sum_{i=0}^{k-|F|} \binom{|F|-1}{i}$ time [24]. The currently fastest hybridization algorithms for two trees use this approach and take $O(3.18^k n)$ time for binary trees [24] and $O(5.08^k n)$ time for multifurcating trees [16, 17].

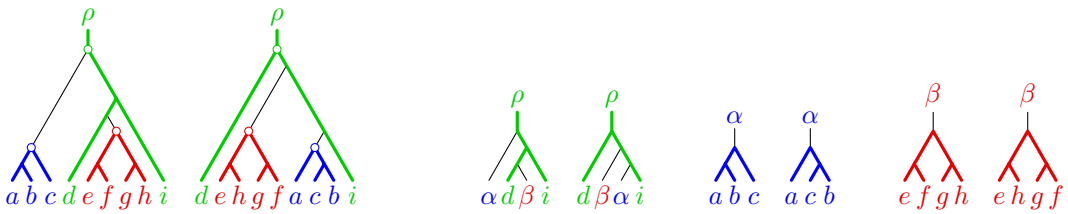


Figure 4 (a) A pair of X-trees (S, T). The highlighted nodes are shared by S and T. **(b)** A cluster partition of (S, T) corresponding to the highlighted subtrees of S and T.

3.4 Cluster Partitioning

Every node of an X-tree defines a *cluster* consisting of the labels of its descendant leaves. An X-tree is fully described by the set of clusters of its nodes, so we can view it as a set of clusters and define $\mathcal{C} = (S \cap T) \setminus \{X \cup \{\rho\}\}$ to be the set of non-root nodes shared by S and T. Each cluster $C \in \mathcal{C}$ defines two subtrees S_C and T_C of S and T consisting of the parents of C in S and T and all nodes that are subsets of C but not proper subsets of any cluster $C' \in \mathcal{C}$ with $C' \subset C$. Let $\mathcal{L} = \{\ell(C) \mid C \in \mathcal{C}\}$ be a label set disjoint from $X \cup \{\rho\}$. We label the roots of S_C and T_C with $\ell(C)$ and each leaf $C' \in \mathcal{C}$ of S_C and T_C with $\ell(C')$. The cluster partition of (S, T) is the collection of instances $\{(S_C, T_C) \mid C \in \mathcal{C}\}$; see Figure 4.

Remarkably, a MAAF of (S, T) can be obtained by computing a MAAF for each pair (S_C, T_C) with $C \in \mathcal{C}$ [3]: A MAAF F_C of each pair (S_C, T_C) can be obtained by cutting a set of edges of S_C. Cutting every edge of S that belongs to the union of these sets produces a MAAF of (S, T). A MAF of (S, T) can similarly be obtained from a collection of AFs of the clusters, but the details are more complicated [19, 27].

4 Experimental Evaluation

4.1 The Competitors

We implemented the techniques discussed in Section 3 in C++, compiled with gcc -O2, and evaluated different combinations of these techniques for finding M(A)AFs of binary and multifurcating trees. Our platform was a 2.4GHz AMD Opteron workstation with 16GB of DDR-1333 RAM running Debian GNU/Linux 7. The algorithms we evaluated were:

K: Apply kernelization and then solve the kernel using exponential search.

CP: Apply cluster partitioning and then solve each cluster using exponential search.

CP+K: Apply cluster partitioning, apply kernelization to each cluster, and then solve each cluster kernel using exponential search.

CP+DBS: Apply cluster partitioning and solve each cluster using depth-bounded search.

We also included two competing algorithms in our evaluation. Since we did not have the (source) code available, we are only able to refer to the experimental results reported by the authors or were able to run the compiled code provided by the authors:

ILP [28]: This solution expresses the problem of finding a MAAF as an integer linear program and then uses CPLEX to solve this instance. The experimental results were obtained on a 3.2GHz Intel Xeon workstation using the Poaceae data set below [28].

INTER [9]: This algorithm uses cluster partitioning and solves each cluster using kernelization and exponential search, applying kernelization in each recursive call. The authors provided a Java implementation as a JAR file.

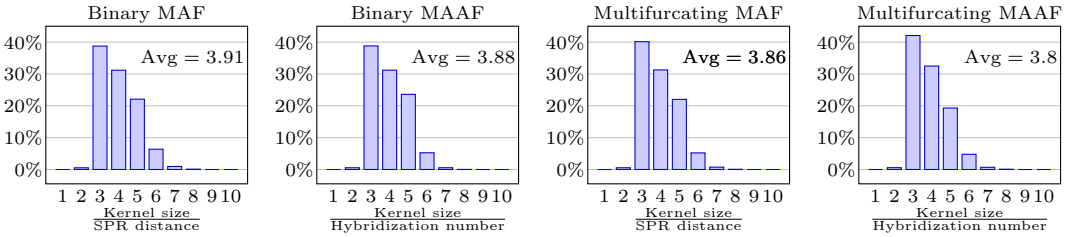


Figure 5 Kernel sizes for the Aquificae data set. The x-axis shows the ratio between kernel size and SPR distance or hybridization number, grouped into buckets where the ith bucket contains all instances with a ratio in the interval $(i-1, i]$. The y-axis shows the percentage of the inputs in each bucket.

We excluded a number of competitors from our evaluation. One is the dynamic programming algorithm of [12]. It achieves a running time of $O(2^n \text{poly}(n))$ but uses exponential space, which is prohibitive. Faster depth-bounded search algorithms with running times of $O(2^k n)$ for binary trees [23, 26] and $O(2.42^k n)$ for multifurcating trees [25] exist and translate into corresponding improvements for exponential search. However, both algorithms are difficult to implement. An implementation of the $O(2^k n)$-time algorithm for binary trees exists [27], while the $O(2.42^k n)$-time algorithm for multifurcating trees has not been implemented yet. In order to avoid performance differences due only to differences in the implementation, we chose to implement all competitors (except ILP and INTER) ourselves and opted for the simpler algorithms discussed in Section 3. Since any improvement applicable to depth-bounded search is applicable to exponential search and vice versa, the qualitative conclusions of our results apply also to faster branching algorithms.

4.2 Data Sets

Aquificae. This data set was provided by Robert Beiko [4] and contained gene trees of the phylum Aquificae, which is generally believed to have a high rate of reticulation events in its history. The input trees were unrooted. A rooting was obtained by Chris Whidden [27], who used a subset of 40,463 of these trees over a set of 1,251 taxa, computing an MRP supertree and rooting the gene trees to match the MRP supertree. Each tree had between 4 and 74 taxa. Comparisons between pairs of trees were made only on the subtrees induced by their common taxa. The original trees were binary. Multifurcating versions were obtained by collapsing bipartitions with support below 0.8. We carried out pairwise comparisons between all pairs of these 40,463 trees. Our experimental evaluation excluded all pairs with SPR distance 0, leaving us with roughly 170,000 non-trivial input pairs.

Poaceae. This data set was provided by Heiko Schmidt [21], who constructed rooted binary trees from the sequence data of six loci (ITS, ndhF, phyB, rbcL, rpoC2, and waxy) provided by the Grass Phylogeny Working Group [10]. The resulting data set contained 15 tree pairs. We used this data set in our final experiment that included ILP and INTER because we did not have the code of ILP available and the authors of [28] reported results on this data set.

4.3 Results

Kernel size. The first question we wanted to answer was: How much smaller than the theoretical prediction are the kernels produced by the kernelization algorithms in practice? Figure 5 shows the kernel sizes observed for the Aquificae data set in our experiments.

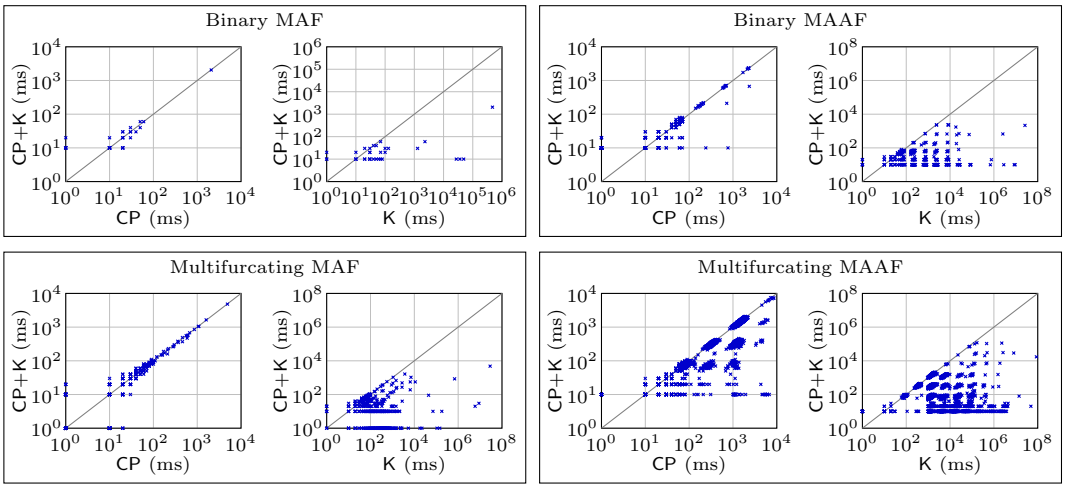


Figure 6 Running times of CP, K, and CP+K on the Aquificae data set. Each instance is represented as a point with the running time of CP or K as the x-coordinate and the running time of CP+K as the y-coordinate. Points below the diagonal indicate that CP+K performed better.

The average kernel size was less than $4k$ for both MAF and MAAF and for binary and multifurcating trees, and the kernel sizes were quite tightly concentrated in the range $[3, 5]$.

Kernelization vs. cluster partitioning. Next we aimed to quantify the relative impact of cluster partitioning and kernelization on the performance of algorithms that combine these techniques. Figure 6 compares the running times of K, CP, and CP+K on a subset of the Aquificae data set. For binary M(A)AF and multifurcating MAF, we removed trivial instances that all three methods were able to solve in less than 1ms. We also removed all instances that K was not able to solve in 8 hours, even though some of these instances could be solved by CP+K in a reasonable time. Multifurcating MAAF is a much harder problem, so even the easy instances took up to 1s to solve and took roughly the same amount of time to solve with any of the three methods. Thus, for multifurcating MAAF, we removed all instances that could be solved in less than 1s or for which K took more than 8 hours. This left 6,800 binary MAF instances, 25,000 binary MAAF instances, 40,000 multifurcating MAF instances, and 5,300 multifurcating MAAF instances. The right-hand figures in the four panels show that adding cluster partitioning to K led to significant performance improvements for almost all instances. For binary M(A)AF and multifurcating MAF, adding kernelization to CP led to only very modest performance improvements. Moreover, there were about as many instances where the overhead of kernelization hurt performance as there were instances where performance improved. The exception is multifurcating MAAF, where adding kernelization to CP led to more significant performance improvements in many instances.

Another useful comparison can be obtained by summing the running times of each algorithm across all test instances, as this amplifies performance gains made on the difficult instances that took a long time to solve. For all but multifurcating MAAF, kernelization did not help and in fact increased the total running time of CP by a factor of almost 5 in the case of binary MAF. For multifurcating MAAF, a modest speed-up by a factor of 2.2 was achieved. In contrast, the performance improvements achieved by adding cluster partitioning to K ranged from 8.5 for binary SPR to 160.8 for multifurcating SPR, again demonstrating the effectiveness of cluster partitioning. The results for binary SPR are to be treated with caution. Almost all binary SPR instances were solved by CP and CP+K

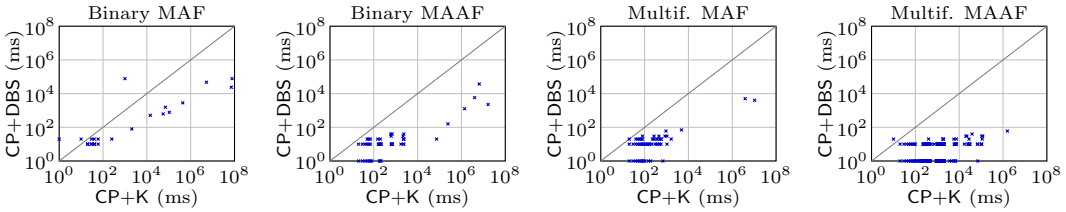


Figure 7 Running times of CP+K and CP+DBS on the Aquificae data set. Each instance is represented as a point with the running time of CP+K as the x-coordinate and the running time of CP+DBS as the y-coordinate. Points below the diagonal indicate that CP+DBS performed better.

in less than 100ms and in less than 10s even using only K. On such "easy" instances, the overhead of any added optimization is fairly large compared to the achievable gain, which we believe explains the detrimental impact of kernelization on the performance of CP and the only modest improvement of performance when adding cluster partitioning to K.

Impact of kernelization on maximal cluster size. Since the running time of an exponential search algorithm across multiple clusters is dominated by the running time on the largest cluster, the lack of impact of kernelization on the performance of CP can be explained by investigating the decrease of the maximal cluster size for each instance due to kernelization. In our experiments, no decrease was achieved in over 99.8% of the inputs for binary M(A)AF and multifurcating MAF, which correlates with our running time comparisons above. For multifurcating MAAF, no reduction was achieved for 93.8% of the inputs while about 6% of the inputs achieved a reduction of the maximal cluster size by 10-30%. While we expected the impact of kernelization on the cluster size to be modest, we were surprised to see that the vast majority of instances did not see *any* decrease in the maximal cluster size.

Kernelization vs depth-bounded search. The next question we aimed to answer was which algorithm to choose to solve individual clusters. Our first experiment with this goal compared CP+K vs CP+DBS. Figure 7 shows that CP+DBS was significantly faster than CP+K. We excluded trivial instances that both methods were able to solve in less than 10ms as well as instances that took CP+K more than 8 hours to solve from the evaluation, even though CP+DBS was able to solve all instances in a reasonable time. A comparison of the total running times of these two methods across all inputs shows that overall CP+DBS was between 135 times (for multifurcating MAAF) and 1,000 times (for binary MAF) faster than CP+K. Since kernelization of the individual clusters was largely ineffective, this is not surprising.

Which is the fastest MAAF algorithm? Since we did not have the source code of ILP and the results in [28] were reported on the Poaceae data set, we chose this data set for a horse race between all the competitors listed in Section 4.1. INTER only computes binary MAAFs, so this was the only type of MAAF we computed in our experiments. As a result, this evaluation is fairly limited. Table 1 shows the results. We report two running times for INTER. The first (A) was obtained on inputs where string labels were replaced with integer labels; the second (B) was obtained using the Poaceae data files bundled with the code of [9], where every leaf was labelled with the name of the taxon. Apart from that, the inputs were identical. We do not know why this change would have such a significant impact on the running time of the implementation. Table 1 shows that ILP, INTER, and CP+DBS each achieved the fastest running time on at least one input and were significantly faster

Table 1 Running times of the different algorithms for computing MAAFs of the instances in the Poaceae data set. For each input consisting of "Tree 1" and "Tree 2", we list its number of taxa (n) and hybridization number (k).

Input				ILP	INTER		CP+	CP	K	CP+K
Tree 1	Tree 2	n	k		A	B	DBS			
ndhF	phyB	40	14	5s	20s	14s	13s	210s	>3h	206s
ndhF	rbcL	36	13	10s	3s	3s	16s	220s	>3h	218s
ndhF	rpoC2	34	12	7s	6s	3s	10s	559s	>3h	563s
ndhF	waxy	19	9	1s	<1s	1s	<1s	2s	56s	2s
ndhF	**ITS**	**46**	**19**	**51s**	**255s**	**1197s**	**78s**	**>3h**	**>3h**	**>3h**
phyB	rbcL	21	4	<1s	<1s	<1s	<1s	1s	6s	1s
phyB	rpoC2	21	7	3s	<1s	<1s	<1s	2s	222s	2s
phyB	waxy	14	3	1s	<1s	<1s	<1s	1s	1s	1s
phyB	ITS	30	8	1s	<1s	<1s	<1s	1s	>3h	1s
rbcL	rpoC2	26	13	14s	7s	5s	32s	662s	>3h	619s
rbcL	waxy	12	7	1s	<1s	<1s	<1s	2s	2s	2s
rbcL	**ITS**	**29**	**14**	**80s**	**586s**	**1979s**	**49s**	**>3h**	**>3h**	**>3h**
rpoC2	waxy	10	1	<1s	<1s	<1s	<1s	1s	1s	1s
rpoC2	ITS	31	15	115s	53s	1650s	17s	>3h	>3h	>3h
waxy	ITS	15	8	1s	<1s	<1s	<1s	6s	67s	6s

than K, CP, and CP+K. ILP and CP+DBS substantially outperformed INTER on the hardest inputs (ndhF/ITS and rbcL/ITS), highlighted in bold. These inputs have among the highest hybridization numbers in this data set, suggesting that INTER cannot keep up with ILP and CP+DBS as the hybridization number increases.

5 Conclusions

We investigated the impact of cluster partitioning on the performance of kernelization-based M(A)AF algorithms. Together with results for depth-bounded search reported in [27], our results support the following conclusions: (i) Cluster partitioning is by far the most important tool for obtaining fast M(A)AF algorithms. (ii) When used in conjunction with cluster partitioning, kernelization offers very little benefit and may even hurt performance due to the cost of computing the kernel. (iii) Depth-bounded search offers superior performance over kernelization. The exception is an approach that re-kernelizes the input after each branching step in the exponential search (INTER). Depth-bounded search and INTER have much in common in that the cherry and singleton reductions can be viewed as partially applying kernelization until it is safe to apply the next branching step.

Given the importance of cluster partitioning for the performance of M(A)AF algorithms, an important question is whether cluster partitioning can be improved further. When an input or large cluster cannot be split into smaller clusters, "long-distance" reticulations between distant taxa are often to blame. Empirical evidence suggests that most reticulations happen between fairly closely related taxa, so long-distance reticulations should be rare. If there exists an efficient algorithm for finding these long-distance reticulations, they could be eliminated, resulting in a modified input with only local reticulations that can therefore be split into small clusters for which M(A)AFs can be found efficiently. This would likely allow us to find M(A)AFs for larger and harder inputs currently well beyond our reach.

References

1. Benjamin Albrecht, Céline Scornavacca, Alberto Cenci, and Daniel H. Huson. Fast computation of minimum hybridization networks. *Bioinformatics*, 28(2):191–197, 2012.
2. M. Baroni, S. Grünewald, V. Moulton, and C. Semple. Bounding the number of hybridisation events for a consistent evolutionary history. *Journal of Mathematical Biology*, 51(2):171–182, 2005.
3. M. Baroni, C. Semple, and M. Steel. Hybrids in real time. *Systematic Biology*, 55:46–56, 2006.
4. Robert G. Beiko. Telling the whole story in a 10,000-genome world. *Biology Direct*, 6(1):34, 2011.
5. M. Bordewich and C. Semple. Computing the minimum number of hybridization events for a consistent evolutionary history. *Discrete Applied Mathematics*, 155(8):914–928, 2007.
6. Magnus Bordewich, Céline Scornavacca, Nihan Tokac, and Mathias Weller. On the fixed parameter tractability of agreement-based phylogenetic distances. *Journal of Mathematical Biology*, 74(1):239–257, 2017.
7. Magnus Bordewich and Charles Semple. On the computational complexity of the rooted subtree prune and regraft distance. *Annals of Combinatorics*, 8(4):409–423, 2005.
8. Magnus Bordewich and Charles Semple. Computing the hybridization number of two phylogenetic trees is fixed-parameter tractable. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(3):458–466, 2007.
9. Joshua Collins, Simone Linz, and Charles Semple. Quantifying hybridization in realistic time. *Journal of Computational Biology*, 18(10):1305–1318, 2011.
10. Grass Phylogeny Working Group. Phylogeny and subfamilial classification of the grasses (poaceae). *Annals of the Missouri Botanical Garden*, page 373–457, 2001.
11. D. M. Hillis, C. Moritz, and B. K. Mable, editors. *Molecular Systematics*. Sinauer Associates, 1996.
12. Leo van Iersel, Steven Kelk, Nela Lekić, and Leen Stougie. A short note on exponential-time algorithms for hybridization number. *CoRR*, abs/1312.1255, 2013.
13. Leo van Iersel, Steven Kelk, Nela Lekić, and Leen Stougie. Approximation algorithms for nonbinary agreement forests. *SIAM Journal on Discrete Mathematics*, 28(1):49–66, 2014.
14. Leo van Iersel and Simone Linz. A quadratic kernel for computing the hybridization number of multiple trees. *Information Processing Letters*, 113(9):318–323, 2013.
15. Steven Kelk and Céline Scornavacca. Towards the fixed parameter tractability of constructing minimal phylogenetic networks from arbitrary sets of nonbinary trees. *CoRR*, abs/1207.7034, 2012.
16. Zhijiang Li. Fixed-parameter algorithm for hybridization number of two multifurcating trees. Master's thesis, Faculty of Computer Science, Dalhousie University, 2015.
17. Zhijiang Li and Norbert Zeh. A fast algorithm for computing a soft hybridization network of two multifurcating trees. Manuscript in preparation.
18. Simone Linz and Charles Semple. Hybridization in nonbinary trees. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(1):30–45, 2009.
19. Simone Linz and Charles Semple. A cluster reduction for computing the subtree distance between phylogenies. *Annals of Combinatorics*, 15(3):465–484, 2011.
20. V. Rosas-Magallanes, P. Deschavanne, L. Qintana-Murci, R. Brosch, B. Gicquel, and O. Neyrolles. Horizontal transfer of a virulence operon to the ancestor of mycobacterium tuberculosis. *Molecular Biology and Evolution*, 23(6):1129–1135, 2006.
21. Heiko Schmidt. *Phylogenetic Trees from Large Datasets*. PhD thesis, Heinrich-Heine-Universität, Düsseldorf, Germany, 2003.

22 Feng Shi, Jie You, and Qilong Feng. Improved approximation algorithm for maximum agreement forest of two trees. In *Proceedings of the 8th International Workshop on Frontiers in Algorithmics*, pages 205–215, 2014.

23 Chris Whidden. *Efficient Computation of Maximum Agreement Forests and Their Applications*. PhD thesis, Faculty of Computer Science, Dalhousie University, 2013.

24 Chris Whidden, Robert G. Beiko, and Norbert Zeh. Fixed-parameter algorithms for maximum agreement forests. *SIAM Journal on Computing*, 42(4):1431–1466, 2013.

25 Chris Whidden, Robert G. Beiko, and Norbert Zeh. Fixed-parameter and approximation algorithms for maximum agreement forests of multifurcating trees. *Algorithmica*, 74(3):1019–1054, 2016.

26 Chris Whidden and Norbert Zeh. Computing the SPR distance of rooted binary trees in $O(2^k n)$ time. Manuscript in preparation.

27 Chris Whidden, Norbert Zeh, and Robert G. Beiko. Supertrees based on the subtree prune-and-regraft distance. *Systematic Biology*, 63(4):566–581, 2014.

28 Yufeng Wu and Jiayin Wang. Fast computation of the exact hybridization number of two phylogenetic trees. In *Proceedings of the 6th International Symposium on Bioinformatics Research and Applications*, pages 203–214. Springer-Verlag, 2010.

A Linear-Time Parameterized Algorithm for Node Unique Label Cover*

Daniel Lokshtanov[1], M. S. Ramanujan[2], and Saket Saurabh[3]

1 **University of Bergen, Bergen, Norway**
daniello@ii.uib.no
2 **Algorithms and Complexity Group, TU Wien, Vienna, Austria**
ramanujan@ac.tuwien.ac.at
3 **University of Bergen, Bergen, Norway; and**
The Institute of Mathematical Sciences, Chennai, India
saket@imsc.res.in

Abstract

The optimization version of the Unique Label Cover problem is at the heart of the Unique Games Conjecture which has played an important role in the proof of several tight inapproximability results. In recent years, this problem has been also studied extensively from the point of view of parameterized complexity. Chitnis et al. [FOCS 2012, SICOMP 2016] proved that this problem is fixed-parameter tractable (FPT) and Wahlström [SODA 2014] gave an FPT algorithm with an improved parameter dependence. Subsequently, Iwata, Wahlström and Yoshida [SICOMP 2016] proved that the *edge* version of Unique Label Cover can be solved in *linear* FPT-time, and they left open the existence of such an algorithm for the *node* version of the problem. In this paper, we resolve this question by presenting the first linear-time FPT algorithm for Node Unique Label Cover.

1998 ACM Subject Classification G.2.1 Combinatorics, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Algorithms and data structures, Fixed Parameter Tractability, Unique Label Cover, Linear Time FPT Algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.57

1 Introduction

In the Unique Label Cover problem we are given an undirected graph G, where each edge $uv = e \in E(G)$ is associated with a permutation $\phi_{e,u}$ of a constant size alphabet Σ. The goal is to construct a labeling $\Psi : V(G) \setminus X \to \Sigma$ maximizing the number of edge constraints, that is, edges for which $(\Psi(u), \Psi(v)) \in \phi_{uv,u}$ holds. For some $\epsilon > 0$ and given Unique Label Cover instance L, Unique Label Cover(ϵ) is the decision problem of distinguishing between the following two cases: (a) there is a labeling Ψ under which at least $(1 - \epsilon)|E(G)|$ edges are satisfied; and (b) for every labeling Ψ at most $\epsilon|E(G)|$ edges are satisfied. This problem is at the heart of famous Unique Games Conjecture (UGC) of Khot [27]. Essentially, UGC says that for any $\epsilon > 0$, there is a constant M such that it is NP-hard to decide Unique Label Cover(ϵ) on instances with label set of size M. The

* Daniel Lokshtanov acknowledges support from *Pareto-Optimal Parameterized Algorithms*, ERC Starting Grant 715744. M. S. Ramanujan acknowledges support from FWF, project P26696. Saket Saurabh acknowledges support from *Parameterized Approximation*, ERC Starting Grant 306992.

Unique Label Cover(ϵ) problem over the years has become a canonical problem to obtain tight inapproximability results. We refer the reader to a survey of Khot [28] for more detailed discussion on UGC.

In recent times Unique Label Cover has also attracted a lot of attention in the realm of parameterized complexity. In particular two parameterizations, namely, Edge Unique Label Cover and Node Unique Label Cover have been extensively studied. These problems are, not only, interesting combinatorial problems on its own but they also generalize several well-studied problems in the realm of parameterized complexity. The objective of this paper is to study the following problem.

Node Unique Label Cover **Parameter:** $|\Sigma| + k$

Input: A simple graph G, finite alphabet Σ, integer k and for every edge $e = (u, v) \in E(G)$, permutations $\phi_{e,u}$ and $\phi_{e,v}$ of Σ such that $\phi_{e,u} = \phi_{e,v}^{-1}$ and a function $\tau : V(G) \to 2^{\Sigma}$.

Question: Does there exist a set $X \subseteq V(G)$ and a function $\Psi : V(G) \setminus X \to \Sigma$ such that $|X| \leq k$ and for any $v \in V(G) \setminus X$, for any $(u, v) \in E(G - X)$, we have $(\Psi(u), \Psi(v)) \in \phi_{uv,u}$ where $\Psi(v) \in \tau(v)$ for every $v \in V(G)$?

We remark that the standard formulation of this problem excludes the function τ. However, this formulation is a clear generalization of the standard formulation (simply set $\tau(v) = \Sigma$ for every vertex v) and the way we describe our algorithm makes it notationally convenient to deal with this statement. To make the presentation simpler, we assume that $\Sigma = [|\Sigma|] = \{1, \ldots, |\Sigma|\}$.

The parameterized complexity of the Node Unique Label Cover problem was first studied by Chitnis et al. [5] who proved it is FPT by giving an algorithm running in time $2^{\mathcal{O}(k^2 \cdot \log |\Sigma|)} n^4 \log n$. They complemented this result by proving that an FPT algorithm for this problem parameterized only by k is unlikely to exist. Subsequently, Wahlström [38] (see also [25]) improved the parameter dependence by giving an algorithm running in time $\mathcal{O}(|\Sigma|^{2k} n^{\mathcal{O}(1)})$. The *edge* version of this problem was proved to be solvable in FPT-*linear* time by Iwata et al. [25] who gave an algorithm running in time $\mathcal{O}(|\Sigma|^{2k}(m + n))$. However, their approach does not apply to the much more general *node* version of the problem and they asked whether there is an FPT algorithm for the node version with a linear time dependence on the input size. In this paper, we answer this question in the affirmative by giving a linear time FPT algorithm for this problem. Note that we have stated the problem in a slightly more general form than is usually seen in literature. However, this modification does not affect the solvability of the problem in linear FPT time. We now state our theorem formally.

▶ **Theorem 1.1.** *There is a* $2^{\mathcal{O}(k \cdot |\Sigma| \log |\Sigma|)}(m + n)$ *algorithm solving* Node Unique Label Cover*, where* m *and* n *are the number of edges and vertices respectively in the input graph.*

Not only does our result answer the open question of Iwata et al. [25], when the label set Σ is of constant-size for some fixed constant, our algorithm also achieves optimal asymptotic dependence on the budget k under the Exponential Time Hypothesis [22].

By its very nature, the Node Unique Label Cover problem is a problem about breaking various types of dependencies between vertices. Since these dependencies are propagated along edges, it is reasonable to view the problem as breaking these dependencies by hitting appropriate sets of paths in the graph. Chitnis et al. [5] used this idea to argue that highly connected pairs of vertices will always remain dependent on each other and hence one can recursively solve the problem by first designing an algorithm for graphs that are 'nearly' highly connected and then use this algorithm as a base case in a divide and conquer type

approach. However, the polynomial dependence of their algorithm is $\mathcal{O}(n^4 \log n)$ where n is the number of vertices in the input. Subsequently, Wahlström [38] improved the parameter dependence by using a branching algorithm based on the solution to a specific linear program. However, since this algorithm requires solving linear programs, the dependence on the input is far from linear. Iwata et al. [25] showed that for several special kinds of LP-relaxations, including those involved in the solution of the *edge* version of Unique Label Cover, the corresponding linear program can be solved in linear-time using flow-based techniques and hence they were able to obtain the first linear-time FPT algorithm for the edge version of Unique Label Cover. However, their approach fails when it comes to the *node* version of this problem.

Our Techniques. In this paper, we view the Node Unique Label Cover problem as a problem of hitting paths between certain pairs of vertices in an appropriately designed *auxiliary* graph H whose size is greater than that of the input graph G by a factor depending only on the parameter. The high level road map for the solution follows those in the algorithms developed for solving graph separation problems via important separators in [31, 4], the LP-guided branching in [14, 6, 29, 23], the Valued CSP-based algorithms in [38, 25], the skew-symmetric branching algorithm for 2-SAT Deletion in [33] and most recently, the branching algorithm for the *edge* version of Group Feedback Vertex Set [32]. We show that for any prescribed labeling on the vertices of G, it is possible to select (in linear time) a constant-size set of vertices of G such that after guessing the intersection of this set with a hypothetical solution, if we augment the labeling by branching over all permitted labelings of the remaining vertices in this set then we reduce a pre-determined measure of the input which depends only on the parameter. By repeatedly doing this, we obtain a branching algorithm for this problem where each step requires linear time. The main technical content of the paper is in proving that

(a) there exists a constant-size vertex set and an appropriate measure for the instance such that the measure 'improves' in each step of the branching and
(b) such a vertex set can be computed in linear time.

Related work on improving dependence on input size in FPT algorithms. Our algorithm for Node Unique Label Cover belongs to a large body of work where the main goal is to design linear time algorithms for NP-hard problems for a fixed value of k. That is, to design an algorithm with running time $f(k) \cdot \mathcal{O}(|I|)$, where $|I|$ denotes the size of the input instance. This area of research predates even parameterized complexity. The genesis of parameterized complexity is in the theory of graph minors, developed by Robertson and Seymour [35, 36, 37]. Some of the important algorithmic consequences of this theory include $\mathcal{O}(n^3)$ algorithms for Disjoint Paths and $\mathcal{F}$-Deletion for every fixed values of k. These results led to a whole new area of designing algorithms for NP-hard problems with as small dependence on the input size as possible; resulting in algorithms with improved dependence on the input size for Treewidth [1, 2], FPT approximation for Treewidth [3, 34], Planar $\mathcal{F}$-Deletion [1, 2, 8, 10, 9], and Crossing Number [11, 12, 19], to name a few.

The advent of parameterized complexity started to shift the focus away from the running time dependence on input size to the dependence on the parameter. That is, the goal became designing parameterized algorithms with running time upper bounded by $f(k)n^{\mathcal{O}(1)}$, where the function f grows as slowly as possible. Over the last two decades researchers have tried to optimize one of these objectives, but rarely both at the same time. More recently, efforts have been made towards obtaining linear (or polynomial) time parameterized algorithms that

compromise as little as possible on the dependence of the running time on the parameter k. The gold standard for these results are algorithms with linear dependence on input size as well as provably optimal (under ETH) dependence on the parameter. New results in this direction include parameterized algorithms for problems such as Odd Cycle Transversal [24, 33], Subgraph Isomorphism [7], Planarization [26, 15], Subset Feedback Vertex Set [30] as well as a single-exponential and linear time parameterized constant factor approximation algorithm for Treewidth [3]. Other recent results include parameterized algorithms with improved dependence on input size for a host of problems [13, 16, 17, 18, 20, 21].

2 Preliminaries

We fix a label set Σ and assume that all instances of Node Unique Label Cover we deal with are over this label set. When we refer to a set X being a *solution* for a given instance of Node Unique Label Cover, we implicitly assume that X is a set of *minimum* size. We denote the set of functions $\{\phi_{e,u}\}_{e\in E(G),u\in e}$ simply as ϕ (without any subscript).

Before we proceed to describe our algorithm for Node Unique Label Cover, we make a few remarks regarding the representation of the input. We assume that the input graph is given in the form of an adjacency list and for every edge $e = (u, v)$ the permutations $\phi_{e,v}$ and $\phi_{e,u}$ are included in the two nodes of the adjacency list corresponding to the edge e. This is achieved by representing the permutations as $|\Sigma|$-length arrays over the elements in $[|\Sigma|]$. It is straightforward to check that given the input to Label Cover in this form, the decision version of the problem can be solved in time $\mathcal{O}(|\Sigma|^{\mathcal{O}(1)}(m + n))$. We assume that the input to Node Unique Label Cover is also given in the same manner.

3 Setting up the tools

3.1 Defining the auxiliary graph

▶ **Definition 3.1.** Let (G, k, ϕ, τ) be an instance of Node Unique Label Cover and let $\Psi : V(G) \to \Sigma$. We say that Ψ is a **feasible labeling** for this instance if for all $(u, v) \in E(G)$, $(\Psi(u), \Psi(v)) \in \phi_{uv,u}$. For $\tau : V(G) \to 2^{\Sigma}$, we say that Ψ is **consistent with** τ if for every $v \in V(G)$, $\Psi(v) \in \tau(v)$.

For an instance $I = (G, k, \phi, \tau)$ of Node Unique Label Cover, we define an associated auxiliary graph H_I as follows. The vertex set of H_I is $V(G) \times \Sigma$. For notational convenience, we denote the vertex (v, i) by v_i. The vertex v_i is meant to represent the (eventual) labeling of v by the label i. The edge set of H_I is defined as follows. For every edge $e = (u, v)$ and for every $i \in \Sigma$, we have an edge $(u_i, v_{\phi_{e,u}(i)})$. That is, we add an edge between u_i and u_j where j is the image of i under the permutation $\phi_{e,u}$.

We now prove certain structural lemmas regarding this auxiliary graph which will be used in the design as well as analysis of our algorithm. For ease of description, we will treat instances of Label Cover as instances of Node Unique Label Cover. To be precise, we represent an instance (G, ϕ) of Label Cover as the trivially equivalent instance $(G, 0, \phi, \tau^0)$ of Node Unique Label Cover where, $\tau^0(v) = \Sigma$ for every $v \in V(G)$. The first observation follows from the definition of H_I and the fact that since G is a simple graph, for every edge $e \in E(G)$, the set of edges in H_I that correspond to this edge form a matching.

▶ **Observation 3.2.** *Let $I = (G, 0, \phi, \tau)$ be an instance of* Node Unique Label Cover. *Then, for every $v \in V(G)$, for every distinct $i, j \in \Sigma$, v_i and v_j have no common neighbors in H_I.*

▶ **Observation 3.3.** *Let $I = (G, 0, \phi, \tau)$ be a* Yes *instance of* Node Unique Label Cover *and let Ψ be a feasible labeling for this instance. Let $v \in V(G)$ and $i = \Psi(v)$. Then, for any vertex $u \in V(G)$ and $j \in \Sigma$, if u_j is in the same connected component as v_i in H_I then $\Psi(u) = j$.*

The above observation describes the 'dependency' between pairs of vertices which are in the same connected component of G. Moving forward, we will characterize the dependencies between vertices when subjected to additional constraints.

▶ **Definition 3.4.** Let $I = (G, k, \phi, \tau)$ be an instance of Node Unique Label Cover. For $v \in V(G)$, we use $[v]$ to denote the set $\{v_1, \ldots, v_{|\Sigma|}\}$. For a subset $S \subseteq V(G)$, we use $[S]$ to denote the set $\bigcup_{v \in S}[v]$. Similarly, for $e = (u, v) \in E(G)$, we use $[e]$ to denote the set $\{(u_i, v_j)\}_{i \in \Sigma, j = \phi_{e,u}(i)}$ of edges and for a subset $X \subseteq E(G)$, we use $[X]$ to denote the set $\bigcup_{e \in X}[e]$. For the sake of convenience, we also reuse the same notation in the following way. For $v \in V(G)$ and $\alpha \in \Sigma$, we also use $[v_\alpha]$ to denote the set $\{v_1, \ldots, v_{|\Sigma|}\}$. This definition extends in a natural way to sets of vertices and edges of the auxiliary graph H_I. Finally, for a set $S \subseteq V(H_I) \cup E(H_I)$, we denote by S^{-1} the set $\{s | s \in V(G) \cup E(G) : [s] \cap S \neq \emptyset\}$.

▶ **Definition 3.5.** Let $I = (G, k, \phi, \tau)$ be an instance of Node Unique Label Cover. We say that a set $Z \subseteq V(H_I) \cup E(H_I)$ is **regular** if $|Z \cap [v]| \leq 1$ for any $v \in V(G)$ and $|Z \cap [e]| \leq 1$ for any $e \in V(G)$ and **irregular** otherwise. That is, regular sets contain at most 1 copy of any vertex and edge of G.

Now that we have defined the notion of regularity of sets, we prove the following lemma which shows that the auxiliary graph displays a certain symmetry with respect to regular paths. This will allow us to transfer arguments which involve a regular path between vertices v_i and u_j to one between vertices v_{i_1} and u_{j_1} where $i \neq i_1$ and $j \neq j_1$.

▶ **Lemma 3.6.** *Let $I = (G, k, \phi, \tau)$ be an instance of* Node Unique Label Cover. *Let P be a regular path in H_I from v_i to u_j. Let $V(P)$ denote the set of vertices of G in P and let U denote the set $[V(P)]$. Then, there are vertex disjoint paths $P_1, \ldots, P_{|\Sigma|}$ in H_I and a partition of U into sets $U_1, \ldots, U_{|\Sigma|}$ such that for each $r \in [|\Sigma|]$, $V(P_r) = U_r$ and P_r is a path from v_{i_1} to u_{i_2} for some $i_1, i_2 \in \Sigma$.*

In the next lemma, we describe additional structural properties of the auxiliary graph. In particular, we establish the relation between various copies of the same vertex set. Intuitively, the following lemma says that for every connected and regular set of vertices Z, simply observing the set $N[Z]$ can allow one to make certain useful assertions about the set of vertices in the neighborhood of the set $Z' = [Z] \setminus Z$. Note that for a graph H and set $Z \subseteq V(H)$, we use $N_H[Z]$ and $N_H(Z)$ to denote the closed and open neighborhoods of Z in H respectively. If H is clear from the context, then we drop the subscript.

▶ **Lemma 3.7.** *Let $Z \subseteq V(H_I)$ be a connected regular set of vertices and let $Y = N(Z)$. Further, suppose that $N[Z]$ is regular. Let $Z' = [Z] \setminus Z$ and $Y' = [Y] \setminus Y$. Then, $Y' \subseteq N(Z') \subseteq [Y]$. Furthermore, for every connected component C in $H_I[Z']$, $N(C) \cap [v] \neq \emptyset$ for every $v \in V(G)$ for which there is a $j \in \Sigma$ such that $v_j \in Y$.*

Using the observations and structural lemmas proved so far, we will now give a forbidden-structure characterization of Yes instances of Node Unique Label Cover.

▶ **Lemma 3.8.** *Let $I = (G, 0, \phi, \tau)$ be a* Yes *instance of* Node Unique Label Cover *where G is connected. Let $v \in V(G)$ and $i \in \Sigma$. Then, there is a feasible labeling Ψ such that $\Psi(v) = i$ if and only if there is no $j \in \Sigma$ such that v_i and v_j are in the same connected component of H_I.*

So far, we have studied the structure of YES instances of this problem when the budget $k = 0$. The next lemma is a direct consequence of Lemma 3.8 and allows us to characterize YES instances of the problem for values of k greater than 0.

▶ **Lemma 3.9.** *Let $I = (G, k, \phi, \tau)$ be an instance of* NODE UNIQUE LABEL COVER. *Then, I is a* YES *instance if and only if there is a set $S \subseteq V(G)$ of at most k vertices such that for every $v \in V(G) \setminus S$, there is an $i_v \in \Sigma$ such that $[S]$ intersects all paths from v_{i_v} to v_j for every $\Sigma \ni j \neq i_v$ in the graph H_I. Moreover if there is a feasible labeling for $G - S$ consistent with τ that labels v with the label $i \in \tau(v)$ then for every $u \in V(G)$ and $j \in \Sigma \setminus \tau(u)$, $[S]$ intersects all v_i-u_j paths.*

Using the above lemma, we will interpret the NODE UNIQUE LABEL COVER problem as a parameterized cut-problem and use separator machinery to design a linear-time FPT algorithm for this problem.

3.2 Defining the associated cut-problem

We begin by recalling standard definitions of separators in undirected graphs.

▶ **Definition 3.10.** Let G be a graph and X and Y be disjoint vertex sets. A set S disjoint from $X \cup Y$ is said to be an X-Y **separator** if there is no X-Y path in the graph $G - S$. We denote the vertices in the components of $G - S$ which intersect X by $R(X, S)$ and we denote by $R[X, S]$ the set $R(X, S) \cup S$. We say that an X-Y separator S_1 **covers** an X-Y separator S_2 if $R(X, S_1) \supseteq R(X, S_2)$.

▶ **Definition 3.11.** Let I be an instance of NODE UNIQUE LABEL COVER and let X and Y be disjoint vertex sets of H_I. We say that a minimal X-Y separator S is **good** if the set $R[X, S]$ is regular and **bad** otherwise.

Note that if S is a minimal X-Y separator then $N(R(X, S)) = S$. We are now ready to prove the *Persistence* Lemma which plays a major role in the design of the algorithm. In essence this lemma says that if we are guaranteed the existence of a solution whose deletion leaves a graph with a feasible labeling Ψ and if we are given a vertex v excluded from the deletion set which has a single label α in its allowed label set, then we can define a set T such that the solution under consideration *must* separate v_α from T. Furthermore, if we find a *good* minimum v_α-T separator S, then we can correctly fix the labels of all vertices which have exactly one copy in $R(v_\alpha, S)$. It will be shown later that once we fix the labels of these vertices, the subsequent exhaustive branching steps will decrease a pre-determined measure of the input instance.

▶ **Lemma 3.12** (Persistence Lemma)**.** *Let $I = (G, k, \phi, \tau)$ be a* YES *instance of* NODE UNIQUE LABEL COVER. *Let $X \subseteq V(G)$ be a minimal set of size at most k such that $G - X$ has a feasible labeling and let Ψ be a feasible labeling for $G - X$ consistent with τ. Let v be a vertex not in X with $|\tau(v)| = 1$ and let $\alpha \in \Sigma$ be such that $\alpha = \Psi(v)$ and $\tau(v) = \{\alpha\}$. Let T denote the set $\bigcup_{u \in V(G)} \bigcup_{\gamma \in \Sigma \setminus \tau(u)} \{u_\gamma\}$.*

- *$[X]$ is a v_α-T separator in H_I.*
- *Let S be a good v_α-T minimum separator in H_I and let $Z = R(v_\alpha, S)$. Then, there is a solution for the given instance disjoint from Z^{-1}.*

Proof. The first statement follows from Lemma 3.9. We now prove the second statement. We begin by observing that T contains the set $[v] \setminus \{v_\alpha\}$. This is because $\tau(v)$ is a singleton and only contains the label α. As a result, we know that the set $[X]$ must intersect all v_α-v_β

paths for $\alpha \neq \beta$. Let X_1 denote the set $X \cap Z^{-1}$. If X_1 is empty then we are already done. Therefore, $X_1 \neq \emptyset$. Let S' denote the subset of $S \setminus [X]$ which is not reachable from v_α in the graph $H_I - [X]$ via paths whose internal vertices lie in Z. We now have 2 cases depending on S' being empty or non-empty. We will argue that the first case cannot occur since it contradicts the minimality of X. In the second case we use very similar arguments but show that we can modify X to get an alternate solution X' which is disjoint from the set Z.

Case 1: S' is empty. That is, every vertex in $S \setminus [X]$ is reachable from v_α in $H_I - [X]$ via paths whose internal vertices lie in Z. Let $u \in X_1$ and let $b \in \Sigma$ such that $u_b \in Z$. Since Z is regular, $Z \cap [u]$ must in fact be equal to $\{u_b\}$. We now claim that $X' = X \setminus \{u\}$ is also a set such that $G - X'$ has a feasible labeling, contradicting the minimality of X.

Suppose that this is not the case. That is, $G - X'$ does not have a feasible labeling. Since every connected component of $G - X'$ which does not contain u is also a connected component of $G - X$, all such components do have a feasible labeling. Indeed any feasible labeling of $G - X$ restricted to the vertices in these components is a feasible labeling for these components. Therefore, there is a single component in $G - X'$ which does not have a feasible labeling – the component containing u.

By Lemma 3.8, if there is no $b' \in \Sigma \setminus \{b\}$ such that the connected component of $H_I - [X']$ containing u_b also contains $u_{b'}$, then there is a feasible labeling of the component of $G - X'$ which contains u, a contradiction. Therefore, there is a $b' \in \Sigma \setminus \{b\}$ such that there is a u_b-$u_{b'}$ path in $H_I - [X']$. If this path contains vertices of $[u]$ other than u_b and $u_{b'}$, then we pick the vertex of $[u] \setminus \{u_b\}$ which is closest to u_b on this path and call it $u_{b'}$. Therefore, the path P from u_b to $u_{b'}$ is internally disjoint from $[u]$. We now have the following claim regarding P.

▶ **Claim 3.13.** *The path P is internally regular.*

We now return to the proof of the first case. Since $u_b \in Z$ and $u_{b'} \notin Z$ (as $N[Z]$ is regular), P must intersect $N(Z)$ which is the same as S, in $S \setminus [X]$. Furthermore, P must intersect $N(C)$ where C is the connected component of $Z' = [Z] \setminus Z$ containing the vertex $u_{b'}$. We now have the following 2 subcases based on the intersection of P with the(not necessarily non-empty) set $S \cap N(C)$. In both subcases we will demonstrate the presence of a v_α-v_β path in $H_I - [X]$ for some $\beta \in \Sigma \setminus \{\alpha\}$.

Case 1.1: *P contains a vertex in $S \cap N(C)$.* Let w_ℓ be a vertex in $S \cap N(C)$ which appears in P. We let P_1 denote the subpath of P from u_b to w_ℓ and P_2 denote the subpath of P from w_ℓ to $u_{b'}$. Furthermore, since P is internally regular, P_1 and P_2 are regular. We apply Lemma 3.6 to the regular path P_2 to get a path P_2' with u_b as one endpoint and w_h as the other endpoint, where $w_h \neq w_\ell$. Now, since $W_\ell \in N[Z]$ and $N[Z]$ is regular by our assumption, it must be the case that $w_h \notin Z$. Therefore the path P_2' must intersect S at a vertex other than w_ℓ. Let x_r be such a vertex, where $x \in V(G)$ and $r \in \Sigma$. However, in the case we are in, we know that x_r (which is contained in $S \setminus [X]$) is reachable from v_α in $H_I - [X]$ by a path Q whose internal vertices lie in Z. We let the subpath of P_2' from x_r to w_h be denoted by J. Furthermore, the case we are in guarantees that w_ℓ is reachable from v_α in $H_I - [X]$ via a path L whose internal vertices lie in Z. Since L lies completely in $N[Z]$, it is regular and we may apply Lemma 3.6 on this path to obtain a path L' with w_h s one endpoint and v_β as the other endpoint for some $\beta \in \Sigma$. Since we have already argued that $w_h \neq w_\ell$, it follows that $\beta \neq \alpha$. Therefore, we get a concatenated walk $Q + J + L'$ which is a walk that is present in the graph $H_I - [X]$ and contains v_α and v_β, contradicting the premise

of the lemma that there is a feasible labeling for $G - X$ setting v to α. This completes the argument for this subcase.

Case 1.2: *P does not contain a vertex in $S \cap N(C)$.* Let x_r be the last vertex of S which is encountered when traversing P from u_b to $u_{b'}$ and let w_ℓ be the last vertex of $N(C)$ encountered in the same traversal. Observe that since the previous subcase does not hold, it must be the case that x_r occurs before w_ℓ in this traversal. We let J denote the subpath of P between x_r and w_ℓ. Now, Lemma 3.7 implies that there is a $h \in \Sigma \setminus \{\ell\}$ such that $w_h \in S$. This is because $N(C) \subseteq [S]$. Now, the case we are in guarantees the presence of paths L and Q from v_α to w_h and x_r respectively such that L and Q both lie strictly inside $N[Z]$ and hence are regular. Now, we apply Lemma 3.6 on the regular path J to get a path J' with w_h as one endpoint and x_{r_1} as the other for some $r_1 \in \Sigma$. Since we have already argued that $w_h \neq w_\ell$, it must be the case that $r_1 \neq r$. Now, we apply Lemma 3.6 on the regular path Q to get a path Q' with x_{r_1} as one endpoint and v_β as the other for some $\beta \in \Sigma$. Since we have shown that $r_1 \neq r$, we infer that $\beta \neq \alpha$. Now, the concatenated walk $L + J' + Q'$ implies the presence of a v_α-v_β path in $H_I - [X]$, a contradiction to the premise of the lemma. This completes the argument for this subcase.

Thus we have concluded that $G - X'$ has a feasible labeling, contradicting the minimality of X. This completes the argument for the first case.

Case 2: S' is non-empty. Let $\mathcal{Q}$ be a set of $|S|$-many v_α-S paths contained entirely in $N[Z]$ which are vertex disjoint except for the vertex v_α. Since S is a minimum v_α-T separator, such a set of paths exists. Recall that X_1 denotes the set $X \cap Z^{-1}$. We let $\hat{X}_1$ denote the set $[X] \cap Z$. That is, those copies of X_1 present in Z. Due to the presence of the set of paths $\mathcal{Q}$ and the fact that v is disjoint from X, it must be the case that $\hat{X}_1$ contains at least one vertex in each path in $\mathcal{Q}$ that connects v and S'. Furthermore, since S is a good separator, we conclude that $|X_1| = |(\hat{X}_1)^{-1}| \geq |(S')^{-1}|$. We now claim that $X' = (X \setminus X_1) \cup (S')^{-1}$ is also a solution for the given instance. That is, $|X'| \leq |X|$ and $G - X'$ has a feasible labeling. By definition, $|X'| \leq |X|$ holds. Therefore, it remains to prove that $G - X'$ has a feasible labeling.

Again, it must be the case that any connected component of $G - X'$ which does not have a feasible labeling must intersect the set X_1. Any other component of $G - X'$ is contained in a component of $G - X$ and already has a feasible labeling by the premise of the lemma.

By Lemma 3.8, there must be a vertex $u^1 \in X_1$ and distinct labels $b, b' \in \Sigma$ such that $u^1_b \in Z$ and there is a $u^1_b - u^1_{b'}$ path P in $H_I - [X']$. We now consider the intersection of P with the set $[X_1]$ and let p_{γ_1} and q_{γ_2} be vertices on P such that $p_{\gamma_1}, q_{\gamma_2} \in [Z]$, the subpath of P from p_{γ_1} to q_{γ_2} is internally disjoint from $[X_1]$ *and* $p_{\gamma_1} \in Z$ and $q_{\gamma_2} \notin Z$. We first argue that such a pair of vertices exist.

We begin by setting $p_{\gamma_1} = u^1_b$ and $q_{\gamma_2} = u^1_{b'}$. If the path P is already internally disjoint from $[X_1]$ then we are done. Otherwise, let u^2_c be the vertex of $[X_1]$ closest to p_{γ_1} along the subpath between p_{γ_1} and q_{γ_2}. Now, if u^2_c is not in Z then we are done by setting $q_{\gamma_2} = u^2_c$. Otherwise, we continue by setting $p_{\gamma_1} = u^2_c$. Since this process must terminate, we conclude that the vertices p_{γ_1} and q_{γ_2} with the requisite properties must exist.

For ease of notation we will now refer to the path between p_{γ_1} and q_{γ_2} as P. Note that by definition, P is internally disjoint from $[X_1]$. We now have a claim identical to that in the previous case.

▶ **Claim 3.14.** *The path P is internally regular.*

We now complete the proof of this case. Since $p_{\gamma_1} \in Z$ and $q_{\gamma_2} \in [Z] \setminus Z$, P must intersect $N(Z)$ in $(S \setminus [X]) \setminus S'$. Furthermore, P must also intersect $N(C)$ where C is the connected component of $H_I[Z']$ containing q_{γ_2}, where $Z' = [Z] \setminus Z$. We again consider 2 subcases based on the intersection of the path P with the (not necessarily non-empty) set $N(C) \cap S$.

Case 2.1: *P contains a vertex in $S \cap N(C)$.* Let w_ℓ be a vertex in $S \cap N(C)$ which appears in P. We let P_1 denote the subpath of P from p_{γ_1} to w_ℓ and P_2 denote the subpath of P from w_ℓ to q_{γ_2} . Since P is internally regular, P_1 and P_2 are regular. Furthermore, since $q_{\gamma_2} \notin Z$, there is a $\gamma_3 \in \Sigma \setminus \{\gamma_2\}$ such that $q_{\gamma_3} \in Z$. We now apply Lemma 3.6 on the regular path P_2 to get a path P_2' with q_{γ_3} as one endpoint and w_h as the other, where $h \neq \ell$ since $\gamma_2 \neq \gamma_3$. Furthermore, since $w_\ell \in N[Z]$ and $N[Z]$ is regular, it must be the case that $w_h \notin Z$. Therefore the path P_2' must intersect $N(Z)$ at a vertex x_r. Let J be the subpath of P_2' from x_r to w_h. Now, since $x_r \in (S \setminus [X]) \setminus S'$, we know that there is a v_α-x_r path in $H_I - [X]$ which lies entirely in $N[Z]$. Let Q be such a path. Similarly, we know that there is a v_α-w_ℓ path L in $H_I - [X]$ which also lies entirely in $N[Z]$ and hence is regular. We now apply Lemma 3.6 on L to get a path L' with w_h as one endpoint and v_β as the other endpoint for some $\beta \in \Sigma$. Since we have already argued that $w_h \neq w_\ell$, we conclude that $\beta \neq \alpha$. However, the concatenated walk $Q + J + L'$ is present in $H_I - [X]$, implying a v_α-v_β path in $H_I - [X]$, a contradiction to the premise of the lemma. We now address the second subcase under the assumption that this subcase does not occur.

Case 2.2: *P does not contain a vertex in $S \cap N(C)$.* Let x_r be the last vertex of S which is encountered when traversing P from p_{γ_1} to q_{γ_2} and let w_ℓ be the last vertex of $N(C)$ encountered in the same traversal. Since the previous subcase is assumed to not hold, x_r must occur before w_ℓ in this traversal. We let J denote the subpath of P between x_r and w_ℓ. Lemma 3.7 implies the existence of a label $h \in \Sigma \setminus \{\ell\}$ such that $w_h \in S$. This follows from the fact that $N(C) \subseteq [S]$. Also, since w_ℓ occurs in P, w_h is not contained in S' or $[X]$. The same holds for x_r Therefore, the case we are in guarantees the presence of paths L and Q from v_α to w_h and x_r respectively, where L and Q are contained within the set $N[Z]$ and hence they must be regular and amenable to applications of Lemma 3.6. We begin by applying Lemma 3.6 on the regular path J to get a path J' with w_h as one endpoint and x_{r_1} as the other for some $r_1 \in \Sigma$. However, since $h \neq \ell$, we conclude that $r_1 \neq r$. Therefore, we now apply Lemma 3.6 on the path Q to obtain a path Q' with x_{r_1} as one endpoint with the other endpoint being v_β for some $\beta \in \Sigma$. Again, since $r_1 \neq r$, we conclude that $\beta \neq \alpha$. Now, observe that the concatenated walk $L + J' + Q'$ implies the presence of a v_α-v_β path in $H_I - [X]$, a contradiction to the premise of the lemma. This completes the argument for this subcase as well and consequentially that for Case 2.

We have thus proved that Case 1 cannot occur at all and in Case 2, there is an exchange argument which constructs an alternate solution X' which is disjoint from Z. This completes the proof of the lemma. ◀

The main consequence of the above lemma is that at any point in the run of our algorithm solving an instance $I = (G, k, \phi, \tau)$, if there is a vertex v whose label is 'fixed', i.e. $\tau(v) = \{\alpha\}$ for some $\alpha \in \Sigma$ and there is a good v_α-T separator S where T is defined as in the premise of the above lemma, then we can correctly 'fix' the labelings of all vertices in the set $(R(v_\alpha, S))^{-1}$. That is, we can define a new function τ' as follows. For every $u \in V(G)$ and $\gamma \in \Sigma$, we set $\tau'(u) = \{\gamma\}$ if $u_\gamma \in R(v_\alpha, S)$ and $\tau'(u) = \tau(u)$ otherwise. Lemma 3.12 implies that the given graph has a deletion set of size at most k which leaves a graph with a feasible labeling consistent with τ if and only if the graph has deletion set of size at most k which leaves a graph with a feasible labeling consistent with τ'.

3.3 Computing good separators

▶ **Lemma 3.15.** *Let $I = (G, k, \phi, \tau)$ be an instance of* Node Unique Label Cover*, v be a vertex in G and let $\alpha \in \Sigma$. Let T_v^α denote the set $[v] \setminus \{v_\alpha\}$ and $T \supseteq T_v^\alpha$ be a set not containing v_α. There is an algorithm that, given I, v, α, and T runs in time $\mathcal{O}(|\Sigma| \cdot k(m+n))$ and either*

- *correctly concludes that there is no v_α-T separator of size at most $|\Sigma| \cdot k$ or*
- *returns a pair of minimum v_α-T separators S_1 and S_2 such that S_2 covers S_1, S_1 is good, S_2 is bad and for any vertex $u \in R(v_\alpha, S_2) \setminus R[v_\alpha, S_1]$, the size of a minimal v_α-T separator containing u is at least $|S_1| + 1$ or*
- *returns a good minimum v_α-T separator S such that no other minimum v_α-T separator covers S or*
- *correctly concludes that there is no good v_α-T minimum separator.*

▶ **Lemma 3.16.** *Let $I = (G, k, \phi, \tau)$ be an instance of* Node Unique Label Cover*, v be a vertex in G, $\alpha \in \Sigma$, $T \supseteq [v] \setminus \{v_\alpha\}$ be a set not containing v_α and let $\ell > 0$ be the size of a minimum v_α-T separator in H_I. Let S_1 and S_2 be a pair of minimum v_α-T separators such that S_1 is good, S_2 is bad, and for any vertex $y \in R(v_\alpha, S_2) \setminus R[v_\alpha, S_1]$, the size of a minimal v_α-T separator containing y is at least $\ell + 1$. Let $u \in V(G)$ and $\gamma_1, \gamma_2 \in \Sigma$ such that $u_{\gamma_1}, u_{\gamma_2} \in R[v_\alpha, S_2]$. Then,*

1. *$R[v_\alpha, S_2]$ contains a pair of paths P_1 and P_2 such that for each $i \in \{1, 2\}$, the path P_i is a v_α-u_{γ_i} path and both paths are internally vertex disjoint from S_2 and contain at most one vertex of S_1.*
2. *Given I, v_α, S_1 and S_2, there is an algorithm that, in time $\mathcal{O}(|\Sigma| \cdot k(m+n))$, computes a pair of paths with the above properties.*
3. *For $i \in \{1, 2\}$, any minimum v_α-$T \cup \{u_{\gamma_i}\}$ separator disjoint from $V(P_i) \cap (S_1 \cup S_2)$ and $R(v_\alpha, S_1)$ has size at least $\ell + 1$, where ℓ is the size of a minimum v_α-T separator.*

We are now ready to prove Theorem 1.1 by describing our algorithm for Node Unique Label Cover. Before doing so, we make the following important remark regarding the way we use the algorithms described in this subsection. In the description of our main algorithm, there will be points where we make a choice to *not* delete certain vertices. That is, we will choose to exclude them from the solution being computed. At such points, we say that we make these vertices *undeletable*.

All the above algorithms also work when given an undeletable set of vertices in the graph and the minimum separators we are looking for are the minimum *among* those separators disjoint from the undeletable set of vertices. Regarding the running time of these algorithms, there will be a multiplicative factor of $|\Sigma| \cdot k$ which arises due to potentially blowing up the size of the graph by a factor of $|\Sigma| \cdot k$ by making $(|\Sigma| \cdot k) + 1$ copies of every undeletable vertex.

4 The Linear time algorithm for Node Unique Label Cover

Before we describe our algorithm, we state certain assumptions we make regarding the input. We assume that at any point, we are dealing with a connected graph G. Furthermore, we assume that instances of Node Unique Label Cover are given in the form of a tuple – $(G, k, \phi, \tau, w^*, V^\infty)$ where the element w^* denotes either a vertex from $V(G)$ or it is undefined. If w^* denotes a vertex then, $|\tau(w^*)| = 1$ and we will attempt to solve the problem on the tuple $(G, k, \phi, \tau, w^*, V^\infty)$ under the assumption that w^* is not in the solution (which is required to be disjoint from V^∞). Furthermore the definition of the problem allows us to

assume that if there is a feasible labeling for this instance (after deleting a solution) then there is one consistent with τ. Since $\tau(w^*)$ is singleton, any feasible labeling consistent with τ must set w^* to the unique label in $\tau(w^*)$.

We first check if G already has a feasible labeling (not necessarily one consistent with τ). If so, then we are done. If not and $k = 0$ then we return No. If any connected component of G has a feasible labeling then we remove this component. Otherwise, we check if w^* is defined. If w^* is undefined, then we pick an arbitrary *deletable* vertex $v \in V(G)$. That is $v \notin V^\infty$. We then recursively solve the problem on the instances $I_{q_0}, \ldots, I_{q_r}$ where $\{q_1, \ldots, q_r\} = \tau(v)$ and for each q_i where $i \geq 1$, the instance I_{q_i} is defined to be $(G, k, \phi, \tau_{v=q_i}, w^*, V_1^\infty)$ with $\tau_{v=q_i}$ defined as the function obtained from τ by restricting the image of v to the singleton set $\{q_i\}$, w^* defined as $w^* = v$ and V_1^∞ defined as $V_1^\infty = V^\infty \cup \{v\}$. The instance I_{q_0} is defined as $(G - \{v\}, k-1, \phi', \tau', w^*, V^\infty)$ where ϕ' and τ' are restrictions of ϕ and τ to the graph $G - \{v\}$. This will be the only branching rule which has a branching factor depending on the parameter (in this case the size of the label set Σ) and we call this rule, $\mathbf{B_0}$.

We now describe the steps executed by the algorithm in the case when w^* is defined. Suppose that $w^* = v$, $\tau(v) = \alpha$. Recall that by our assumption regarding well-formed inputs, if w^* is defined then $\tau(w^*)$ must be a singleton set. We set $T = \bigcup_{u \in V(G)} \bigcup_{\gamma \in \Sigma \setminus \tau(u)} u_\gamma$. Intuitively, T is the set of all vertices u_γ such that if there is a feasible labeling of G (after deleting the solution) which sets v to α then it cannot be consistent with τ unless the solution hits all paths in H_I (where I is the given instance) between v_α and u_γ. We remark that since T depends only on the input instance I, we use $T(I)$ to denote the set T corresponding to any input instance I. Once we set T as described we first check if there is a v_α-T path in H_I. If not, then the algorithm deletes the component of G containing v and recurses by setting w^* to be undefined. The correctness of this operation is argued as follows. Observe that T contains all vertices of $[v] \setminus \{v_\alpha\}$ and excludes v_α. Therefore, Lemma 3.8 implies that the component of G containing v already has a feasible labeling and hence can be removed.

Otherwise if there is a v_α-T path in H_I, then we execute the algorithm of Lemma 3.15 with this definition of v, α and T and undeletable set $[V^\infty]$. Observe that T contains all vertices of $[v] \setminus \{v_\alpha\}$ but excludes v_α. This is because $\tau(v) = \{\alpha\}$. The next steps of our algorithm depend on the output of this subroutine. For each of the four possible outputs, we describe an exhaustive branching.

Case 1: *The subroutine returns that there is no v_α-T separator of size at most $|\Sigma| \cdot k$ which is disjoint from $[V^\infty]$.* In this case, our algorithm returns No. The correctness of this step follows from Lemma 3.12.

Case 2: *The subroutine returns a good v_α-T separator S which is smallest among all v_α-T separators disjoint from $[V^\infty]$ such that no other v_α-T separator disjoint from $[V^\infty]$ and having the same size as S, covers S.* In this case, we do the following. For each vertex u_γ in the set $R(v_\alpha, S)$ where $u \in V(G)$ and $\gamma \in \Sigma$, we set $\tau(u) = \{\gamma\}$ and add u to V^∞. That is, we set $V^\infty = V^\infty \cup (R(v_\alpha, S))^{-1}$. Note that prior to this operation, $\gamma \in \tau(u)$ since otherwise u_γ would belong to T. We then pick an arbitrary vertex $x_\delta \in S$ and recursively solve the problem on 2 instances I_1 and I_2 defined as follows. The instance I_1 is defined to be $(G - \{x\}, k-1, \phi', \tau', V^\infty)$ where ϕ' and τ' are restrictions of ϕ and τ to $G - \{x\}$. The instance I_2 is defined to be $(G, k, \phi, \tau', V_1^\infty)$ where $V_1^\infty = V^\infty \cup \{x\}$ and τ' is defined to be the same as τ on all vertices but x and $\tau'(x) = \{\delta\}$. We call this branching rule, $\mathbf{B_1}$. The exhaustiveness of this branching step follows from the fact that once the vertices in $(R(v_\alpha, S)^{-1})$ are made undeletable, unless the vertex x is deleted, Observation 3.3 forces any feasible labeling that labels v with α to label x with δ.

Case 3: *The subroutine correctly concludes that there is no good v_α-T separator which is also smallest among all v_α-T separators disjoint from $[V^\infty]$.* In this case, we compute S, the minimum v_α-T separator that is disjoint from V^∞ and closest to v_α. Since S is not good, $R[v_\alpha, S]$ contains a pair of vertices u_{γ_1} and u_{γ_2} for some $u \in V(G)$ and $\gamma_1, \gamma_2 \in \Sigma$. Furthermore, since S is a v_α-T separator, it must be the case that u_{γ_1} and u_{γ_2} are not in T. This implies that $\{\gamma_1, \gamma_2\} \subseteq \tau(u)$. We now recursively solve the problem on 3 instances I_0, I_1, I_2 defined as follows. The instance I_0 is defined as $(G - \{u\}, k-1, \phi', \tau', w^*, V^\infty)$, where ϕ' and τ' are defined as the restrictions of ϕ and τ to the graph $G - \{u\}$. The instance I_1 is defined as $(G, k, \phi, \tau', w^*, V_1^\infty)$ where $V_1^\infty = V^\infty \cup \{u\}$ and τ' is defined to be be the same as τ on all vertices but u and $\tau'(u) = \tau(u) \setminus \{\gamma_1\}$. Similarly, the instance I_2 is defined as $(G, k, \phi, \tau', w^*, V_1^\infty)$ where $V_1^\infty = V^\infty \cup \{u\}$ and τ' is defined to be be the same as τ on all vertices but u and $\tau'(u) = \tau(u) \setminus \{\gamma_2\}$. We call this branching rule $\mathbf{B_2}$.

The exhaustiveness of this branching follows from the fact that if u is not deleted (the first branch) then any feasible labeling of $G - X$ for a hypothetical solution X must label u with at most one label out of γ_1 and γ_2. Therefore, if I is a YES instance then for at least one of the 2 instances I_1 or I_2, there is a feasible labeling of $G - X$ consistent with the corresponding τ'.

Case 4: Finally, we address the case when the subroutine returns a pair of minimum (among those disjoint from $[V^\infty]$) v_α-T separators S_1 and S_2 such that S_2 covers S_1, S_1 is good, S_2 is bad and there is no minimum (among those disjoint from V^∞) v_α-T separator which covers S_1 and is covered by S_2. In this case, $R[v_\alpha, S_2]$ contains a pair of vertices u_γ, u_δ for some vertex $u \in V(G)$ and $\gamma, \delta \in \Sigma$.

We execute the algorithm of Lemma 3.16 to compute in time $\mathcal{O}(|\Sigma| \cdot k(m+n))$, a v_α-u_γ path P_1 and a v_α- u_δ path P_2 such that both paths are internally vertex disjoint from S_2 and contain at most one vertex of S_1 each. Let $x^1, x^2 \in V(G)$ and $\beta_1, \beta_2 \in \Sigma$ be such that $x^1_{\beta_1}$ and $x^2_{\beta_2}$ are the vertices of S_1 in P_1 and P_2 respectively. Note that P_1 or P_2 may be disjoint from S_1. If P_i ($i \in \{1, 2\}$) is disjoint from S_1 then we let $x^i_{\beta_i}$ be undefined. We now recurse on the following (at most) 5 instances $I_1, \ldots, I_5$ defined as follows.

- $I_1 = (G - x^1, k-1, \phi', \tau', w^*, V^\infty)$ where ϕ' and τ' are restrictions of ϕ and τ to $G - \{x^1\}$.
- $I_2 = (G - x^2, k-1, \phi', \tau', w^*, V^\infty)$ where ϕ' and τ' are restrictions of ϕ and τ to $G - \{x^2\}$.
- $I_3 = (G - u, k-1, \phi', \tau', w^*, V^\infty)$ where ϕ' and τ' are restrictions of ϕ and τ to $G - \{u\}$.
- $I_4 = (G, k, \phi, \tau', w^*, V_1^\infty)$ where $V_1^\infty = V^\infty \cup (R(v_\alpha, S_1))^{-1} \cup \{x^1\}$ and τ' is the same as τ on all vertices of G except u and $\tau'(u) = \tau(u) \setminus \{\gamma\}$.
- $I_5 = (G, k, \phi, \tau', w^*, V_1^\infty)$ where $V_1^\infty = V^\infty \cup (R(v_\alpha, S_1))^{-1} \cup \{x^2\}$ and τ' is the same as τ on all vertices of G except u and $\tau'(u) = \tau(u) \setminus \{\delta\}$.

This branching rule is called $\mathbf{B_3}$ and we now argue the exhaustiveness of the branching. The first three branches cover the case when the solution intersects the set $\{x^1, x^2, u\}$. Suppose that a hypothetical solution, say X, is disjoint from $\{x^1, x^2, u\}$. By Lemma 3.12, we may assume that X is disjoint from $R(v_\alpha, S_1)$. Since any feasible labeling of $G - X$ sets u to at most one of $\{\gamma_1, \gamma_2\}$, branching into 2 cases by excluding γ_1 from $\tau(u)$ in the first case and excluding γ_2 from $\tau(u)$ in the second case gives us an exhaustive branching. This completes the description of the algorithm. The correctness follows from the exhaustiveness of the branchings. We will now prove the running time bound.

Analysis of running time. It follows from the description of the algorithm and the bounds already proved on the running time of each subroutine, that each step can be performed in time $\mathcal{O}((\Sigma + k)^{\mathcal{O}(1)}(m+n))$. Therefore, we only focus on bounding the number of nodes in

the search tree resulting from this branching algorithm. In order to analyse this number, we introduce the following measure for the instance $I = (G, k, \phi, \tau, w^*, V^\infty)$ corresponding to any node of the search tree. We define $\mu(I) = (\Sigma + 1)k - \lambda(I)$ where $\lambda(I)$ is $\lambda(w^*, T(I))$ if w^* is defined and 0 otherwise.

Note that $\lambda(w^*, T(I))$ denotes the size of the smallest w^*-$T(I)$ separator in H_I among those disjoint from $[V^\infty]$. Furthermore, observe that $\mu(I) \leq (|\Sigma| + 1) \cdot k$ for any instance on which the algorithm can potentially branch. We now argue that this measure strictly decreases in each branch of every branching rule and since the number of branches in any branching rule is bounded by $max\ \{|\Sigma| + 1, 5\}$ (Rules $\mathbf{B_0}$ and $\mathbf{B_3}$), the time bound claimed in the statement of Theorem 1.1 follows.

References

1. Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 226–234, 1993.
2. Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. 25(6):1305–1317, 1996.
3. Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. An $O(c^k n)$ 5-approximation algorithm for treewidth. In *FOCS*, pages 499–508, 2013. doi:10.1109/FOCS.2013.60.
4. Jianer Chen, Yang Liu, and Songjian Lu. An improved parameterized algorithm for the minimum node multiway cut problem. *Algorithmica*, 55(1):1–13, 2009. doi:10.1007/s00453-007-9130-6.
5. Rajesh Hemant Chitnis, Marek Cygan, MohammadTaghi Hajiaghayi, Marcin Pilipczuk, and Michal Pilipczuk. Designing FPT algorithms for cut problems using randomized contractions. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 460–469, 2012. doi:10.1109/FOCS.2012.29.
6. Marek Cygan, Marcin Pilipczuk, Michal Pilipczuk, and Jakub Onufry Wojtaszczyk. On multiway cut parameterized above lower bounds. *TOCT*, 5(1):3, 2013. doi:10.1145/2462896.2462899.
7. Frederic Dorn. Planar subgraph isomorphism revisited. In *STACS*, pages 263–274, 2010.
8. Michael R. Fellows and Michael A. Langston. Nonconstructive tools for proving polynomial-time decidability. *J. ACM*, 35(3):727–739, 1988. doi:10.1145/44483.44491.
9. Fedor V. Fomin, Daniel Lokshtanov, Neeldhara Misra, M. S. Ramanujan, and Saket Saurabh. Solving d-sat via backdoors to small treewidth. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 630–641, 2015.
10. Fedor V. Fomin, Daniel Lokshtanov, Neeldhara Misra, and Saket Saurabh. Planar f-deletion: Approximation, kernelization and optimal FPT algorithms. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 470–479, 2012.
11. Martin Grohe. Computing crossing numbers in quadratic time. In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 231–236, 2001.
12. Martin Grohe. Computing crossing numbers in quadratic time. *J. Comput. Syst. Sci.*, 68(2):285–302, 2004.

13 Martin Grohe, Ken ichi Kawarabayashi, and Bruce A. Reed. A simple algorithm for the graph minor decomposition – logic meets structural graph theory. In *SODA*, pages 414–431, 2013. doi:10.1137/1.9781611973105.30.

14 Sylvain Guillemot. FPT algorithms for path-transversal and cycle-transversal problems. *Discrete Optimization*, 8(1):61–71, 2011.

15 Ken ichi Kawarabayashi. Planarity allowing few error vertices in linear time. In *FOCS*, pages 639–648, 2009. doi:10.1109/FOCS.2009.45.

16 Ken ichi Kawarabayashi, Yusuke Kobayashi, and Bruce A. Reed. The disjoint paths problem in quadratic time. *J. Comb. Theory, Ser. B*, 102(2):424–435, 2012. doi:10.1016/j.jctb.2011.07.004.

17 Ken ichi Kawarabayashi and Bojan Mohar. Graph and map isomorphism and all polyhedral embeddings in linear time. In *STOC*, pages 471–480, 2008. doi:10.1145/1374376.1374443.

18 Ken ichi Kawarabayashi, Bojan Mohar, and Bruce A. Reed. A simpler linear time algorithm for embedding graphs into an arbitrary surface and the genus of graphs of bounded tree-width. In *FOCS*, pages 771–780, 2008. doi:10.1109/FOCS.2008.53.

19 Ken ichi Kawarabayashi and Bruce A. Reed. Computing crossing number in linear time. In *STOC*, pages 382–390, 2007. doi:10.1145/1250790.1250848.

20 Ken ichi Kawarabayashi and Bruce A. Reed. A nearly linear time algorithm for the half integral parity disjoint paths packing problem. In *SODA*, pages 1183–1192, 2009. doi:10.1145/1496770.1496898.

21 Ken ichi Kawarabayashi and Bruce A. Reed. An (almost) linear time algorithm for odd cycles transversal. In *SODA*, pages 365–378, 2010.

22 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

23 Yoichi Iwata, Keigo Oka, and Yuichi Yoshida. Linear-time fpt algorithms via network flow. In *SODA*, pages 1749–1761, 2014. doi:10.1137/1.9781611973402.127.

24 Yoichi Iwata, Keigo Oka, and Yuichi Yoshida. Linear-time fpt algorithms via network flow. In *SODA*, pages 1749–1761, 2014. doi:10.1137/1.9781611973402.127.

25 Yoichi Iwata, Magnus Wahlström, and Yuichi Yoshida. Half-integrality, lp-branching, and FPT algorithms. *SIAM J. Comput.*, 45(4):1377–1411, 2016. doi:10.1137/140962838.

26 Bart M. P. Jansen, Daniel Lokshtanov, and Saket Saurabh. A near-optimal planarization algorithm. In *SODA*, pages 1802–1811, 2014. doi:10.1137/1.9781611973402.130.

27 Subhash Khot. On the power of unique 2-prover 1-round games. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 767–775, 2002.

28 Subhash Khot. On the unique games conjecture (invited survey). In *Proceedings of the 25th Annual IEEE Conference on Computational Complexity, CCC 2010, Cambridge, Massachusetts, June 9-12, 2010*, pages 99–121, 2010.

29 Daniel Lokshtanov, N. S. Narayanaswamy, Venkatesh Raman, M. S. Ramanujan, and Saket Saurabh. Faster parameterized algorithms using linear programming. *ACM Transactions on Algorithms*, 11(2):15:1–15:31, 2014. doi:10.1145/2566616.

30 Daniel Lokshtanov, M. S. Ramanujan, and Saket Saurabh. Linear time parameterized algorithms for subset feedback vertex set. In *Automata, Languages, and Programming – 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 935–946, 2015. doi:10.1007/978-3-662-47672-7_76.

31 Dániel Marx. Parameterized graph separation problems. *Theor. Comput. Sci.*, 351(3):394–406, 2006. doi:10.1016/j.tcs.2005.10.007.

32 M. S. Ramanujan. A faster parameterized algorithm for group feedback edge set. In *Graph-Theoretic Concepts in Computer Science – 42nd International Workshop, WG*

2016, Istanbul, Turkey, June 22-24, 2016, Revised Selected Papers, pages 269–281, 2016. doi:10.1007/978-3-662-53536-3_23.

33 M. S. Ramanujan and Saket Saurabh. Linear time parameterized algorithms via skew-symmetric multicuts. In *SODA*, pages 1739–1748, 2014. doi:10.1137/1.9781611973402.126.

34 B. Reed. Finding approximate separators and computing tree-width quickly. In *Proceedings of the 24th Annual ACM symposium on Theory of Computing (STOC'92)*, pages 221–228. ACM, 1992.

35 Neil Robertson and Paul D. Seymour. Graph minors .xiii. the disjoint paths problem. *J. Comb. Theory, Ser. B*, 63(1):65–110, 1995.

36 Neil Robertson and Paul D. Seymour. Graph minors. XVIII. tree-decompositions and well-quasi-ordering. *J. Comb. Theory, Ser. B*, 89(1):77–108, 2003.

37 Neil Robertson and Paul D. Seymour. Graph minors. XX. wagner's conjecture. *J. Comb. Theory, Ser. B*, 92(2):325–357, 2004.

38 Magnus Wahlström. Half-integrality, LP-branching and FPT algorithms. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1762–1781, 2014. doi:10.1137/1.9781611973402.128.

Dynamic Space Efficient Hashing

Tobias Maier[1] **and Peter Sanders**[2]

1 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
t.maier@kit.edu
2 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
sanders@kit.edu

Abstract

We consider space efficient hash tables that can grow and shrink dynamically and are always highly space efficient, i.e., their space consumption is always close to the lower bound even while growing and when taking into account storage that is only needed temporarily. None of the traditionally used hash tables have this property. We show how known approaches like linear probing and bucket cuckoo hashing can be adapted to this scenario by subdividing them into many subtables or using virtual memory overcommitting. However, these rather straightforward solutions suffer from slow amortized insertion times due to frequent reallocation in small increments.

Our main result is DySECT (**Dy**namic **S**pace **E**fficient **C**uckoo **T**able) which avoids these problems. DySECT consists of many subtables which grow by doubling their size. The resulting inhomogeneity in subtable sizes is equalized by the flexibility available in bucket cuckoo hashing where each element can go to several buckets each of which containing several cells. Experiments indicate that DySECT works well with load factors up to 98%. With up to 2.7 times better performance than the next best solution.

1998 ACM Subject Classification E.2 Data Storage Representations, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Dynamic data structures, open addressing, closed hashing, cuckoo hashing, space efficiency

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.58

1 Introduction

Dictionares represented as hash tables are among the most frequently used data structures and often play a critical role in achieving high performance. Having several compatible implementations, which perform well under different conditions and can be interchanged freely, allows programmers to easily adapt known solutions to new circumstances.

One aspect that has been subject to much investigation is space efficiency [3, 4, 7, 8, 17, 19]. Modern space efficient hash tables work well even when filled to 95% and more. To reach filling degrees like this, the table has to be initialized with the correct final capacity, thereby, requiring programmers to know tight bounds on the maximum number of inserted elements. This is typically not realistic. For example, a frequent application of hash tables aggregates information about data elements by their key. Whenever the exact number of unique keys is not known a priori, we have to overestimate the initial capacity to guarantee good performance. Dynamic space efficient data structures are necessary to guarantee both good performance and low overhead independent of the circumstances.

To visualize this, assume the following scenario. During a word count benchmark, we know an upper bound $n_{\max}$ to the number of unique words. Therefore, we construct a hash table with at least $n_{\max}$ cells. If an instance only contains $0.7 \cdot n_{max}$ unique words, no static hash table can fill ratios greater than 70%. Thus, dynamic space efficient hash tables are required to achieve guaranteed near-optimal memory usage. In scenarios where the final size is not known, the hash table has to grow closely with the actual number of elements. This cannot be achieved efficiently with any of the current techniques used for hashing and migration.

Many libraries – even ones that implement space efficient hash tables – offer some kind of growing mechanism. However, all existing implementations either lose their space efficiency or suffer from degraded performance once the table grows above its original capacity. Growing is commonly implemented either by creating additional hash tables – decreasing performance especially for lookups or by migrating all elements to a new table – losing the space efficiency by multiplying the original size.

To avoid the memory overhead of full table migrations, during which both the new and the old table coexist, we propose an in-place growing technique that can be adapted to most existing hashing schemes. However, frequent migrations with small relative size changes remain necessary to stay space efficient at all times.

To avoid both of these pitfalls we propose a variant of (multi-way) bucket cuckoo hashing [7, 8]. A technique where each element can be stored in one of several associated constant sized buckets. When all of them are full, we move an element into one of its other buckets to make space. To solve the problem of efficient migration, we split the table into multiple subtables, each of which can grow independently of all others. Because the buckets associated with one element are spread over the different subtables, growing one subtable alleviates pressure from all others by allowing moves from a dense subtable to the newly-grown subtable.

Doubling the size of one subtable increases the overall size only by a small factor while moving only a small number of elements. This makes the size changes easy to amortize. The size and occupancy imbalance between subtables (introduced by one subtable growing) is alleviated using displacement techniques common to cuckoo hashing. This allows our table to work efficiently at fill rates exceeding 95%.

We begin our paper by presenting some previous work (Section 2). Then we go into some notations (Section 3) that are necessary to describe our main contribution DySECT (Section 4). In Section 5 we show our in-place migration techniques. Afterwards, we test all Hashtables on multiple benchmarks (Section 6) and draw our conclusion (Section 7)

2 Related Work

The use of hash tables and other hashing based algorithms has a long history in computer science. The classical methods and results are described in all major algorithm textbooks [14].

Over the last one and a half decades, the field has regained attention, both from theoretical and the practical point of view. The initial innovation that sparked this attention was the idea that storing an element in the less filled of two "random" chains leads to incredibly well balanced loads. This concept is called the power of two choices [16].

It led to the development of cuckoo hashing [19]. Cuckoo hashing extends the power of two choices by allowing to move elements within the table to create space for new elements (see Section 3.2 for a more elaborated explanation). Cuckoo hashing revitalized research into space efficient hash tables. Probabilistic bounds for the maximum fill degree [3, 4] and expected displacement distances [9, 10] are often highly non-trivial.

Cuckoo hashing can be naturally generalized into two directions in order to make it more space efficient: allowing H choices [8] or extending cells in the table to *buckets* that can store B elements. We will summarize this under the term *bucket cuckoo hashing*.

Further adaptations of cuckoo hashing include:multiple concurrent implementations either powered by bucket locking, transactional memory [15], or fully lock-less [18]; a de-amortization technique that provides provable worst case guarantees for insertions [1, 13]; and a variant that minimizes page-loads in a paged memory scenario [6].

Some non-cuckoo space efficient hash tables continue to use linear probing variants. *Robin Hood hashing* is a technique that was originally introduced in 1985 [2]. The idea behind Robin Hood hashing is to move already stored elements during insertions in a way that minimizes the longest possible search distance. Robin Hood hashing has regained some popularity in recent years, mainly for its interesting theoretical properties and the possibility to reduce the inherent variance of linear probing.

All these publications show that there is a clear interest in developing hash tables that can be more and more densely filled. Dynamic hash tables on the other hand seem to be considered a solved problem. One paper that takes on the problem of dynamic hash tables was written by Dietzfelbinger at al. [5]. It predates cuckoo hashing, and much of the attention for space efficient hashing. All memory bounds presented are given without tight constant factors. The lack of implementations and theory about dense dynamic hash tables is where we pick up and offer a fast hash table implementation that supports dynamic growing with tight space bounds.

3 Preliminaries

A hash table is a data structure for storing key-value-pairs ($\langle key, data\rangle$) that offers the following functionality: `insert` – stores a given key-value pair or returns a reference to it, if it is already contained; `find` – given a key returns a reference to said element if it was stored, and $\perp$ otherwise; and `erase` – removes a previously inserted element (if present).

Throughout this paper n denotes the number of elements and m the number of cells ($m > n$) in a hash table. We define the load factor as $\delta = n/m$. Tables can usually only operate efficiently up to a certain maximum load factor. Above that, operations get slower or have a possibility to fail. When implementing a hash table one has to decide between storing elements directly in the hash table – *Closed Hashing* – or storing pointers to elements – *Open Hashing*. This has an immediate impact on the amount of memory required (*closed*: $m \cdot |element|$ and *open*: $m \cdot |pointer| + n \cdot |element|$).

For large elements (i.e., much larger then the size of a pointer), one can use a non-space efficient hash table with open hashing to reduce the relevant memory factor. Therefore, we restrict ourselves to the common and more interesting case of elements whose size is close to that of a pointer. For our experiments we use 128bit elements (64bit keys and 64bit values). In this case, open hashing introduces a significant memory overhead (at least 1.5×). For this reason, we only consider closed hash tables. Their memory efficiency is directly dependent on the table's load. To reach high fill degrees with closed hashing tables, we have to employ *open addressing* techniques. This means that elements are not stored in predetermined cells, but can be stored in one of several possible places (e.g. linear probing, or cuckoo hashing).

3.1 α–Space Efficient Hash Tables

Static. We call a hashing technique α-space efficient when it can work efficiently using at most $\alpha \cdot n_{curr} \cdot size(element) + O(1)$ memory. We define working efficiently, for a table with

load factor δ, as having average insertion times in $O(\frac{1}{1-\delta})$. This is a natural estimation for insertion times, since it is the expected number of fully random probes needed to hit an empty cell.

In many closed hashing techniques (e.g. linear probing, cuckoo hashing) cells are the same size as elements. Therefore, being α-space efficient is the same as operating with a load factor of $\delta = \alpha^{-1}$. Because of this, we will mostly talk about the load factor of a table instead of its memory usage.

Dynamic. The definition of a space efficient hashing technique given above is specifically targeted for statically sized hash tables. We call an implementation *dynamically α-space efficient* if an instantiated table can grow arbitrarily large over its original capacity while remaining smaller than $\alpha \cdot n_{\max} \cdot size(element) + O(1)$ at all times.

One problem for many implementations of space efficient hash tables is the migration. During a normal full table migration, both the original table and the new table are allocated. This requires $m_{new} + m_{old}$ cells. Therefore, a normal full table migration is never more than 2-space efficient. The only option for performing a full table migration with less memory is to increase the memory in-place (see Section 5). Similar to static α-space efficiency, we will mostly talk about the *minimum load factor* $\delta_{\min} = \frac{1}{\alpha}$ instead of α.

3.2 Cuckoo Hashing

Cuckoo hashing [7, 8, 17, 19] is a technique to resolve hash conflicts in a hash table using open addressing. Its main draw is that it guarantees constant lookup times even in densely filled tables. The distinguishing technique of cuckoo hashing is that H hash functions $(h_1, ..., h_H)$ are used to compute H independent positions. Each element is stored in one of its positions. Even if all positions are occupied one can often move elements to create space for the current element. We call this process *displacing* elements.

Bucket cuckoo hashing is a variant where the cells of the hash table are grouped into buckets of size B. Each element assigned to one bucket can be stored in any of the bucket's cells. Using buckets one can drastically increase the number of elements that can be displaced to make room for a new one, thus decreasing the expected length of displacement paths.

Find and *erase* operations have a guaranteed constant running time. Independent from the table's density, there are H buckets – $H \cdot B$ cells – we have to check to find an element.

During an *insert* the element is hashed to H buckets. We store the element in the bucket with the most free space. When all buckets are full we have to move elements within the table such that a free cell becomes available.

To visualize the problem of displacing elements, one can think of the directed graph implicitly defined by the hash table. Each bucket corresponds to a node and each element induces an edge between the bucket it is stored in and its $H - 1$ alternate buckets. To insert an element into the hash table we have to find a path from one of its associated buckets to a bucket that has free capacity. Then we move elements along this path to make room in the initial bucket. The two common techniques to find such paths are *random walks* and *breadth first searches*.

4 DySECT (Dynamic Space Efficient Cuckoo Table)

A commonly used growing technique is to double the size of a hash table by migrating all its elements into a table with twice its capacity. This is of course not memory efficient. The idea behind our dynamic hashing scheme is to double only parts of the overall data structure.

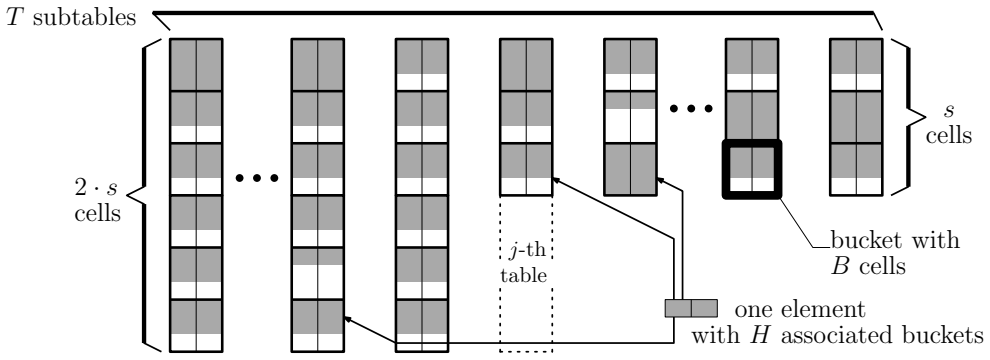


Figure 1 Schematic Representation of a DySECT Table.

This increases the space in part of our data structure without changing the rest. We then use cuckoo displacement techniques to make this additional memory reachable from other parts of the hash table.

4.1 Overview

Our DySECT hash table consists of T subtables (shown in Figure 1) that in turn consist of buckets, wich can store B elements each. Each element has H associated buckets – similar to cuckoo hashing – which can be in the same or in different subtables. T, B, and H are constants, which will not change during the lifetime of the table. Additionally, each table is initialized with a minimum fill ratio $\delta_{\min}$. The table will never exceed $\delta_{\min}^{-1} \cdot n$ cells once it begins to grow over its initial size.

To find a bucket associated with an element e, we compute e's hash value using the appropriate hash function $h_i(e)$. The hash is then used to compute the subtable and the bucket within that subtable. To make this efficient we use powers of two for the number of subtables ($T = 2^t$), as well as for the number of buckets per subtable (subtable size $s = 2^x \cdot B$). Since the number of subtables is constant, we can use the first t bits from the hashed key to find the appropriate subtable. From the remaining bits we compute the bucket within that subtable using a bitmask ($h_i(e) \,\&\, (2^x - 1) = h_i(e) \mod 2^x$).

4.2 Growing

As soon as the (overall) table contains enough elements such that the memory constraint can be kept during a subtable migration, we grow one subtable by migrating it into a table twice its size. We migrate subtables in order from first to last. This ensures that no subtable can be more than twice as large as any other.

When the data structure contains j large subtables ($2s$) then there are $m = (T + j) \cdot s$ cells. When $\delta_{\min}^{-1} \cdot n > m + 2s$ we can grow the first subtable while obeying the size constraint (the newly allocated table will have $2s$ cells). Doubling the size of a subtable increases the global number of cells from $m_{old} = (T + j) \cdot s$ to $m_{new} = m_{old} + s = (T + j + 1) \cdot s$ (grow factor $\frac{T+j+1}{T+j}$). Note that all subsequent growing operations migrate one of the smaller tables until all tables have the same size. Therefore, each grow until then increases the overall capacity by the same absolute amount (smaller relative to the current size).

The cost of growing a subtable is amortized by all insertions since the last subtable migration. There are $\delta_{\min} \cdot s = \Omega(s)$ insertions between two migrations. One migration takes $\Theta(s)$ time. Apart from being amortized, the migration is cache efficient since it accesses cells

in a linear fashion. Even in the target table cells are accessed linearly. We assign elements to buckets by using bits from their hash value. In the grown table we use exactly one more bit than before (double the number of buckets). This ensures that all elements from one original bucket are split between two buckets in the target table. Therefore no bucket can overflow and no displacements are necessary.

In the implicit graph model of the cuckoo table (Section 3.2), growing a subtable is equivalent to splitting each node that represents a bucket within that subtable. The resulting graph becomes more sparse, since the edges (elements) are not doubled, making it easier to insert subsequent elements.

4.3 Shrinking

If shrinking is necessary it can work similarly to growing. We replace a subtable with a smaller one by migrating elements from one to the other. During this migration we join elements from two buckets into one. Therefore it is possible for a bucket to overfill. We reinsert these elements at the end of the migration. Obviously, this can only affect at most half the migrated elements.

When automatically triggering the size reduction, one has to make sure that the migration cost is amortized. Therefore, a grow operation cannot immediately follow a shrink operation. When shrinking is enabled we propose to shrink one subtable when $\delta_{\min}^{-1} \cdot n < m - s'$ elements (s' size of a large table, $m_{new} = m_{old} - s'/2$). Alternatively, one could implement a *shrink to size* operation that is explicitly called by the user.

4.4 Difficulties for the Analysis of DySECT

There are two factors specific to DySECT impacting its performance: *inhomogeneous table resolution* and *element imbalance*.

Imbalance through Inhomogeneous Table Resolution. By growing subtables individually we introduce a size imbalance between subtables. Large subtables contain more buckets but the number of elements hashed to a large subtable is not generally higher than the number of elements that are hashed to a small subtable. This makes it difficult to spread elements evenly among buckets. Imbalanced bucket fill ratios can lead to longer insertion times.

Assume there are n elements in a hash table with T subtables, j of which have size $2s$ the others have size s. If elements are spread equally among buckets then all small tables have around $n/(T + j)$ elements, and the bigger tables have $2n/(T + j)$ elements. For each table there are about Hn/T elements that have an associated bucket within that table. This shows that having more hash functions can lead to a better balance.

For two hash functions ($H = 2$) and only one grown table ($j = 1$) this means that $\approx 2n/(T + 1)$ elements should be stored in the first table to achieve a balanced bucket distribution. Therefore, nearly all elements associated with a bucket in the first table ($\approx 2n/T$) have to be stored there. This is one reason why $H = 2$ does not work well in practice.

Imbalance through Size Changes. In addition to the problem of inhomogeneous tables there is an inherent balancing problem introduced by resizing subtables. It is clear that a newly grown table is not filled as densely as other tables. Since we double the table size, grown tables can only be filled to about 50%.

Assume the global table is filled close to 100% when the first table grows. Now there is capacity for s new elements but this capacity is only in the first table, elements that are not hashed to the first table, automatically trigger displacements leading to slow insertions. Notice that repeated `insert` and `erase` operations help to equalize this imbalance, because elements are more likely inserted into the sparser areas, and more likely to be deleted from denser areas.

4.5 Implementation Details

For our experiments (Section 6) we use three hash functions ($H = 3$) and a bucket size of ($B = 8$). These values have consistently outperformed other options both in maximum load factor and in `insert` performance. T is set to 256 subtables for all our tests. To find displacement opportunities we use breadth first search. In our tests it performed better than random walks, since it better uses the read cache lines from one bucket.

The hash table itself is implemented as a constant sized array of pointers to subtables. We have to lookup the corresponding pointer whenever a subtable is accessed. This does not impact performance much since all subtable pointers will be cached – at least if the hash table is a performance bottleneck.

Reducing the Number of Computed Hash Functions. Evaluating hash functions is expensive, therefore, reducing the number of hash functions computed per operation can increase the performance of the table. The hash function we use computes 64bit hash values (i.e. xxHash[1]). We split the 64bit hash value into two 32bit values. All common bucket hash table sizes can be addressed using 32 bits (up to 2^{32} buckets $2^{35} \approx 34$ billion elements consuming 512GiB memory).

When $H > 2$ we can use *double hashing* [11, 12] to further reduce the number of computed hash functions. Double hashing creates an arbitrary number of hash values using only two original hash functions h' and h''. The additional values are linear combinations computed from the original two values, $h_i(key) = h'(key) + i \cdot h''(key)$.

Combining both of these techniques, we can reduce the number of computed hash functions to one 64bit hash function. This is especially important during large displacements where each encountered element has to be rehashed to find its alternative buckets.

5 (Ab)Using Virtual Memory

In this section we show how one can use virtual memory and memory overcommitting, to eliminate the indirections from a DySECT hash table. The same technique also allows us to implement hash tables that can grow using an in-place full table migration. If we grow these tables in small increments, they can grow while enforcing a strict size constraint.

To explain these techniques, we first have to explain how to use memory overcommitting and virtual memory to create a piece of memory that can grow in-place. Note that this technique violates best programming practices and is not fully portable to some systems.

The idea is the following: the operating system will – if configured to do so – allow memory allocations larger than the machine's main memory, with the anticipation that not all allocated memory will actually be used. Only memory pages that are actually used will be mapped from virtual to physical memory pages. Thus, for the purpose of space efficiency the memory is not yet used. Initializing parts of this memory is similar to allocating and initializing new memory.

[1] `http://xxhash.com`

5.1 Improving DySECT

Accessing a DySECT subtable usually takes one indirection. The pointer to the subtable has to be read from an array of pointers before accessing the actual subtable. Instead of using an array of pointers, we can implement the subtables as sections within one large allocation (size u). We choose u larger than the actual main memory, to allow all possible table sizes. This has the advantage that the offset for each table can be computed quickly ($t_i = \frac{u}{T} \cdot i$), without looking it up from a table.

The added advantage is that we can grow subtables in-place. To increase the size of a subtable, it is enough to initialize a consecutive section of the table (following the original subtable). Once this is done, we have to redistribute the table's elements. This allows us to grow a subtable without the space overhead of reallocation. Therefore, we can grow earlier, staying closer to the minimum load factor $\delta_{\min}$. The in-place growing mechanism is easy in this case, since the subtable size is doubled.

5.2 Implementing other size constrained tables

Similarly to the technique above, we can implement any hash table using a large allocation, initializing only as much memory as the table initially needs. The used hash table size can be increased in-place by initializing more memory. To use this additional memory for the hash table, we have to perform an in-place migration.

To implement fast in-place migration, we need the correct addressing technique. There are two natural ways to map a hash value $h(e)$ to a cell in the table (size s). Most programmers would use a slow modulo operation ($h(e) \mod s$). This is the same as using the least significant digits when addressing a table whose size is a power of two. A better way is to use a scale factor ($\lfloor h(e) \cdot \frac{s}{max(h)} \rfloor$). This is similar to using the most significant bits to address a table whose size is a power of two. The second method has two important advantages, it is faster to compute and it helps to make the migration cache efficient. When we use the second method the elements in the hash table are close to being sorted by their hash value (in the absence of collisions they would be sorted).

The main idea of all our in-place migration techniques is the following. If we use a scale factor for our mapping – in the new table – most elements will be mapped to a position that is larger than their position in the old table. Therefore, rehashing elements starting from the back of the original table creates very few conflicts. Elements that are mapped to a position earlier than their current position are buffered and reinserted at the end of the migration. When using this technique, both the old and the new table, are accessed linearly in reverse order (from back to front). Making the migration cache efficient and easy to implement.

For a table that was initialized with a min load factor $\delta_{\min}$ we trigger growing once the table is loaded more than $\frac{\delta_{\min}+1}{2}$. We then increase the capacity m to $\delta_{\min}^{-1} \cdot n$. Repeated migrations with small growing amounts are still inefficient, since each element has to be moved.

This blueprint can be used, to implement in-place growing variants of most if not all common hashing techniques. We used these same ideas to implement variants of *linear probing*, *robin hood hashing*, and *bucket cuckoo hashing*. Although some variants have their own optimized migration. Robin Hood hashing can be adapted such that the table is truly sorted by hash value (without much overhead) making the migration faster than repeated reinsertions. Bucket cuckoo hashing has a somewhat more complicated migration technique – each element has multiple possible positions and buckets can overflow. Here we first try to reinsert each element with the hash function $(h_1, ..., h_H)$ that was previously used to store it.

6 Experiments

There are many factors that impact hash table performance. To show that our ideas work in practice we use both micro-benchmarks and practical experiments.

All reported numbers are averaged by running each experiment five times. The experiments were executed on a server with two Intel Xeon E5-2670 CPUs (2.3GHz base frequency) and 128GB RAM (using gcc 6.2.0 and Ubuntu 14.04).[2]

To put the performance of our DySECT table into perspective, we implement and test several other options for space efficient hashing using the method described in Section 5.2. We use our own implementations, since no hash table found online supports our strict space-efficiency constraint. With the technique described in Section 5.2, we implement and test hash tables with *linear probing*, *robin hood hashing*, and *bucket cuckoo hashing* (similar to DySECT we choose $B = 8$ and $H = 3$). For each table, we implemented an individually tuned cache efficient in-place migration algorithm.

Alternative Implementation Without Virtual Overcommitting (dashed lines). All implementations described above work with the trick described in Section 5. The usefulness of this technique is arguable, since abusing the concept of virtual memory in this way is bad from a software design perspective. It violates best practices, and reduces the portability to many systems. Therefore, we want to present some additional measurements that do not use this technique. These measurements are presented using dashed lines.

The only table that can achieve dynamic α-space efficiency without virtual memory overcommitting is our DySECT hash table. It is notable that the variant without overcommitting is never significantly worse than DySECT with overcommitting.

For each competitor table we implement a variant that uses subtables combined with subtable migrations. Elements are first hashed to subtables and then hashed within that subtable. They cannot move between subtables. Even when growing by small steps these versions can violate their space efficiency. But since subtables are small ($\approx n/T$), migrations will usually not violate the size constraint significantly. There can be larger subtables, since imbalances between subtables cannot be regulated.

6.1 Influence of Fill Ratio (Static Table Size)

The following test was performed by initializing a table with $\approx 25\,000\,000$ cells (non-growing). Then elements are inserted until there is a failing insertion. At different stages, we measure the running time of new insertion (Figure 2), and find (Figure 3) operations (averaged over 1000 operations). Finds are measured using either randomly selected elements from within the table (successful), or by searching random elements from the whole key space (unsuccessful). We omit testing multi table variants of the competitor tables. They are not suitable for this test since forcing a static size limits the possibility to react to size imbalances between subtables (in the absence of displacements).

As to be expected, the insertion performance of depends highly on the fill degree of the table. Therefore, we show it normalized with $\frac{1}{1-\delta}$ which is the expected number of fully random probes to find a free cell and thus a natural estimate for the running time. We see that – up to a certain point – the insertion time behaves proportional to $\frac{1}{1-\delta}$ for all tables.

[2] Experiments on a desktop machine yielded similar results.

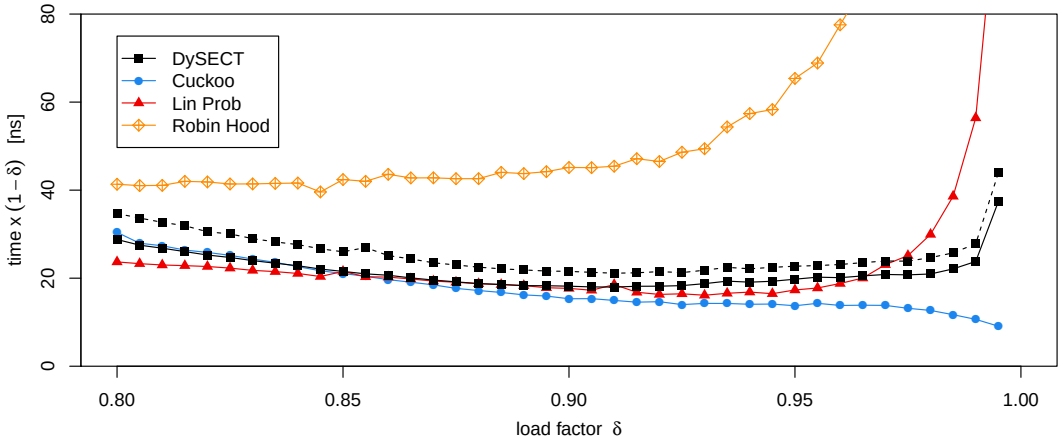


Figure 2 Insertions into a Static Table. Here we show the influence from the load factor, on the performance of insertions. To make insertion time more readable, we normalize it with $t_{op} \cdot (1 - \delta)$.

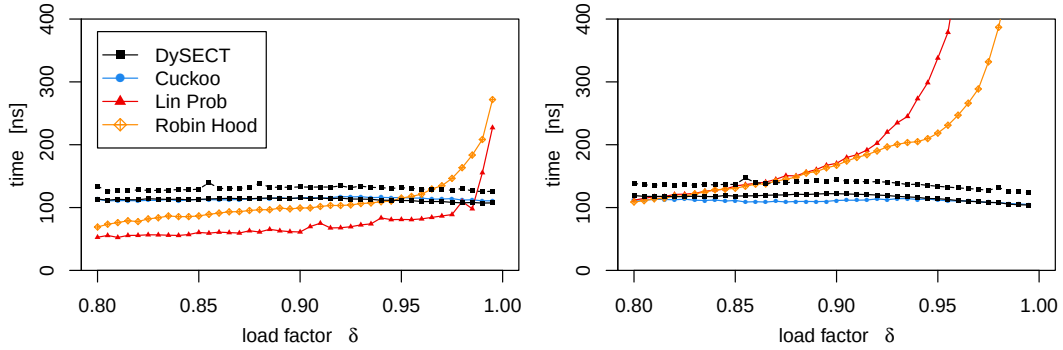


Figure 3 Performance of Successful (*left*) and Unsuccessful (*right*) Finds. DySECT's find performance is independent from the load factor, and the operations success.

Close to the capacity limit of the table, the insertion time increases sharply. DySect has a smaller capacity limit than cuckoo due inhomogeneous table resolution (see Section 4.4).

Figure 3 shows the performance of find operations. Linear probing performs relatively well on successful find operations, up to a fill degree of over 95%. The reason for this is that many elements were inserted into the table when the table was still relatively empty. They have very short search distances, thus improving find performance. Successful find performance can still be an issue in applications. An element that is inserted when the table is already decently filled can have an extremely long search distance. This leads to a high running time variance on find operations. Unsuccessful finds perform really badly, since all cells until the next free cell have to be probed. Their performance is much more related to the filling degree of the hash table. Robin Hood hashing performs somewhat similar to linear probing. It worsens the successful find performance by moving previously inserted elements from their original position, in order to achieve better unsuccessful find performance on highly filled tables. Overall, Robin Hood hashing is objectively worse than both DySECT and classic cuckoo hashing. Cuckoo hashing and its variants like DySECT have guaranteed constant running times for all find operations – independent of their success and the table's filling degree.

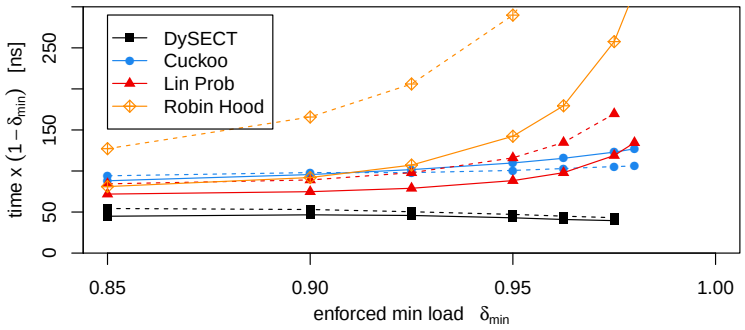


Figure 4 Insertions into a dynamic growing table enforcing a minimum load factor $\delta_{\min}$.

6.2 Influence of Fill Ratio (Dynamic Table Size)

In this test 20 000 000 elements are inserted into an initially empty table. The table is initialized expecting 50 000 elements, thus growing is necessary to fit all elements. The tables are configured to guarantee a load factor of at least $\delta_{\min}$ at all times. Figure 4 shows the performance in relation to the load factor. Insertion times are computed as average of all 20 000 000 insertions. They are normalized similar to Figure 2 (divided by $\frac{1}{1-\delta_{\min}}$).

We see that DySECT performs by far the best even with load factors around 85%. Here we achieve a speedup of 1.6 over the next best solution (299ns vs. linear probing 479ns). On denser instances with 97.5% load, we can increases thes speedup to 2.7 (1580ns vs Cuckoo with subtables 4210ns). With growing load, we see the insertion times of our competitors degrade due to the combination of long insertion times, and frequent growing phases. There are only few insertions between two growing phases. Making the amortization of each growing phase challenging since each growing phase has to move all elements. Any growing technique that uses a less cache efficient migration algorithm, would likely perform significantly worse. DySECT however remains close to $O(\frac{1}{1-\delta_{\min}})$ even for fill degrees up to 97.5%. This is possible, because only very few elements are touched by each subtable migration ($\approx \frac{n}{T}$).

We also measured the performance of find operations on the created tables, they are similar to the performance on the static table in Section 6.1 (see Figure 3), therefore, we omit displaying them for space reasons.

6.3 Word Count – a Practical use Case

Word count and other aggregation algorithms are some of the most common use cases for hash tables. Data is aggregated according to its key. This is a common application, in which static hash tables can never be space efficient, since the final size of the hash table is usually unknown. Here we use the first block of the *CommonCrawl* dataset (`http://commoncrawl.org/the-data/get-started`) and compute a word count of the contained words. The chosen block has 4.2GB and contains around 240 000 000 words, with around 20 000 000 unique words. For the test, we hash each word to a 64 bit key and insert it together with a counter. Subsequent accesses to the same key increase this counter. Similar to the growing benchmark, we start with an empty table initialized for 50 000 elements.

The performance results can be seen in Figure 5 (*left*). We do not use any normalization since each word is repeated 12 times (on average). This means that most operations will actually behave more like successful find operations instead of insertions. When using our DySECT table, the running time seems to be nearly independent from the fill degree. We experience little to no slowdown until around 97%. The tables using full table migration

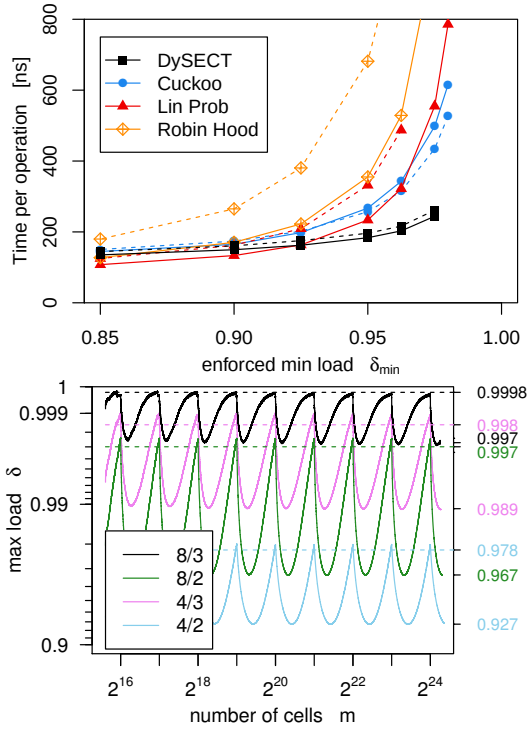


Figure 5 (*left*) Word Count Benchmark. Behaves like a mix of insert and find operations. DySECT's performance is nearly independent form the load factor. (*right*) Experimental Max Load Bounds. Dependent on the number of cells (different B/H parametrizations).

however become very inefficient on high load degrees. For high load factors, the performance closely ressambles that of the insertion benchmark (Figure 4). This indicates that `inserts` can dominate performance even in `find` intensive workloads.

6.4 Experimental Maximum Load Bounds

After some investigation, we are confident that our approach can be analyzed using methods similar to those used for statically sized tables. Until then we can offer some bounds we found experimentally. We used $T = 4096$ subtables (smaller grow steps) and different values of B and H. To fill the table as much as possible we configured the table, to only grow once an insertion fails (16384 probing distance). Figure 5 (*right*) shows the load bound after each subtable migration. It indicates that the maximum load degree depends on the number of large subtables. This creates the periodic nature of the plot with maximums whenever the table has a size close to a power of two. There the table reaches the performance of a classic cuckoo hash table (displayed as dashed lines).

7 Conclusion

We have shown that dynamically growing hash tables can be implemented to always consume space close to the lower bound. We find it surprising that even our simple solutions based on linear probing seem to be new. DySECT is a sophisticated solution that exploits the flexibility offered by bucket cuckoo hashing to significantly decrease the number of object

migrations over more straightforward approaches. When very high space efficiency is desired, it is up to 2.7 times better than simple solutions.

For future work, a theoretical analysis of DySECT looks interesting. After some discussion, we expect that techniques previously used to analyze bucket cuckoo hashing will be applicable to the new situations. But the calculations have to take into account all possible ratios of small versus large subtables. Even for the static case and classical bucket cuckoo hashing, it is a fascinating open question wether the observed proportionality of insertion time to $1/(1-\delta)$ can be proven. Previous results on insertion time show much more conservative bounds [8, 9, 10, 7]. On the practical side, DySECT looks interesting for concurrent hashing [15, 18] since it grows only small parts of the table at a time.

References

1. Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *International Conference on Automata, Languages and Programming (ICALP)*, number 5555 in LNCS, pages 107–118. Springer, 2009.
2. Pedro Celis, Per-Ake Larson, and J. Ian Munro. Robin hood hashing. In *26th Symposium on Foundations of Computer Science (FOCS)*, pages 281–288, Oct 1985.
3. Luc Devroye and Pat Morin. Cuckoo hashing: Further analysis. *Information Processing Letters*, 86(4):215–219, 2003.
4. Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In *27th International Conference on Automata, Languages and Programming (ICALP)*, pages 213–225, 2010.
5. Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.
6. Martin Dietzfelbinger, Michael Mitzenmacher, and Michael Rink. Cuckoo hashing with pages. In *19th European Symposium on Algorithms (ESA)*, number 6942 in LNCS, pages 615–627. Springer, 2011.
7. Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1):47–68, 2007.
8. Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, 2005.
9. Nikolaos Fountoulakis, Konstantinos Panagiotou, and Angelika Steger. On the insertion time of cuckoo hashing. *SIAM Journal on Computing*, 42(6):2156–2181, 2013.
10. Alan Frieze, Páll Melsted, and Michael Mitzenmacher. An analysis of random-walk cuckoo hashing. *SIAM Journal on Computing*, 40(2):291–308, 2011.
11. Leo J. Guibas and Endre Szemeredi. The analysis of double hashing. *Journal of Computer and System Sciences*, 16(2):226–274, 1978.
12. Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In Yossi Azar and Thomas Erlebach, editors, *14th European Symposium on Algorithms (ESA)*, number 4168 in LNCS, pages 456–467. Springer, 2006.
13. Adam Kirsch and Michael Mitzenmacher. Using a queue to de-amortize cuckoo hashing in hardware. In *45th Annual Allerton Conference on Communication, Control, and Computing*, volume 75, 2007.
14. Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

15 Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *9th European Conference on Computer Systems*, EuroSys'14, pages 27:1–27:14. ACM, 2014.

16 M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, Oct 2001.

17 Michael Mitzenmacher. Some open questions related to cuckoo hashing. In Amos Fiat and Peter Sanders, editors, *17th European Symposium on Algorithms (ESA)*, volume 5757 of *LNCS*, pages 1–10. Springer, 2009.

18 N. Nguyen and P. Tsigas. Lock-free cuckoo hashing. In *2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*, pages 627–636, June 2014.

19 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

Subexponential Parameterized Algorithms for Graphs of Polynomial Growth*

Dániel Marx[1] and Marcin Pilipczuk[2]

1 **Institute for Computer Science and Control, Hungarian Academy of Sciences (MTA SZTAKI), Budapest, Hungary**
dmarx@cs.bme.hu

2 **Institute of Informatics, University of Warsaw, Warsaw, Poland**
marcin.pilipczuk@mimuw.edu.pl

Abstract

We show that for a number of parameterized problems for which only $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ time algorithms are known on general graphs, subexponential parameterized algorithms with running time $2^{\mathcal{O}(k^{1-\frac{1}{1+\delta}}\log^2 k)}n^{\mathcal{O}(1)}$ are possible for graphs of polynomial growth with growth rate (degree) δ, that is, if we assume that every ball of radius r contains only $\mathcal{O}(r^\delta)$ vertices. The algorithms use the technique of *low-treewidth pattern covering,* introduced by Fomin et al. [18] for planar graphs; here we show how this strategy can be made to work for graphs of polynomial growth.

Formally, we prove that, given a graph G of polynomial growth with growth rate δ and an integer k, one can in randomized polynomial time find a subset $A \subseteq V(G)$ such that on one hand the treewidth of $G[A]$ is $\mathcal{O}(k^{1-\frac{1}{1+\delta}}\log k)$, and on the other hand for every set $X \subseteq V(G)$ of size at most k, the probability that $X \subseteq A$ is $2^{-\mathcal{O}(k^{1-\frac{1}{1+\delta}}\log^2 k)}$. Together with standard dynamic programming techniques on graphs of bounded treewidth, this statement gives subexponential parameterized algorithms for a number of subgraph search problems, such as Long Path or Steiner Tree, in graphs of polynomial growth.

We complement the algorithm with an almost tight lower bound for Long Path: unless the Exponential Time Hypothesis fails, no parameterized algorithm with running time $2^{k^{1-\frac{1}{\delta}-\varepsilon}}n^{\mathcal{O}(1)}$ is possible for any $\varepsilon > 0$ and any integer $\delta \geq 3$.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases polynomial growth, subexponential algorithm, low treewidth pattern covering

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.59

1 Introduction

In recent years, research on parameterized algorithms had a strong focus on understanding the optimal form of dependence on the parameter k in the running time $f(k)n^{\mathcal{O}(1)}$ of parameterized algorithms. For many of the classic algorithmic problems on graphs, algorithms with running time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ exist, and we know that this form of running time is best

* The research of D. Marx leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 280152. The research of M. Pilipczuk is supported by Polish National Science Centre grant UMO-2013/09/B/ST6/03136. Part of the research has been done when the authors were participating in the "Fine-grained complexity and algorithm design" program at the Simons Institute for Theory of Computing in Berkeley.

possible, assuming the Exponential-Time Hypothesis (ETH) [8, 22, 26]. This means that we have an essentially tight understanding of these problems when considering graphs in their full generality, but it does not rule out the possibility of improved algorithms when restricted to some class of graphs. Indeed, many of these problems become significantly easier on certain important graph classes. The most well-studied form of this improvement is the so-called "square root phenomenon" on planar graphs (and some if its generalizations): there is a large number of parameterized problems that admit $2^{O(\sqrt{k}\cdot \mathrm{polylog} k)}n^{O(1)}$ time algorithms on planar graphs [7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20, 23, 24, 29, 30, 31]. Many of these positive results can be explained by the theory of bidimensionality [11] and explicity or implicitly rely on the relation between treewidth and grid minors.

Very recently, a superset of the present authors showed a new technique to obtain subexponential algorithms in planar graphs for problems related to the Subgraph Isomorphism problem [18], such as the Long Path problem of finding a simple path of length k in the input graph. The approach of [18] can be summarized as follows: a randomized polynomial-time algorithm is showed that, given a planar graph G and an integer k, selects a random induced subgraph of treewidth sublinear in k in such a manner that, for every connected k-vertex subgraph H of G, the probability that H survives in the selected subgraph is inversely-subexponential in k. Such a statement, dubbed *low-treewidth pattern covering*, together with standard dynamic programming techniques on graphs of bounded treewidth, gives subexponential algorithms for a much wider range of Subgraph Isomorphism-type problems than bidimensionality; for example, while bidimensionality provides a subexponential algorithm for Long Path in undirected graphs, it seems that the new approach of [18] is needed for directed graphs.

The proof of the low treewidth pattern covering theorem of [18] involves a number of different partitioning techniques in planar graphs. In this work, we take one of these techniques – called *clustering procedure*, based on the metric decomposition tool of Linial and Saks [25] and the recursive decomposition used in the construction of Bartal's hierarchically well-separated trees (so-called HSTs) [3] – and observe that it is perfectly suited to tackle the so-called *graphs of polynomial growth*.

To explain this concept formally, let us introduce some notation. All graphs in this paper are unweighted, and the distance function $\mathrm{dist}_G(u,v)$ measures the minimum possible number of edges on a path from u to v in G. For a graph G, integer r, and vertex $v \in V(G)$ by $B_G(v,r)$ we denote the set of vertices $w \in V(G)$ that are within distance *less than* r from v in G, $B_G(v,r) = \{w \in V(G) : \mathrm{dist}_G(v,w) < r\}$, while by $\partial B_G(v,r)$ we denote the set of vertices within distance *exactly* r, that is, $\partial B_G(v,r) = \{w \in V(G) : \mathrm{dist}_G(v,w) = r\}$. We omit the subscript if the graph is clear from the context.

▶ **Definition 1.1** (polynomial growth, [4])**.** We say that a graph G (or a graph class $\mathcal{G}$) has *polynomial growth* of degree (growth rate) δ if there exists a universal constant C such that for (every graph $G \in \mathcal{G}$ and) every radius r and every vertex $v \in V(G)$ we have

$$|B(v,r)| \leq C \cdot r^{\delta}.$$

The algorithmic consequences (and some of its variants) of this definition have been studied in the literature in various contexts (see, for example, [2, 21, 4, 1]). A standard example of a graph of polynomial growth with degree δ is a δ-dimensional grid. Graph classes of polynomial growth include graphs of bounded doubling dimension (with unit-weight edges), a popular assumption restricting the growth of a metric space in approximation algorithms or routing in networks (cf. the thesis [5] of Chan or [1] and references therein).

Our main result is the following low treewidth pattern covering statement.

▶ **Theorem 1.2.** *For every graph class $\mathcal{G}$ of polynomial growth with growth rate δ, there exists a polynomial-time randomized algorithm that, given a graph $G \in \mathcal{G}$ and an integer k, outputs a subset $A \subseteq V(G)$ with the following properties:*

1. *the treedepth of $G[A]$ is $\mathcal{O}(k^{1-\frac{1}{1+\delta}} \log k)$;*
2. *for every set $X \subseteq V(G)$ of size at most k, the probability that $X \subseteq A$ is $2^{-\mathcal{O}(k^{1-\frac{1}{1+\delta}} \log^2 k)}$.*

Note that Theorem 1.2 uses the notion of *treedepth*, a much more restrictive graph measure than treewidth (cf. [28]), that in particular implies the same treewidth bound. Thus, together with standard dynamic programming techniques on graphs of bounded treewidth, Theorem 1.2 gives the following.

▶ **Corollary 1.3.** *There exist randomized parameterized algorithms with running time bound $2^{\mathcal{O}(k^{1-\frac{1}{1+\delta}} \log^2 k)} n^{\mathcal{O}(1)}$ for* Long Path *and* Steiner Tree *parameterized by the size of the solution, when restricted to a graph class of polynomial growth with growth rate δ.*

In the corollary above we only listed the two most classic applications, refraining from repeating the lengthy discussion on the applications of low treewidth pattern covering statements that can be found in the introduction of Fomin et al. [18].

We complement the algorithmic statement of Theorem 1.2 with the following lower bound.

▶ **Theorem 1.4.** *If there exists an integer $\delta \geq 3$, a real $\varepsilon > 0$, and an algorithm that decides if a given subgraph of a δ-dimensional grid of side length n contains a Hamiltonian path in time $2^{\mathcal{O}(n^{\delta-1-\varepsilon})}$, then the ETH fails.*

Since a subgraph of a δ-dimensional grid of side length n has polynomial growth with degree at most δ and at most n^δ vertices, Theorem 1.4 shows that, unless the ETH fails, one cannot hope for a better term than $k^{1-\frac{1}{\delta}}$ in the low treewidth pattern covering statement as in Theorem 1.2.

2 Upper bound: proof of Theorem 1.2

In this section we prove Theorem 1.2. Without loss of generality, we assume $k \geq 4$.

Our main tool is a clustering procedure, or metric decomposition tool of [25], which can be informally described as follows. As long as the analysed graph G is not empty, we carve out a new cluster as follows. We pick any vertex $v \in V(G)$ as a center of the new cluster, and set its radius $r := 1$. Iteratively, with some chosen probablity p, we accept the current radius, and with the remaining probability $1 - p$ we increase r by one and repeat. That is, we choose r with geometric distribution with success probability p. Once a radius r is accepted, we set $B_G(v, r)$ as a new cluster, and delete $B_G(v, r) \cup \partial B_G(v, r)$ from G. In this manner, $B_G(v, r)$ is carved out as a separated cluster, at the cost of sacrificing $\partial B_G(v, r)$. A typical usage would be as follows: If one chooses p of the order of k^{-1}, then a simple analysis shows that every cluster has radius $\mathcal{O}(k \log n)$ w.h.p., while a fixed set $X \subseteq V(G)$ of size k is fully retained in the union of clusters with constant probability.

We apply the aforementioned clustering procedure in two steps. In the first one, we use $p \sim k^{-1}$ and the goal is to chop the graph into components of radius $\mathcal{O}(k \log k)$, which – by the polynomial growth property – are of polynomial size. The polynomial size bound is crucial for the second phase, when we consider every component independently, sparsifying it further using the clustering procedure with much higher cutoff probability, namely $p \sim k^{-\frac{1}{1+\delta}}$. These two steps are described in the subsequent two subsections.

We remark here that, because we rely only on the clustering procedure, and not the other arguments of [18], we do not need the assumption on the connectivity of the pattern $G[X]$. This assumption was essential for the planar case of [18].

2.1 Chopping the graph into parts of polynomial size

The goal of the first step is to delete a number of vertices from the graph so that on one hand every connected component of G has radius $\mathcal{O}(k \log k)$, and on the other hand the probability of deleting a vertex from an unknown vertex set $X \subseteq V(G)$ of size at most k is small. The proof of the following lemma is of the same nature as the clustering step in [18, Section 4.1 of the full version], with one subtlety: the obtained radii are of order $k \log k$ instead of $k \log n$. This improvement, crucial for the second step, heavily depends on the polynomial growth property.

▶ **Lemma 2.1.** *Let $\mathcal{G}$ be a graph class of polynomial growth with growth rate δ. There exists a constant $c_r > 0$ and a polynomial-time randomized algorithm that, given a graph $G \in \mathcal{G}$ and positive integer $k \geq 4$, outputs a subset $A \subseteq V(G)$ such that*
1. *every connected component of $G[A]$ is of radius at most $c_r k \log k$;*
2. *for every set $X \subseteq V(G)$ of size at most k, the probability that $X \subseteq A$ is at least $17/256$.*

Proof. For a sufficiently large constant $c_r > 0$ depending on the graph class $\mathcal{G}$, we perform the following iterative process. We start with $G_0 := G$ and $A_0 := \emptyset$. In i-th iteration ($i = 1, 2, 3, \ldots$), we consider the graph G_{i-1}. If the graph G_{i-1} is empty, we stop. Otherwise, we pick an arbitrary vertex $v_i \in V(G_{i-1})$ and pick a radius r_i according to the geometric distribution with success probability $1/k$, capped at value $R := c_r k \log k$ (i.e., if the choice of the radius is greater than R, we set $r_i := R$). For further analysis, we would like to look at the choice of the radius r_i as the following iterative process: we start with $r_i = 1$ and iteratively accept the current radius with probability $1/k$ or increase it by one and repeat with probability $1 - 1/k$, stopping unconditionally at radius R. Given v_i and r_i, we set $A_i := A_{i-1} \cup B_{G_{i-1}}(v_i, r_i)$ and $G_i := G_{i-1} - (B_{G_{i-1}}(v_i, r_i) \cup \partial B_{G_{i-1}}(v_i, r_i))$. That is, we remove from G_i all vertices within distance at most r_i from v_i, while retaining in A_i only those that are within distance less than r_i.

Clearly, as we remove a vertex from G_i at every step, the process stops after at most $|V(G)|$ steps. Let ι be the last index of the iteration. Consider the graph $G' := G[A_\iota]$. Recall that in the i-th step we put $B_{G_{i-1}}(v_i, r_i)$ into A_i, but remove not only $B_{G_{i-1}}(v_i, r_i)$ from G_{i-1} but also $\partial B_{G_{i-1}}(v_i, r_i) = N_{G_{i-1}}(B_{G_{i-1}}(v_i, r_i))$. Consequently, the vertex sets of the connected components of G' are exactly sets $B_{G_{i-1}}(v_i, r_i)$ for $1 \leq i \leq \iota$. Since the radii r_i are capped at value $R = c_r k \log k$, every connected component of G' has radius at most R.

We now claim the following.

▶ **Claim 2.2.** *For every $X \subseteq V(G)$ of size at most k, the probability that $X \subseteq V(G')$ is at least $17/256$.*

Proof. Fix $X \subseteq V(G)$ of size at most k. Note that $X \not\subseteq V(G')$ only if at some iteration i, some vertex $x \in X$ is exactly within distance r_i from v_i in the graph G_{i-1}. We now bound the probability that this happens, split into two subcases: either $r_i = R$ or $r_i < R$.

Case 1: hitting a vertex within distance $r_i = R$. Let $Y = \bigcup_{x \in X} B_G(x, R+1)$. Note that if $x \in X$ is exactly within distance $r_i \leq R$ from v_i in the graph G_{i-1}, then necessarily $v_i \in Y$. On the other hand, by the polynomial growth property,

$$|Y| \leq k \cdot C \cdot (R+1)^\delta = Ck(c_r k \log k + 1)^\delta = \mathcal{O}(k^{\delta+1} \log^\delta k).$$

We consider ourselves *lucky* if whenever $v_i \in Y$, we have $r_i < R$, that is, the process choosing r_i does not hit the cap of R for every center in Y. Note that, for a fixed iteration i, we have

$$\Pr(r_i = R) = \left(1 - \frac{1}{k}\right)^{R-1} = \left(1 - \frac{1}{k}\right)^{c_r k \log k - 1} \leq k^{-0.1 \cdot c_r}.$$

Thus, for sufficiently large constant c_r (depending only on C and δ), we have that

$$\Pr(r_i = R) < (k \cdot |Y|)^{-1}.$$

We infer that, for such a choice of c_r, the probability that we are not lucky is at most $1/k$.

Case 2: hitting a vertex within distance $r_i < R$. It is convenient to think here of the choice of the radius r_i as an interative process that starts from $r_i = 1$, accepts the current radius with probability $1/k$, or increases it by one and repeats with probability $1 - 1/k$. For a fixed iteration i and a choice of v_i, consider a potential radius $r_i < R$ when there is a vertex $x \in X$ within distance exactly r_i from v_i in G_{i-1}. If we do not accept this radius (which happens with probability $1 - 1/k$), the vertex x is included in $B_{G_{i-1}}(v_i, r_i)$ and is surely included in G'. Consequently, in the whole process we care about not accepting a given radius only k times, at most once for every vertex $x \in X$. We infer that the probability that for some iteration i there is a vertex $x \in X$ within distance exactly r_i from v_i and $r_i < R$ is at most $1 - (1 - 1/k)^k$.

Considering both cases, by union bound, the probability that $X \subseteq V(G')$ is at least

$$1 - \left(1 - \left(1 - \frac{1}{k}\right)^k + \frac{1}{k}\right) = \left(1 - \frac{1}{k}\right)^k - \frac{1}{k} \geq \frac{17}{256}.$$

The last estimate uses the assumption $k \geq 4$. ◀

Claim 2.2 concludes the proof of Lemma 2.1. ◀

2.2 Handling a component of polynomial size

▶ **Lemma 2.3.** *Let $\mathcal{G}$ be a graph class of polynomial growth with growth rate δ. For every constant $c_r > 0$ there exists a constant $c > 0$ and a polynomial-time randomized algorithm that, given a positive integer k, and a connected graph $G \in \mathcal{G}$ of radius $c_r k \log k$, outputs a subset $A \subseteq V(G)$ such that*

1. *the treedepth of $G[A]$ is $\mathcal{O}(k^{1 - \frac{1}{1+\delta}} \log k)$;*
2. *for every set $X \subseteq V(G)$ of size at most k, the probability that $X \subseteq A$ is at least $2^{-c \cdot |X| \cdot k^{-\frac{1}{1+\delta}} \cdot \log^2 k}$.*

We emphasize here the linear dependency on $|X|$ in the exponent of the probability bound. This dependency, similarly as in the analysis of [18], allows us to easily analyse independent runs of the algorithm on multiple connected components.

To prove Lemma 2.3, we again use the clustering procedure, but with a significantly higher cutoff probability, namely of the order of $k^{-\frac{1}{1+\delta}}$, as opposed to k^{-1} from the previous section. This yields clusters of sublinear size, namely of size roughly $k^{\frac{\delta}{1+\delta}}$. However, this comes with a cost: we can no longer claim that the solution X survives in the clustered graph with large probability, but – on average – $k^{\frac{\delta}{1+\delta}}$ vertices of X of size k will be deleted by the clustering clustering procedure. To recover from that, we crucially depend on the fact that the graph

has size polynomial in k: there is only a subexponential, namely $\binom{\mathrm{poly}(k)}{k^{\frac{\delta}{1+\delta}}} = 2^{\mathcal{O}(k^{1-\frac{1}{1+\delta}}\log k)}$, number of choices for the removed vertices of X, and we can afford to guess them.

Let us make a quick comparison with the techniques of [18]. The usage of the clustering technique in Lemma 2.3 is significantly different than the one in [18, Section 4.1 of the full version]: we choose a higher cutoff probability, which leads to smaller radii, at the cost of allowing some vertices of the set X on the boundary (that need to be subsequently guessed). The charging argument used here (Claim 2.5) is inspired by the argument of [18, Claim 28 in the full version]. However, the reason why we obtain sublinear treedepth (Claim 2.4) and the consequent tradeoffs in the exponent are specific to our polynomial growth setting.

Let us now proceed with the formal arguments.

Proof of Lemma 2.3. The random process we employ is similar to the one of the previous section, but more involved. Let $c'_r > 0$ be a constant to be fixed later.

We start with $G_0 = G$, $A_0 = \emptyset$ and $B_0 = \emptyset$. In the i-th iteration of the process, we consider the graph G_{i-1}. If the graph G_{i-1} is empty, we stop. Otherwise, we pick an arbitrary vertex $v_i \in V(G_{i-1})$ and pick a radius r_i according to the geometric distribution with success probability $k^{-1/(1+\delta)}\log k$, capped at value $R' := c'_r k^{1/(1+\delta)}$ (i.e., as before, if the choice of the radius is greater than R', we set $r_i := R'$). In other words, we start with $r_i = 1$ and iteratively accept the current radius with probability $k^{-1/(1+\delta)}\log k$ or increase it by one and repeat with the remaining probability, stopping unconditionally at radius R'.

As before, we set $A_i := A_{i-1} \cup B_{G_{i-1}}(v_i, r_i)$ and $G_i := G_{i-1} - (B_{G_{i-1}}(v_i, r_i) \cup \partial B_{G_{i-1}}(v_i, r_i))$. However, now, as the radii are smaller, we may want to retain some vertices of $\partial B_{G_{i-1}}(v_i, r_i)$, as they can be part of the vertex set X; for this, we use the sets B_i. With probability $1 - 1/(k|V(G)|)$ we put $P_i = \emptyset$ and $B_i = B_{i-1}$. With the remaining probability, we proceed as follows. Uniformly at random, we choose a number $1 \leq \ell_i \leq k^{1-1/(1+\delta)}\log k$ and a set P_i of ℓ_i vertices of $\partial B_{G_{i-1}}(v_i, r_i)$ (or all of them, if there are less than ℓ_i vertices in this set). We put $B_i := B_{i-1} \cup P_i$.

Let i_0 be the index of the last iteration. If $|B_{i_0}| > k^{1-1/(1+\delta)}\log k$, then we output $A = \emptyset$. Otherwise, we output $A := A_{i_0} \cup B_{i_0}$. Let us now verify that A has the desired properties.

▶ **Claim 2.4.** *The treedepth of $G[A]$ is $\mathcal{O}(k^{\delta/(1+\delta)}\log k)$.*

Proof. The claim is trivial if $A = \emptyset$, so assume otherwise; in particular, $|B_{i_0}| \leq k^{1-1/(1+\delta)}\log k$. We use the following inductive definition of treedepth: the treedepth of an empty graph is 0, while for any graph G on at least one vertex we have that

$$\mathrm{treedepth}(G) = \begin{cases} 1 + \min\{\mathrm{treedepth}(G-v) : v \in V(G)\} & \text{if } G \text{ is connected} \\ \max\{\mathrm{treedepth}(C) : C \text{ connected component of } G\} & \text{otherwise.} \end{cases}$$

Upon deleting from $G[A]$ the at most $k^{1-1/(1+\delta)}\log k$ vertices of B_{i_0}, we are left with $G[A_{i_0}]$. Similarly as in the previous section, every connected component of $G[A_{i_0}]$ is of radius at most $R' = c'_r k^{1/(1+\delta)}$. Consequently, every connected component of $G[A_{i_0}]$ is of size at most $C \cdot (c'_r)^\delta k^{\delta/(1+\delta)}$. The claim follows. ◀

▶ **Claim 2.5.** *For every set $X \subseteq V(G)$ of size at most k, the probability that $X \subseteq A$ is at least $2^{-c|X|k^{-1/(1+\delta)}\log^2 k}$ for some constant $c > 0$ depending only on c_r, δ, and C.*

Proof. Fix a vertex set X. The claim is trivial for $X = \emptyset$ so assume otherwise. In particular, as $|X| \geq 1$, then we can estimate the desired probability as

$$2^{-c|X|k^{-1/(1+\delta)}\log^2 k} \leq 2^{-ck^{-1/(1+\delta)}\log^2 k} = 1 - \Omega\left(\frac{\log^2 k}{k^{1/(1+\delta)}}\right). \tag{1}$$

Consider a fixed iteration i, and the moment when, knowing v_i, we choose the radius r_i. Given G_{i-1} and v_i, we say that a radius r is *bad* if

$$\left|X \cap \partial B_{G_{i-1}}(v_i, r)\right| > \left(k^{-1/(1+\delta)} \log k\right) \cdot \left|X \cap B_{G_{i-1}}(v_i, r)\right|. \tag{2}$$

Let $1 \leq r^0 < r^1 < r^2 < \ldots < r^t$ be a sequence of bad radii. First, note that $X \cap \partial B(v_i, r^0) \neq \emptyset$, and thus $|X \cap B(v_i, r^1)| \geq 1$. Furthermore, as for every $j \geq 1$ we have $\partial B(v_i, r^j) \subseteq B(v_i, r^{j+1})$, we have

$$|X \cap B(v_i, r^{j+1})| \geq \left(1 + k^{-1/(1+\delta) \log k}\right) |X \cap B(v_i, r^j)|.$$

Consequently,

$$|X \cap B(v_i, r^j)| \geq \left(1 + k^{-1/(1+\delta) \log k}\right)^{j-1}.$$

Since $|X| \leq k$, we infer that

$$t < 10k^{1/(1+\delta)}. \tag{3}$$

We are interested in the following event $\mathbf{A}$: every chosen radius r_i is not bad and is smaller than R' (i.e., we did not hit the cap of R'). Recall the iterative interpretation of the choice of the radii r_i: we start with $r_i = 1$, accept the current radius with probability $k^{-1/(1+\delta)} \log k$, or increase r_i by one and repeat with the remaining probability. Thus, we are interested in the intersection of the following two events: we do not accept any bad radius, but we accept some good radius before the cap R'.

Whenever we do not accept a bad radius r, a vertex of $X \cap \partial B(v_i, r)$ is included in $B(v_i, r_i) \subseteq A_i$. Consequently, in the whole algorithm we encounter at most $|X|$ bad radii; each is independently accepted with probability $k^{-1/(1+\delta)} \log k$.

By (3), in a fixed iteration i there are at most $10k^{1/(1+\delta)}$ bad radii. Consequently, if we count only acceptance of good radii, the probability that the radius r_i reaches the bound R' is at most

$$\left(1 - k^{-1/(1+\delta)} \log k\right)^{(c'_r - 10)k^{1/(1+\delta)}} \leq k^{-0.1c'_r}.$$

Consequently, since $|V(G)| \leq C \cdot (c_r k \log k)^\delta$, by choosing c'_r large enough, we can ensure that the probability that there exists a radius r_i equal to R' is at most k^{-1}. Since the choices of acceptance of different radii are independent, we infer that the probability of the event $\mathbf{A}$ is at least

$$\left(1 - k^{-1}\right) \cdot \left(1 - k^{-1/(1+\delta)} \log k\right)^{|X|} \geq 2^{-c_1 |X| k^{-1/(1+\delta)} \log k}$$

for some positive constant c_1. Here, we have used (1) to estimate the first factor.

Assume that the event $\mathbf{A}$ happens, and let us fix one choice of v_i and r_i. Note that these choices determine the sets A_i and the graphs G_i; the only remaining random choices are whether to include some vertices into the sets B_i.

For an iteration i, define $X_i := X \cap \partial B_{G_{i-1}}(v_i, r_i)$. We are now considering the following event $\mathbf{B}$: in every iteration i we have $P_i = X_i$. Note that if $\mathbf{B}$ happens, then $X \subseteq A$. Thus, we need to estimate the probability of the event $\mathbf{B}$.

If $X_i = \emptyset$, then we guess so with probability $1 - 1/(k|V(G)|)$. As there are at most $|V(G)|$ iterations, with probability at least $1 - 1/k$ we will make correct decision in all iterations i for which $X_i = \emptyset$.

Consider now an iteration i for which $X_i \neq \emptyset$. Since the radius r_i is good, we have

$$\left|X \cap \partial B_{G_{i-1}}(v_i, r_i)\right| \leq k^{-1/(1+\delta)} \log k \left|X \cap B_{G_{i-1}}(v_i, r_i)\right|. \tag{4}$$

In particular, $|X \cap B_{G_{i-1}}(v_i, r_i)| \geq k^{1/(1+\delta)}/\log k$, and thus there are at most $k^{\delta/(1+\delta)} \log k$ such iterations. Furthermore,

$$\left|\bigcup_{i=1}^{i_0} X_i\right| \leq |X| k^{-1/(1+\delta)} \log k.$$

In every such iteration i, we need to correctly guess that X_i is nonempty ($1/(k|V(G)|)$ success probability), correctly guess $\ell_i = |X_i|$ (at least $1/k$ success probability) and correctly guess $P_i = X_i$ (at least $|V(G)|^{-|X_i|}$ success probability). All these choices are independent. Since $|V(G)|$ is bounded polynomially in k, the probability of the event $\mathbf{B}$ is at least

$$\left(1 - \frac{1}{k}\right) \cdot \prod_{i:X_i \neq \emptyset} \frac{1}{k|V(G)|} \cdot \frac{1}{k} \cdot \frac{1}{|V(G)|^{|X_i|}} \geq \left(1 - \frac{1}{k}\right) \cdot (|V(G)|^2 \cdot k)^{-|X| \cdot k^{-1/(1+\delta)} \log k}$$
$$\geq 2^{-c_2 |X| \cdot k^{-1/(1+\delta)} \log^2 k}$$

for some constant c_2 depending on c_r, δ, and C. This finishes the proof of the claim. ◀

Lemma 2.3 follows directly from Claims 2.4 and 2.5. ◀

2.3 Summary

Let us now wrap up the proof of Theorem 1.2, using Lemmata 2.1 and 2.3. We first apply the algorithm of Lemma 2.1 to the input graph G and integer k, obtaining a set $A_0 \subseteq V(G)$. Then, we apply the algorithm of Lemma 2.3 independently to every connected component C of $G[A_0]$, obtaining a set $A_C \subseteq C$; recall that every such component is of radius at most $R = c_r k \log k$. As the output A, we return the union of the returned sets A_C. Clearly, the treedepth bound holds. If we denote $X_C := X \cap C$ for a component C, we have that the probability that $X \subseteq A$ is at least

$$\frac{17}{256} \cdot \prod_C 2^{-c|X_C| k^{-1/(1+\delta)} \log^2 k} \geq \frac{17}{256} \cdot 2^{-ck^{1-1/(1+\delta)} \log^2 k}.$$

This finishes the proof of Theorem 1.2.

3 Lower bound: proof of Theorem 1.4

In this section we prove Theorem 1.4. The reduction is heavily inspired by the reduction for δ-dimensional Euclidean TSP by Marx and Sidiropolous [27]. In particular, our starting point is the same CSP pivot problem.

▶ **Theorem 3.1** ([27]). *For every fixed $\delta \geq 2$, there is a constant λ_δ such that for every constant $\varepsilon > 0$ an existence of an algorithm solving in time $2^{\mathcal{O}(n^{\delta-1-\varepsilon})}$ CSP instances with binary constraints, domain size at most λ_δ, and Gaifman graph being a δ-dimensional grid of side length n would refute ETH.*

Let us recall that a *binary CSP instance* consists of a *domain* D, a set V of *variables*, and a set E of *constraints*. Every constraint is a binary relation $\psi_{u,v} \subseteq D \times D$ that binds two variables $u, v \in V$. The goal is to find an assignment $\phi : V \to D$ that satisfies every

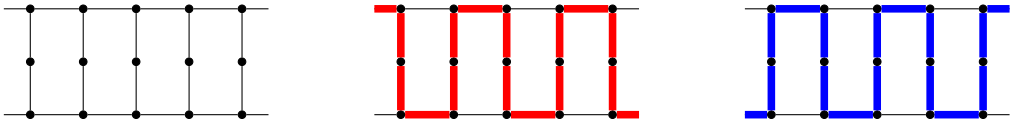


Figure 1 A 2-chain with two ways how a Hamiltonian path can traverse it, called henceforth *modes*.

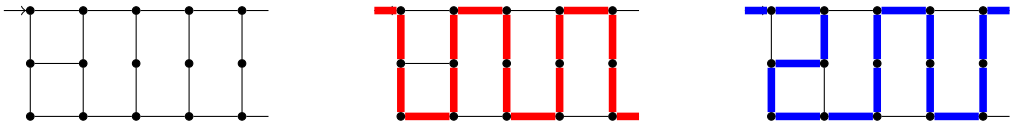


Figure 2 An endpoint of a 2-chain, allowing traversing the 2-chain in both modes.

constraint; a constraint $\psi_{u,v}$ is satisfied if $(\phi(u), \phi(v)) \in \psi_{u,v}$. The *Gaifman graph* of a binary CSP instance has vertex set V and an edge uv for every constraint $\psi_{u,v}$.

Similarly as in the case of [27], our goal is to take a given CSP instance as in Theorem 3.1 and turn it into a Hamiltonian path instance by local gadgets. That is, we are going to replace every variable of the CSP instance with a constant-size gadget (i.e., with size depending only on δ and λ_δ); the way the gadget is traversed by the Hamiltonian path indicates the choice of the value of the variable. The neighboring gadgets are wired up to ensure that the constraint binding them is satisfied.

More formally, let us fix an integer $\delta \geq 3$. The input of a reduction is a CSP instance as in Theorem 3.1: of domain size at most λ_δ and whose Gaifman graph is a δ-dimensional grid of size length n. The output is a subgraph of a δ-dimensional grid of side length cn for some constant c depending only on δ and λ_δ that has a Hamiltonian path if and only if the input CSP instance is satisfiable.

Let us fix a δ-dimensional graph of side length cn for some sufficiently large constant c to be defined later (we will see that $c = \Theta(\delta\lambda_\delta^2)$ suffices). We partition this grid into n^δ subgrids of side length c, each corresponding to a variable of the input CSP instance in a natural fashion.

3.1 2-chains

The base gadget of the construction is a 2-chain as presented on Figure 1. A direct check shows that there are two ways how a 2-chain can be traversed by a Hamiltonian path, as depicted on the figure.

Figure 2 shows a gadget present on both left and right endpoints of a 2-chain. As shown on the figure, it allows choosing how the 2-chain is traversed.

We will refer to the two depicted Hamiltonian paths of a 2-chain as *modes* of the chain. Given one of the horizontal edges of the 2-chain, a mode is *consistent* with this edge if the corresponding Hamiltonian path traverses the edge in question, and *inconsistent* otherwise.

We will attach various gadgets to 2-chains via one of the horizontal edges. To maintain the properties of the 2-chains, in particular the effectively two ways of traversing a 2-chain, we need to space out the attached gadgets. More formally, we partition every 2-chain into sufficiently long chunks (chunks of length 8 are more than sufficient), and allow gadgets to attach only to one of the two middle horizontal edges on one side of the chain (see Figure 3), with at most one gadget per chunk. A gadget is always attached to an edge e by adding two new vertices u and v near the edge e, in the same 2-dimensional plane as the 2-chain itself,

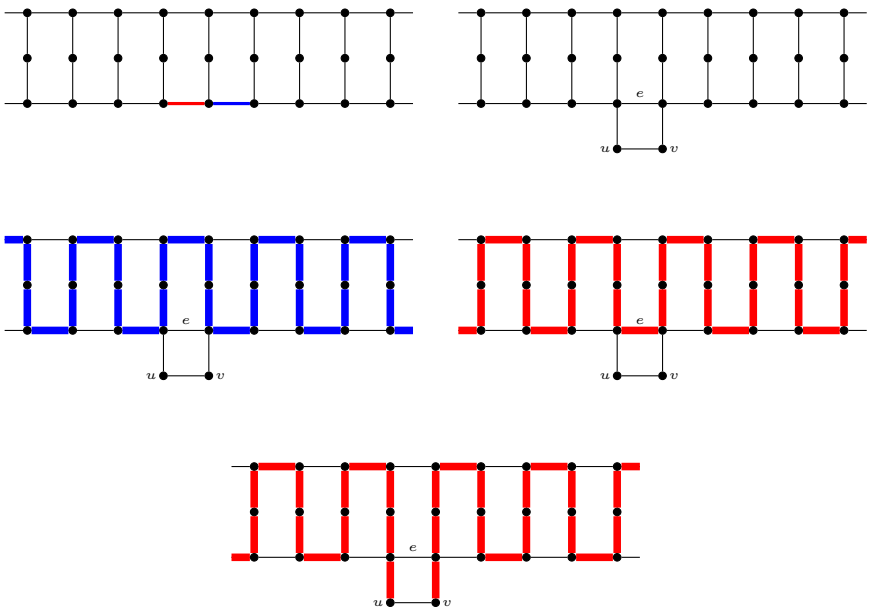


Figure 3 From top to bottom, left to right: a chunk on a 2-chain, with two attachment edges marked red and blue; a standard attachment of a gadget; three ways how a 2-chain with attached gadget can be traversed.

such that the endpoints of e, u, and v form a square. Properties of such an attachment can be summarized in the following straightforward claim.

▶ **Claim 3.2.** *Consider a chunk c on a 2-chain A, and a gadget attached to an edge e in c. Then every Hamiltonian path traverses c in one of the following three ways (see Figure 3):*

1. *as on Figure 1, inconsistently with e;*
2. *as on Figure 1, consistently with e;*
3. *as on Figure 1, consistently with e, but with the edge e replaced with an edge towards vertex u and towards vertex v.*

In particular, Claim 3.2 allows us to formally speak about a mode of a 2-chain, even if multiple gadgets are attached to it.

3.2 Placing 2-chains

For every variable of the input CSP instance, we create λ_δ 2-chains of length $L = \mathcal{O}(d\lambda_\delta)$ (to be determined later). They are positioned parallelly in the following fashion (see Figure 4): we choose an arbitrary 3-dimensional subspace of the δ-dimensional subgrid of sidelength c devoted to a particular variable, and place 2-chains such that the i-th 2-chain occupies vertices $\{0, 1, \ldots, L\} \times \{0, 1, 2\} \times \{i\}$. The edges indicated as attachment points for gadgets are on the one side of all chains.

All chains, for all variables, are wired up into a Hamiltonian path: for every variable, we connect the constructed 2-chains into a path in a straightforward fashion, we take an arbitrary Hamiltonian path of the original Gaifman graph of the input CSP instance (which is a δ-dimensional grid, and thus trivially admits a Hamiltonian path), and connect endpoints of the 2-chains in the same order using simple paths. This is straightforward to perform if we space out the variable gadgets enough.

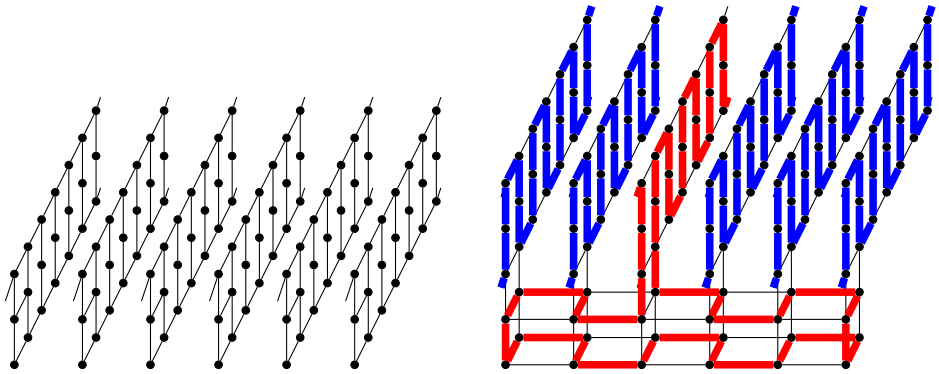


Figure 4 Left: Placing parallel 2-chains for a single variable x. Right: A tube gadget attached to the 2-chains, with intended Hamiltonian path.

Since all constructed 2-chains are isomorphic, we indicate one mode of a 2-chain as a *low mode*, and the other one as *high mode*. Our goal is to introduce gadgets that (i) ensure that for every variable, exactly one of the corresponding 2-chains is in high mode, indicating the choice of the value for this variable; (ii) for every two variables that are bound by a constraint, for every pair of values that is forbidden by the constraint, ensure that the two variables in question do not attain the values in question at the same time, that is, the corresponding two 2-chains are not both in high mode at the same time.

3.3 OR-checks

The construction of 2-chains allow us to implement a simple "OR" constraint on two 2-chains. Consider two 2-chains A and B, and two horizontal edges e_A and e_B on A and B, respectively. By attaching an OR-check to these edges we mean the following construction:

1. we create vertices u_A and v_A near e_A as well as u_B and v_B near e_B, as in the description of gadget attachment;
2. we connect u_A to u_B by a path and v_A to v_B by a path.

If the 2-chains are spaced enough, it is straightforward to implement the above construction such that the resulting graph is a subgraph of a d-dimensional grid.

Claim 3.2 allows us to observe the following.

▶ **Claim 3.3.** *If A is traversed in a way consistent with e_A, then one can modify the Hamiltonian path traversing A so that it visits the OR gadget: replace e_A with a path traversing first a path from u_A to u_B, the edge u_Bv_B, and then the path from v_B to v_A. A symmetrical claim holds if B is traversed in a way consistent with e_B.*

In the other direction, there is no Hamiltonian path that traverses both A and B in a way inconsistent with e_A and e_B, respectively.

We now observe that, by attaching OR-checks in a straightforward manner, we can ensure that:

1. for every variable x, at most one 2-chain corresponding to x is in high mode (we wire up every pair of 2-chains with an OR-check forbidding two high modes at the same time);
2. for every two variables x and y that are bound by a constraint ψ, for every pair of values (α_x, α_y) that is forbidden by the constraint ψ, the α_x-th 2-chain of x and the α_y-th 2-chain of y are not in the high mode at the same time.

We are left with ensuring that for every variable x, at least one of the corresponding 2-chains is in the high mode. This is the aim of the next gadget.

3.4 Tube gadget

Fix a variable x. Without loss of generality, we can assume that the first chunk of every 2-chain for x has not been used by the OR-checks introduced previously. Let e_i be the attachment edge of the i-th 2-chain that is consistent with the high mode of the 2-chain; note that the edges e_i lie next to each other (see Figure 4).

We create a $2 \times 2 \times \lambda_\delta$ grid, called henceforth a *tube gadget*, placed near the edges e_i, such that every edge e_i can be attached to an edge of the grid in a standard way discussed earlier. See Figure 4 for an illustration.

Since a $2 \times 2 \times \lambda_\delta$ grid admits a Hamiltonian cycle that traverses every edge in one of the first two dimensions, if the i-th chain is traversed in high mode for some i, we can replace e_i on the Hamiltonian path with a traversal along the aforementioned Hamiltonian cycle. This observation, together with Claim 3.2, proves the following claim.

▶ **Claim 3.4.** *If there exists an index i such that the i-th 2-chain is traversed in high mode, then the Hamiltonian path of this 2-chain can be altered to visit every vertex of the $2 \times 2 \times \lambda_\delta$ grid.*

On the other hand, any Hamiltonian path of the entire graph needs to traverse at least one 2-chain in high mode, in order to visit the vertices of the $2 \times 2 \times \lambda_\delta$ grid.

3.5 Summary

The tube gadgets ensure that, for every variable, at least one corresponding 2-chain is in high mode. The first type of the attached OR-checks ensure that at most one such 2-chain is in high mode. Thus, effectively the gadgets introduced for a single variable x can be in one of λ_δ by choosing the 2-chain that is in high mode, which corresponds to the choice of the value for x in an assignment.

The second type of the attached OR-checks ensure that the values of the neighboring variables satisfy the constraint that binds them, completing the proof of the correctness of the reduction.

To conclude, let us observe that every 2-chain is attached to one tube gadget and $\mathcal{O}(\delta\lambda_\delta)$ OR-checks, and the whole gadget replacing a single variable takes part in $\mathcal{O}(\delta\lambda_\delta^2)$ OR-checks. Thus taking $L = \mathcal{O}(\delta\lambda_\delta^2)$ suffices. By leaving space of size $\mathcal{O}(\delta\lambda_\delta^2)$ between consecutive variable gadgets we can ensure more than enough space for all connections. This gives $c = \mathcal{O}(\delta\lambda_\delta^2)$, that is, the constructed graph is a subgraph of a d-dimensional grid of side length $\mathcal{O}(\delta\lambda_\delta^2 n)$, and admits a Hamiltonian path if and only if the input CSP instance is satisfiable. This finishes the proof of Theorem 1.4.

4 Conclusions

We have shown a low treewidth pattern covering statement for graphs of polynomial growth with subexponential term being $2^{k^{1-\frac{1}{1+\delta}}}$, where δ is the growth rate of the graph class. An almost tight lower bound shows that, assuming ETH, one should not hope for a better term than $2^{k^{1-\frac{1}{\delta}}}$.

Two natural questions arise. The first one is to close the gap between $\frac{1}{1+\delta}$ and $\frac{1}{\delta}$; we conjecture that our lower bound is tight, and the term $k^{1-\frac{1}{1+\delta}}$ in the running time bound

of Theorem 1.2 is only a shortfall of our algorithmic techniques. The second one is to derandomize the algorithms of this work and of [18]. The clustering step is the only step of the algorithm of [18] that we do not know how to derandomize, despite its resemblance to the construction of Bartal's HSTs [3] that was subsequently derandomized [6].

References

1. Ittai Abraham, Cyril Gavoille, Andrew V. Goldberg, and Dahlia Malkhi. Routing in networks with low doubling dimension. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006), 4-7 July 2006, Lisboa, Portugal*, page 75. IEEE Computer Society, 2006. doi:10.1109/ICDCS.2006.72.
2. Ittai Abraham and Dahlia Malkhi. Name independent routing for growth bounded networks. In Phillip B. Gibbons and Paul G. Spirakis, editors, *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, pages 49–55. ACM, 2005. doi:10.1145/1073970.1073978.
3. Yair Bartal. On approximating arbitrary metrices by tree metrics. In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 161–168. ACM, 1998. doi:10.1145/276698.276725.
4. Vincent Blondel, Kyomin Jung, Pushmeet Kohli, and Devavrat Shah. Partition-merge: Distributed inference and modularity optimization. *CoRR*, abs/1309.6129, 2013. URL: http://arxiv.org/abs/1309.6129.
5. T. H. Hubert Chan. *Approximation Algorithms for Bounded Dimensional Metric Spaces*. PhD thesis, Carnagie Mellon University, 2007. Available at http://i.cs.hku.hk/~hubert/thesis/thesis.pdf.
6. Moses Charikar, Chandra Chekuri, Ashish Goel, Sudipto Guha, and Serge A. Plotkin. Approximating a finite metric by a small number of tree metrics. In *39th Annual Symposium on Foundations of Computer Science, FOCS'98, November 8-11, 1998, Palo Alto, California, USA*, pages 379–388. IEEE Computer Society, 1998. doi:10.1109/SFCS.1998.743488.
7. Rajesh Hemant Chitnis, MohammadTaghi Hajiaghayi, and Dániel Marx. Tight bounds for Planar Strongly Connected Steiner Subgraph with fixed number of terminals (and extensions). In *SODA 2014*, pages 1782–1801, 2014. doi:10.1137/1.9781611973402.129.
8. Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.
9. Erik D. Demaine, Fedor V. Fomin, Mohammad Taghi Hajiaghayi, and Dimitrios M. Thilikos. Bidimensional parameters and local treewidth. *SIAM J. Discrete Math.*, 18(3):501–511, 2004. doi:10.1137/S0895480103433410.
10. Erik D. Demaine, Fedor V. Fomin, Mohammad Taghi Hajiaghayi, and Dimitrios M. Thilikos. Fixed-parameter algorithms for (k, r)-Center in planar graphs and map graphs. *ACM Transactions on Algorithms*, 1(1):33–47, 2005. doi:10.1145/1077464.1077468.
11. Erik D. Demaine, Fedor V. Fomin, Mohammad Taghi Hajiaghayi, and Dimitrios M. Thilikos. Subexponential parameterized algorithms on bounded-genus graphs and H-minor-free graphs. *J. ACM*, 52(6):866–893, 2005.
12. Erik D. Demaine and Mohammad Taghi Hajiaghayi. Fast algorithms for hard graph problems: Bidimensionality, minors, and local treewidth. In *Graph Drawing*, pages 517–533, 2004. doi:10.1007/978-3-540-31843-9_57.
13. Erik D. Demaine and MohammadTaghi Hajiaghayi. The bidimensionality theory and its algorithmic applications. *Comput. J.*, 51(3):292–302, 2008. doi:10.1093/comjnl/bxm033.

14 Erik D. Demaine and MohammadTaghi Hajiaghayi. Linearity of grid minors in treewidth with applications through bidimensionality. *Combinatorica*, 28(1):19–36, 2008. doi:10.1007/s00493-008-2140-4.

15 Frederic Dorn, Fedor V. Fomin, Daniel Lokshtanov, Venkatesh Raman, and Saket Saurabh. Beyond bidimensionality: Parameterized subexponential algorithms on directed graphs. In *STACS 2010*, pages 251–262, 2010. doi:10.4230/LIPIcs.STACS.2010.2459.

16 Frederic Dorn, Fedor V. Fomin, and Dimitrios M. Thilikos. Subexponential parameterized algorithms. *Computer Science Review*, 2(1):29–39, 2008. doi:10.1016/j.cosrev.2008.02.004.

17 Frederic Dorn, Eelko Penninkx, Hans L. Bodlaender, and Fedor V. Fomin. Efficient exact algorithms on planar graphs: Exploiting sphere cut decompositions. *Algorithmica*, 58(3):790–810, 2010. doi:10.1007/s00453-009-9296-1.

18 Fedor V. Fomin, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. Subexponential parameterized algorithms for planar and apex-minor-free graphs via low treewidth pattern covering. In *FOCS*, pages 515–524, 2016. Full version available at http://arxiv.org/abs/1604.05999.

19 Fedor V. Fomin, Daniel Lokshtanov, Venkatesh Raman, and Saket Saurabh. Subexponential algorithms for partial cover problems. *Inf. Process. Lett.*, 111(16):814–818, 2011. doi:10.1016/j.ipl.2011.05.016.

20 Fedor V. Fomin and Dimitrios M. Thilikos. Dominating sets in planar graphs: Branch-width and exponential speed-up. *SIAM J. Comput.*, 36(2):281–309, 2006. doi:10.1137/S0097539702419649.

21 Ramakrishna Gummadi, Kyomin Jung, Devavrat Shah, and Ramavarapu S. Sreenivas. Computing the capacity region of a wireless network. In *INFOCOM 2009. 28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil*, pages 1341–1349. IEEE, 2009. doi:10.1109/INFCOM.2009.5062049.

22 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774.

23 Philip N. Klein and Dániel Marx. Solving planar k-terminal cut in $O(n^{c\sqrt{k}})$ time. In *Proceedings of the 39th International Colloquium of Automata, Languages and Programming (ICALP)*, volume 7391 of *Lecture Notes in Comput. Sci.*, pages 569–580. Springer, 2012.

24 Philip N. Klein and Dániel Marx. A subexponential parameterized algorithm for Subset TSP on planar graphs. In *SODA 2014*, pages 1812–1830, 2014. doi:10.1137/1.9781611973402.131.

25 Nathan Linial and Michael E. Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993. doi:10.1007/BF01303516.

26 Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Lower bounds based on the exponential time hypothesis. *Bulletin of the EATCS*, 105:41–72, 2011. URL: http://albcom.lsi.upc.edu/ojs/index.php/beatcs/article/view/96.

27 Dániel Marx and Anastasios Sidiropoulos. The limited blessing of low dimensionality: when 1-1/d is the best possible exponent for d-dimensional geometric problems. In Siu-Wing Cheng and Olivier Devillers, editors, *30th Annual Symposium on Computational Geometry, SOCG'14, Kyoto, Japan, June 08-11, 2014*, page 67. ACM, 2014. doi:10.1145/2582112.2582124.

28 Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity – Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012. doi:10.1007/978-3-642-27875-4.

29 Marcin Pilipczuk, Michał Pilipczuk, Piotr Sankowski, and Erik Jan van Leeuwen. Subexponential-time parameterized algorithm for Steiner Tree on planar graphs. In *STACS 2013*, pages 353–364, 2013. doi:10.4230/LIPIcs.STACS.2013.353.

30 Marcin Pilipczuk, Michał Pilipczuk, Piotr Sankowski, and Erik Jan van Leeuwen. Network sparsification for Steiner problems on planar and bounded-genus graphs. In *FOCS 2014*, pages 276–285. IEEE Computer Society, 2014.

31 Dimitrios M. Thilikos. Fast sub-exponential algorithms and compactness in planar graphs. In *ESA 2011*, pages 358–369, 2011. doi:10.1007/978-3-642-23719-5_31.

Benchmark Graphs for Practical Graph Isomorphism*

Daniel Neuen[1] and Pascal Schweitzer[2]

1 **RWTH Aachen University, Aachen, Germany**
neuen@informatik.rwth-aachen.de
2 **RWTH Aachen University, Aachen, Germany**
schweitzer@informatik.rwth-aachen.de

Abstract

The state-of-the-art solvers for the graph isomorphism problem can readily solve generic instances with tens of thousands of vertices. Indeed, experiments show that on inputs without particular combinatorial structure the algorithms scale almost linearly. In fact, it is non-trivial to create challenging instances for such solvers and the number of difficult benchmark graphs available is quite limited.

We describe a construction to efficiently generate small instances for the graph isomorphism problem that are difficult or even infeasible for said solvers.

Up to this point the only other available instances posing challenges for isomorphism solvers were certain incidence structures of combinatorial objects (such as projective planes, Hadamard matrices, Latin squares, etc.). Experiments show that starting from 1500 vertices our new instances are several orders of magnitude more difficult on comparable input sizes. More importantly, our method is generic and efficient in the sense that one can quickly create many isomorphism instances on a desired number of vertices. In contrast to this, said combinatorial objects are rare and difficult to generate and with the new construction it is possible to generate an abundance of instances of arbitrary size.

Our construction hinges on the multipedes of Gurevich and Shelah and the Cai-Fürer-Immerman gadgets that realize a certain abelian automorphism group and have repeatedly played a role in the context of graph isomorphism. Exploring limits, we also explain that there are group theoretic obstructions to generalizing the construction with non-abelian gadgets.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases graph isomorphism, benchmark instances, practical solvers

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.60

1 Introduction

The graph isomorphism problem, which asks for structural equivalence of two given input graphs, has been extensively investigated since the beginning of theoretical computer science, both from a theoretical and a practical point of view. Designing isomorphism solvers in practice is a non-trivial task. Nevertheless various efficient algorithms, namely nauty and traces [15], bliss [9], conauto [11], and saucy [6] are available as software packages. These state-of-the-art solvers for the graph isomorphism problem (or more generally graph canonization) can readily solve generic instances with tens of thousands of vertices. Indeed, experiments show that on inputs without particular combinatorial structure the algorithms scale almost

* A full version of the paper is available at https://arxiv.org/abs/1705.03686.

linearly. In practice, this sets the isomorphism problem aside from typical NP-complete problems which usually have an abundance of difficult benchmark instances. The practical algorithms underlying these solvers differ from the ones employed to obtain theoretical results. Indeed, there is a big disconnect between theory and practice [2]. One could interpret Babai's recent breakthrough, the quasipolynomial time algorithm [1], as a first step of convergence. The result implies that if graph isomorphism is NP-complete then all problems in NP have quasi-polynomial time algorithms, which may lead one to also theoretically believe that graph isomorphism is not NP-complete.

In this paper we are interested in creating difficult benchmark instances for the graph isomorphism problem. It would be a misconception to think that for unstructured randomly generated instances graph isomorphism should be hard in practice. Quite the opposite is true. Instances that encompass sufficient randomness usually turn out to be among the easiest instances (see [14] for extensive tests on various graphs and [3, 10] for theoretical arguments). In fact, it is non-trivial to create challenging instances for efficient isomorphism solvers and so far the number of difficult benchmark graphs available is quite limited.

The prime source for difficult instances are graphs arising from combinatorial structures. In 1978, Mathon [13] provided a set of benchmark graphs arising from combinatorial objects such as strongly regular graphs, block designs and coherent configurations. The actual graphs that are provided, having less than 50 vertices, are small by todays standards. Nowadays, larger combinatorial objects yielding difficult graphs are known. The most challenging examples are for instance incidence graphs of projective planes, Hadamard matrices, or Latin squares (see [14]). It is important to understand that not all such incidence structures yield difficult graphs. Indeed, typically there is an algebraic version that can readily be generated. For example to obtain a projective plane it is possible to consider incidences between one- and two-dimensional subspaces of a three-dimensional vector space over a finite field. If however one uses such an algebraic construction then the resulting graph automatically has a large automorphism group. The practical isomorphism solvers are tuned to finding such automorphisms and to exploit them for search space contraction. On top of that, there are also theoretical reasons that lead one to believe that graphs with automorphisms make easier examples for the practical tools (see [22, Theorem 9]). The non-algebraic counterparts of combinatorial structures often do not have the shortcoming of having a large automorphism group, but in contrast to the algebraic constructions they are rare and difficult to generate.

The second source of difficult instances is based on modifications of the so called Cai-Fürer-Immerman construction [5]. This construction is connected to the family of Weisfeiler-Leman algorithms, which for every integer k contains a k-dimensional version that constitutes a polynomial time algorithm used to distinguish non-isomorphic graphs. The 1-dimensional variant (usually called color refinement) is a subroutine in all competitive practical isomorphism solvers. For larger k, however, the k-dimensional variant becomes impractical due to excessive space consumption. For each k, Cai, Fürer and Immerman construct pairs of non-isomorphic graphs that are not distinguished by the k-dimensional variant of algorithm. In 1996, the construction was adapted by Miyazaki to explicitly show that the then current version of nauty has exponential running time in the worst case [16]. The Cai-Fürer-Immerman graphs and the Miyazaki graphs constitute families of benchmark graphs that (despite being specifically designed to fool the Weisfeiler-Leman isomorphism algorithm and the canonical labeling tool nauty, respectively) are infeasible for ad hoc graph isomorphism algorithms. Nowadays, however, most state-of-the-art solvers scale reasonably well on these instances.

Contribution. We describe a construction to efficiently generate instances on any desired number of vertices for the graph isomorphism problem that are difficult or even infeasible for all state-of-the-art graph isomorphism solvers.

Experiments show the graphs pose by far the most challenging graphs to date. Already for 1500 vertices our new instances are by several orders of magnitude more difficult than all previously available benchmark graphs on comparable input sizes. More importantly, the algorithm that generates the instances is generic and efficient in that one can quickly create an abundance of isomorphism instances on a desired number of vertices. We can thereby generate instances in size ranges for which no other difficult benchmark graphs are available.

Our construction, the resulting graphs of which we call shrunken multipedes, hinges on a construction of Gurevich and Shelah [8]. For every k, they describe combinatorial structures that are rigid but cannot be distinguished by the k-dimensional Weisfeiler-Leman algorithm. Guided by the intuition that rigid graphs (i.e., graphs without non-trivial automorphisms) constitute harder instances, we alter the construction to a simple efficient algorithm. In this algorithm we start with a bipartite rigid base graph that is obtained by connecting two explicit graphs (cycles with added diagonals) randomly. We then apply a suitable adaptation of the Cai-Fürer-Immerman (CFI) construction. We prove that the resulting graph is rigid asymptotically almost surely. Our experiments show that even for small sizes the graphs are already rigid with high probability. The result is a simple randomized algorithm that efficiently creates challenging instances of the graph isomorphism problem. The algorithm can be used to create instances for any desired size range. Moreover it is possible to create many non-isomorphic graphs such that every set of two of them forms a difficult instance.

To create practical benchmark graphs, it is imperative to keep all gadget constructions as small as possible. We therefore employ two techniques to save vertices without changing the local automorphism structure of the gadgets. The first technique is based on linear algebra and reduces the vertices in a manner that maintains the rigidity of the graphs. The second technique bypasses certain vertices of the gadgets used in the CFI-construction.

The CFI-gadgets have repeatedly played a role in the context of graph isomorphism. Exploring limits of such constructions, we also explain that there are group theoretic obstructions to generalizing the construction with non-abelian gadgets.

Finally we show with experimental data that our benchmark instances constitute quite challenging problems for all state-of-the-art isomorphism solvers. We compare the running times on the new benchmarks with the combinatorial graphs, the CFI-graphs and the Miyazaki graphs mentioned above.

Canonical forms vs. Isomorphism testing. The task of computing a canonical form is to compute a graph isomorphic to a given input graph so that the output only depends on the isomorphism type of the input and not the actual input (see [15]). Various practical applications require the computation of canonical forms rather than isomorphism tests.

The isomorphism problem reduces, theoretically and practically, to the task of computing a canonical form. Indeed, to check two graphs for isomorphism one computes their canonical forms and then performs a trivial equality check of the results. While computing a canonical form of a graph could in principle be harder than testing isomorphism, several (but not all) of the currently fastest isomorphism solvers actually compute a canonical labeling and then perform the equality check. Our graphs constitute difficult instances both for the task to compute canonical forms and for graph isomorphism. While for the former single graphs need to be constructed, for the latter we need pairs of graphs. (See Subsection 3.1.)

Theoretical bounds. The benchmark graphs described in this paper are explicitly designed to pose a challenge for practical isomorphism solvers. Since they have bounded degree they can be solved (theoretically) in polynomial time [12]. We should remark that we expect a tailored algorithm can be designed that reduces our benchmarks to instances of bounded color class size and then efficiently performs an isomorphism test. Concerning individualization-refinement algorithms, which the tools mentioned above are, using a different but related construction, exponential lower bounds can be proven. We refer to [20].

Obtaining benchmark graphs. Our families of benchmark graphs can be downloaded at `https://www.lii.rwth-aachen.de/research/95-benchmarks.html`. While some of our instances may constitute the most difficult instances yet, in practice, worst-case running time is not the most important measure. In many applications, the solvers have to sift through an enormous number of instances, most of which are easy. It is therefore important to perform extremely well on easy instances, and adequately on difficult instances, not the other way around. We refer to [14] for a well-rounded library in that regard. Nevertheless, we hope our benchmark graphs help to improve current and future isomorphism solvers.

2 Preliminaries

Graphs and isomorphism. A *graph* is a pair $G = (V, E)$ with vertex set $V = V(G)$ and edge relation $E = E(G)$. In this work all graphs are simple, undirected graphs. The *neighborhood* of $v \in V(G)$ is denoted $N(v)$. A *path* is a sequence $(v_1, \dots, v_\ell)$ of distinct vertices such that $\{v_i, v_{i+1}\} \in E(G)$ for all $i \in \{1, \dots, \ell - 1\}$. If additionally $\{v_1, v_\ell\} \in E(G)$ then the sequence $(v_1, \dots, v_\ell)$ is a *cycle* of G. An *isomorphism* from a graph G to another graph H is a bijective mapping $\varphi\colon V(G) \to V(H)$ which preserves the edge relation, that is $\{v, w\} \in E(G)$ if and only if $\{\varphi(v), \varphi(w)\} \in E(H)$ for all $v, w \in V(G)$. The notation $G \cong H$ indicates that such an isomorphism exists. The graph isomorphism problem asks, given two graphs G and H, whether $G \cong H$. An *automorphism* of a graph G is an isomorphism from G to itself. By $\mathrm{Aut}(G)$ we denote the automorphisms of G. A graph G is *rigid* (or *asymmetric*) if its automorphism group $\mathrm{Aut}(G)$ is trivial, that is, the only automorphism of G is the identity.

The Cai-Fürer-Immerman Construction. Our constructions presented in this paper make use of a construction of Cai, Fürer and Immerman [5]. In terms of the graph-isomorphism problem, they used the construction to show that for every k there is a pair of graphs not distinguished by the k-dimensional Weisfeiler-Leman algorithm. The basic building block for these graphs is the CFI gadget X_3. This gadget has 4 *inner vertices* $m_1, \dots, m_4$ and 3 pairs of *outer vertices* a_i, b_i with $i \in \{1, 2, 3\}$. The edges are a_1m_3, a_1m_4, b_1m_1, b_1m_2, a_2m_2, a_2m_4, b_2m_1, b_2m_3, a_3m_2, a_3m_3, b_3m_1, and b_3m_4. The crucial property of X_3 is that a bijection of the outer vertices mapping the set $\{a_i, b_i\}$ to itself for all $i \in \{1, 2, 3\}$ extends to an automorphism of X_3 if and only if an even number of pairs of outer vertices is swapped.

Now let G be a connected 3-regular graph called the *base graph* of the construction and let $T \subseteq E(G)$ be a subset of its edges. Then the CFI construction applied to G results in the following graph $\mathrm{CFI}(G, T)$. For every vertex $v \in V(G)$ with incident edges $e_i = (v, w_i) \in E(G)$ we add a copy of X_3. The inner vertices are denoted by $m_1(v), \dots, m_4(v)$. Each pair $\{a_i, b_i\}$ is associated with the edge e_i and we denote the vertices by $a(v, w_i)$ and $b(v, w_i)$. For every edge $e = \{v, w\} \in E(G)$ with $e \in T$ we add edges $\{a(v, w), b(w, v)\}$ and $\{b(v, w), a(w, v)\}$. If $e \notin T$ we add edges $\{a(v, w), a(w, v)\}$ and $\{b(v, w), b(w, v)\}$. The edges $e \in T$ are called *twisted* edges whereas edges $e \notin T$ are *non-twisted*.

Using the properties of the gadget X_3 described above one can now show that the isomorphism type of $\mathrm{CFI}(G,T)$ depends only on the parity of $|T|$. That is $\mathrm{CFI}(G,T) \cong \mathrm{CFI}(G,T')$ if and only if $|T| \equiv |T'| \mod 2$. We obtain a pair of non-isomorphic graphs $\mathrm{CFI}(G) := \mathrm{CFI}(G,\emptyset)$ and $\widetilde{\mathrm{CFI}}(G) := \mathrm{CFI}(G,\{e\})$ for some edge $e \in E(G)$.

For appropriate base graphs G (i.e., large tree width [7]) these graphs are difficult to distinguish by the Weisfeiler-Leman algorithms. This makes them candidates for being difficult instances for isomorphism testing. However, practical isomorphism solvers cope with CFI graphs reasonably well. The main reason for this is that the underlying algorithms take advantage of automorphisms of the input graphs to restrict their search space and CFI graphs typically have many automorphisms. Indeed, let $C = (v_1, \ldots, v_\ell)$ be a cycle in the base graph G. Then there is an automorphism of $\mathrm{CFI}(G)$ that exactly swaps those $a(v,w)$ and $b(v,w)$ for which $\{v,w\}$ is an edge in C and fixes all other $a(v,w)$ and $b(v,w)$ vertices. Thus, if ℓ is the dimension of the cycle space of G then $\mathrm{CFI}(G)$ has at least 2^ℓ automorphisms. In the next section we address this issue and describe a similar but probabilistic construction that gives with high probability graphs without non-trivial automorphisms.

3 A Rigid Base Construction

Desirable properties. Before we describe our construction we would like to give some intuition about the desirable properties that we want the final graph to have and why we think they are responsible for making the graphs difficult.

- (rigidity) There are some arguments that may make us believe that rigid graphs are more difficult for isomorphism solvers than graphs with automorphisms. Indeed, the graphs for which we can prove theoretical lower bounds [20] are rigid. Furthermore, it is possible to obtain upper bounds on the running time in terms of $|\Gamma|/|\operatorname{Aut}(G)|$ where Γ is a group known to contain all automorphisms (see [22] for a discussion). However, we can also offer an intuitive explanation. Isomorphism solvers are used in practice in search algorithms (e.g., SAT-solvers) to exploit symmetry and thereby cut off parts of the search tree. However, the isomorphism solvers themselves also exploit this strategy in bootstrapping manner, using symmetries to cut off parts of their own search tree.
- (small constants) For our practical purposes it is imperative to keep the graphs small. We thus need to diverge from the theoretical construction in [20]. As a crucial part of the construction we devise two methods to reduce the number of vertices while maintaining the difficulty level of the graph. These reductions are described in the next section.
- (simplicity) We strive to have a simple construction that is not only easy to understand but can also be quickly generated by a simple algorithm. In contrast, for many other graphs coming from combinatorial constructions it takes far longer to construct the graphs than to perform isomorphism tests. The simplicity also allows us to construct an abundance of benchmark graphs.
- (difficult for the WL algorithm) The Weisfeiler-Leman algorithms are a family of theoretical graph isomorphism algorithms. For a description and intuition as to why one may believe they can be used as a measure of the difficulty of a graph, we refer to [20].

3.1 The multipede construction

In the following we describe an explicit construction for families of similar, rigid, non-isomorphic graphs. This construction is based on the multipedes of Gurevich and Shelah [8] yielding finite rigid structures and uses the construction by Cai, Fürer and Immerman [5].

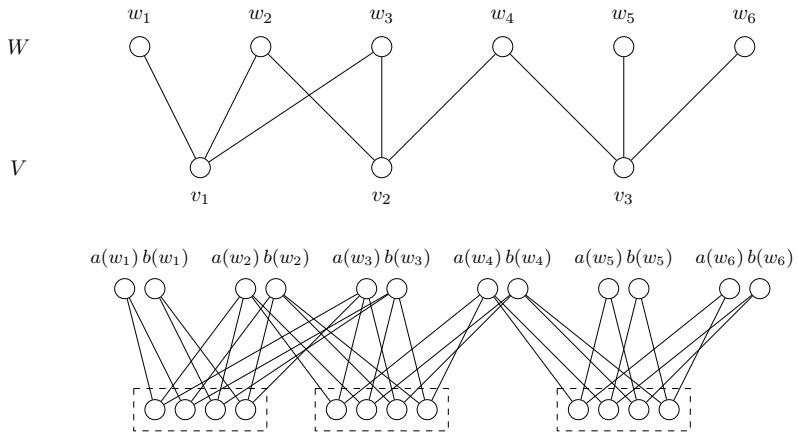


Figure 1 The figure depicts a base graph G on the top and the graph $R(G)$ obtained by applying the multipede construction on the bottom.

Let $G = (V, W, E)$ be a bipartite graph such that every vertex $v \in V$ has degree 3. Define the multipede construction as follows. We replace every $w \in W$ by two vertices $a(w)$ and $b(w)$. For $w \in W$ let $F(w) = \{a(w), b(w)\}$ and for $X \subseteq W$ let $F(X) = \bigcup_{w \in X} F(w)$. We then replace every $v \in V$ by a copy of X_3 and identify the vertices a_i and b_i with $a(w_i)$ and $b(w_i)$ where $w_1, \ldots, w_3$ are the neighbors of v. The resulting graph will be denoted by $R(G)$. An example of this construction is shown in Figure 1.

▶ **Definition 1.** Let $G = (V, W, E)$ be a bipartite graph. We say G is *odd* if for every $\emptyset \neq X \subseteq W$ there is a $v \in V$ such that $|X \cap N(v)|$ is odd.

For a graph G and a vertex $v \in V(G)$ we define the *second neighborhood* of v to be $N^2_G(v) = \{u \in V(G) \mid u \neq v \wedge \exists w \in V(G) \colon \{v, w\}, \{w, u\} \in E(G)\}$.

▶ **Lemma 2.** *Let $G = (V, W, E)$ be a bipartite graph, such that*
1. *$|N(v)| = 3$ for all $v \in V$,*
2. *G is odd,*
3. *G is rigid, and*
4. *there are no distinct $w_1, w_2 \in W$, such that $N^2_G(w_1) = N^2_G(w_2)$.*

Then $R(G)$ is rigid.

We now present a simple randomized algorithm to construct rigid bipartite graphs that are odd. Let G be a 3-regular graph and let $\sigma : E(G) \to E(G)$ be a random permutation of the edges of G. We define the bipartite graph $B(G, \sigma) = (V_B, W_B, E_B)$ by setting $V_B = V(G) \times \{0, 1\}$, $W_B = E(G)$, and $E_B = \{\{(v, 0), e\} \mid v \in e\} \cup \{\{(v, 1), e\} \mid v \in \sigma(e)\}$.

Thus, the edges of G correspond to the vertices in the partition class W_B of $B(G, \sigma)$. Each such edge $e = \{v, w\}$ has an associated edge $\sigma(e) = \{v', w'\}$ and e has exactly four neighbors, namely $(v, 0)$, $(w, 0)$, $(v', 1)$ and $(w', 1)$. Another way of visualizing this construction is to start with two copies of G, subdivide all edges in each copy and then identify the newly added vertices in the one copy with the newly added vertices in the other copy using a randomly chosen matching.

Let G_n be the *$2n$-cycle with diagonals added*. More precisely, $G_n := (V_n, E_n)$ where $V_n = \{1, \ldots, 2n\}$ and $E_n = \{\{i, i + 1 \bmod 2n\} \mid 1 \leq i \leq 2n\} \cup \{\{i, i + n\} \mid 1 \leq i \leq n\}$. We call the edges of the form $\{i, i + n\}$ *diagonals*.

In reference to [8] we call the graphs $R(B(G_n, \sigma))$ the multipede graphs. Observe that the multipede graphs can be constructed in linear time[1]. It can be computed that the graph $R(B(G_n, \sigma))$ has an average degree of $48/11 \approx 4.363$.

We can analyze this construction and show that for the base graphs G_n the resulting graphs $R(B(G_n, \sigma))$ are with high probability rigid. For this, building on Lemma 2, we show that with high probability the graph $B(G_n, \sigma)$ is odd, rigid, and has no distinct vertices in W_B with equal second neighborhoods.

▶ **Theorem 3.** *The probability that $R(B(G_n, \sigma))$ is not rigid is in $\mathcal{O}\left(\frac{\log^2 n}{n}\right)$.*

While it is difficult to extract hidden constants in the theorem, our experiments show that even for small n the probability that $R(B(G_n, \sigma))$ is not rigid is close to 0.

Creating pairs. The construction presented up to this point produces a single graph given n and σ, but instances to the graph isomorphism problem are pairs of graphs. Isomorphic pairs can of course always be obtained using two random permutations of one graph. For non-isomorphic pairs, following [5], to obtain the second graph, we twist one of the connections in one of the gadgets, that is, we switch the neighborhoods of the vertices $a(w)$ and $b(w)$ within one of the gadgets when creating $R(B(G_n, \sigma))$ from $B(G_n, \sigma)$. Arguments similar to those for Theorem 3 show that this creates with high probability pairs of non-isomorphic graphs. (This is not true anymore if the reduction from Subsection 4.1 is applied.)

4 The shrunken multipedes

As we described before we are interested in keeping the number of vertices of our construction small. In this section we describe two methods to reduce the number of vertices of the multipede graphs while, according to our experiments, essentially preserving the difficulty.

4.1 A linear algebra reduction

For a bipartite graph $G = (V, W, E)$ let $A_G \in \mathbb{F}_2^{V \times W}$ be the matrix with $A_{v,w} = 1$ if and only if $vw \in E(G)$ and let $\mathrm{rk}(A)$ be the $\mathbb{F}_2$-rank of A.

▶ **Lemma 4.** *A bipartite graph $G = (V, W, E)$ is odd if and only if $\mathrm{rk}(A_G) = |W|$.*

Proof. For the forward direction suppose $x \in \mathbb{F}_2^W$ such that $A_G x = 0$. Set $X = \{w \in W \mid x_w = 1\}$. Suppose towards a contradiction that $X \neq \emptyset$. Since G is odd there is some $v \in V$ such that $|N(v) \cap X|$ is odd. But then $(A_G)_v x = 1$ where $(A_G)_v$ is the v-th row of A_G. This is a contradiction and thus, $\{x \in \mathbb{F}_2^W \mid A_G x = 0\} = \{0\}$. So $\mathrm{rk}(A_G) = |W|$.

Suppose $\mathrm{rk}(A_G) = |W|$. Then $\{x \in \mathbb{F}_2^W \mid A_G x = 0\} = \{0\}$. Let $\emptyset \neq X \subseteq W$ and let $x \in \mathbb{F}_2^W$ be the vector with $x_w = 1$ if and only if $w \in X$. Since $x \notin \{x \in \mathbb{F}_2^W \mid A_G x = 0\}$ there is some $v \in V$ such that $(A_G)_v x = 1$. But this means that $|N(v) \cap X|$ is odd. ◀

▶ **Corollary 5.** *Let $G = (V, W, E)$ be an odd bipartite graph. Then there is some $V' \subseteq V$ with $|V'| \leq |W|$ such that the induced subgraph $G[V' \cup W]$ is odd.*

Using Gaussian elimination, we can compute such a set in polynomial time. Now suppose $B(G_n, \sigma)$ is odd. Then we can use the previous corollary to compute an induced subgraph $B^* := B^*(G_n, \sigma)$ of $B(G_n, \sigma)$ which is odd and has fewer vertices. Applying the rigid base

[1] Explicit pseudocode for constructing the multipede graphs is given in the full version [19].

construction to B^* the resulting graph has $|V(R(B^*))| = 4 \cdot 3 \cdot n + 2 \cdot 3 \cdot n = 18n$ vertices. The average degree has decreased to 4. In comparison, $|V(R(B(G_n, \sigma)))| = 4 \cdot 4 \cdot n + 2 \cdot 3 \cdot n = 22n$.

We want to stress at this point that the twisted and non-twisted version of $R(B^*)$ are indeed isomorphic. A simple trick to overcome this problem (but not investigated here) is to retain one more element of $V \setminus V'$ and to twist an edge of the respective additional gadget.

4.2 Bypassing the outer vertices

Consider the CFI-construction applied to a 3-regular base graph on n vertices. An easy trick is to identify the a and b vertices for each edge of the base graph instead of connecting them by an edge. More precisely, for each edge $e = \{v, w\}$ in the base graph, we can identify $a(v, w)$ with $a(w, v)$ and $b(v, w)$ with $b(w, v)$ in case of a non-twisted connection or identify $a(v, w)$ with $b(w, v)$ and $b(v, w)$ with $a(w, v)$ in case of a twisted connection. This way the number of vertices reduces from $10n$ to $7n$. This can further be improved by removing all a and b vertices altogether and connecting inner vertices $m_i(v)$ to $m_j(w)$ if both are connected to either $a(v, w)$ or $b(v, w)$. This way the construction only has $4n$ vertices.

A similar technique can also be applied to the rigid base construction. For this we bypass all vertices of degree 8 by directly joining their neighbors and then removing all such vertices. Given a bipartite base graph G we denote by $R^*(G)$ the graph obtained form $R(G)$ by bypassing the outer vertices. When reducing the graphs in that fashion one has to be aware that combinatorial structure of the graph might get lost. In particular it can be the case that vertices of degree 4 (in G) had the same neighborhood before they were removed, leading to inhomogeneity in the graph $R^*(G)$. In fact our experiments show that there is wider spread concerning the difficulty of the graphs $R^*(G)$ than the graphs $R(G)$. In other words, the rigidity and difficulty of those graphs depends more heavily on the choice of the matching σ than for the rigid base construction. However, the graphs $R^*(G)$ turn out to be more difficult than those obtained from the multipede construction.

4.3 Applying both reductions

We can combine the two vertex reduction techniques presented in this section by first applying Corollary 5 to the base graph and then bypassing the outer vertices. The number of vertices in the combined reduction decreases to $12n$. It is not difficult to see that the graphs have an average degree of at most 24, but the average degree varies among graphs on equally many vertices since there may be multiple bypass edges having the same endpoints.

Our experiments show that the combination of the two reductions (Section 6) yields the most difficult graphs. We call the resulting graphs $R^*(B^*(G_n, \sigma))$ the *shrunken multipedes*.

5 Cai-Fürer-Immerman gadgets for other groups

The constructions described in the previous sections revolve around the CFI-gadget X_3. On the outer vertices, the automorphism group of that gadget induces the set of all permutations that swap an even number of pairs. This corresponds to the subgroup of $(\mathbb{Z}_2)^3$ of elements (g_1, g_2, g_3) with $g_1 g_2 g_3 = 1$. A natural idea to push the CFI-construction to its limits is to encode other groups by the use of other gadgets.

Indeed, it is not difficult to see that for every permutation group Δ acting on a set Ω there is a graph gadget X_Δ with a vertex set containing Ω as $\mathrm{Aut}(X_\Delta)$-invariant subset such that $\mathrm{Aut}(X_\Delta) \cong \Delta$ and $\mathrm{Aut}(G)|_\Omega = \Delta$. (See [21, Lemma 16] for a construction.) In other words, every permutation group Δ can be *realized* by a graph gadget X_Δ. As explained

before, the original CFI-gadget realizes a group that is a subdirect product of $\mathbb{Z}_2^3$, which is in particular an abelian group. In analogy to our previous terminology we call the vertices in Ω the outer vertices and the vertices in $V(X_\Delta) \setminus \Omega$ the inner vertices. For our purpose of creating benchmark graphs, we are interested in keeping the number of inner vertices small.

In our constructions, Ω naturally decomposes into sets $C_1, C_2, C_3, \ldots, C_t$ of equal size n as the classes of the outer vertices. We think of the sets as being colored with different colors and only consider color preserving isomorphisms, i.e., permutations mapping each vertex to a vertex of the same color. If we insert a gadget X_Δ with $\Omega = C_1 \cup C_2 \cup \ldots C_t$ we obtain a subgroup of $\mathrm{Sym}(C_1) \times \mathrm{Sym}(C_2) \times \cdots \times \mathrm{Sym}(C_t)$, the direct product of symmetric groups.

For simplicity, we focus on the case $t = 3$ from now on. Thus we consider colored graph gadgets X_Δ such that $C_1 \cup C_2 \cup C_3 \subseteq V(X_\Delta)$ and for which the automorphism group $\mathrm{Aut}(X_\Delta)$ induces on $C_1 \cup C_2 \cup C_3$ a certain subgroup $\Delta \leq \mathrm{Sym}(C_1) \times \mathrm{Sym}(C_2) \times \mathrm{Sym}(C_3)$. Once we have a suitable gadget X_Δ we can use it to obtain a generalized CFI-construction (see [19]).

This brings us to the question, what types of gadgets are suitable for our purpose of creating hard benchmark graphs. Various results in the context of algorithmic group theory can lead one to believe that small groups and also abelian groups may be algorithmically easier than large or non-abelian groups ([4, 12, 24]). Thus, one may wonder whether more difficult benchmark graphs arise when employing graph gadgets inducing more complicated abelian or even non-abelian automorphism groups. We now explore these thoughts.

5.1 Examples for more general groups

General abelian groups. Let $\Gamma \leq S_n$ be an abelian permutation group on n elements. Then $\Delta := \{(a, b, c) \in \Gamma^3 \mid abc = 1\}$ is a subgroup of $S_n \times S_n \times S_n$. Here S_n denotes the symmetric group of the set $\{1, \ldots, n\}$. A description of a gadget realizing this group for the case in which Γ is transitive is described in the full version [19].

For $\Gamma = \mathbb{Z}_2$ we recover the CFI-gadget. Let us point out that for $\Gamma = \mathbb{Z}_k$ these gadgets were used in [23] to show hardness of graph isomorphism for certain complexity classes.

Applying the gadgets to suitable base graphs, this gives us a generalization of the rigid base construction to arbitrary abelian groups. However the difficulty of the graphs does not increase for the algorithms tested here as we will see in Section 6.

Since in various contexts abelian groups are algorithmically easier to handle than non-abelian groups, we would like to generalize the rigid base construction to non-abelian groups.

Semidirect products. Our next example is that of a semidirect product. Let $\Gamma = N \rtimes H \leq S_n$ be a semidirect product with an abelian normal subgroup N. For example Γ could be the dihedral group D_n. Then $\Delta := \{(ah, bh, ch) \in \Gamma^3 \mid a, b, c \in N,\ h \in H,\ abc = 1\}$ is a subgroup of $S_n \times S_n \times S_n$. Applying the gadget construction to suitable base graphs we obtain a family of new benchmark graphs that we call the *dihedral construction*.

Unentwined gadgets. As last example suppose $\Gamma_1 = H_2 \times H_3$, $\Gamma_2 = H_1 \times H_3$ and $\Gamma_3 = H_1 \times H_2$ with arbitrary finite groups H_i. Then the group $\Delta := \{((h_2, h_3), (h_1, h_3), (h_1, h_2))\}$ is a subgroup of $S_n \times S_n \times S_n$. We call these gadgets unentwined since they only create a pairwise interconnection between the classes. In particular, the gadgets are not beneficial for our cause since they do not force an interplay between all three classes C_1, C_2 and C_3.

5.2 Group theoretic restrictions on the gadgets

As discussed before, every group $\Delta \leq \mathrm{Sym}(C_1) \times \mathrm{Sym}(C_2) \times \mathrm{Sym}(C_3)$ can be realized by such a gadget X_Δ. Our intuition is that the isomorphism solvers can locally analyze a bounded size gadget and thus implicitly determine the automorphism group $\Delta = \mathrm{Aut}(X_\Delta)|_{C_1 \cup C_2 \cup C_3}$ of such a gadget. The goal must therefore be to encode the difficulty in the global dependency across the entire graph rather than in a local gadget.

Note that $\Delta \leq \pi_1(\Delta) \times \pi_2(\Delta) \times \pi_3(\Delta)$, where π_i is the projection on the i-th component. We say that Δ is a *subdirect product* of $\pi_1(\Delta) \times \pi_2(\Delta) \times \pi_3(\Delta)$. To understand what kind of gadgets can be constructed we should therefore investigate subdirect products. We call such a group $\Delta \leq \pi_1(\Delta) \times \pi_2(\Delta) \times \pi_3(\Delta)$ *2-factor injective* if each projection onto two of the factors is injective and *2-factor surjective* if each of these projections is surjective.

Intuitively, 2-factor injectivity says that two components determine the third. Thus when two different gadgets that are not 2-factor injective are attached (via parallel edges say) this may create local automorphism in the resulting graph, which we are trying to avoid. We are therefore interested in creating 2-factor injective gadgets. In fact, it is possible to restrict ourselves to 2-factor injective gadgets altogether, since we can quotient out the elements responsible for non-injectivity. It turns out however that the abelian example in the previous subsection are the only 2-factor injective and 2-factor surjective groups.

▶ **Lemma 6** ([18]). *Let $\Gamma = \Gamma_1 \times \Gamma_2 \times \Gamma_3$ be a finite group and Δ a subdirect product of Γ that is 2-factor surjective and 2-factor injective. Then Γ_1, Γ_2 and Γ_3 are isomorphic abelian groups and Δ is isomorphic to the subgroup of Γ_1^3 given by $\{(a, b, c) \in (\Gamma_1)^3 \mid abc = 1\}$, which in turn is isomorphic to $(\Gamma_1)^2$ as an abstract group.*

Thus, if we want to move beyond abelian groups, we cannot require 2-factor surjectivity. In general, however, it can be shown that every 2-factor injective group $\Delta \leq \pi_1(\Delta) \times \pi_2(\Delta) \times \pi_3(\Delta)$ is obtained by a combination of the three example constructions described in the previous subsection. Indeed, in [18] it is in particular shown that if one wants an entwined gadget then it is necessary to use an abelian group of the form described in Lemma 6. One can then take a finite extension similar to the semidirect product construction described above. We refer to [18] for more details. Overall we conclude that there are some group theoretic obstructions to using non-abelian gadgets, and are left with the intuition that the original construction leads to the most difficult examples.

6 Experimental Results

In the following we discuss experimental results for the constructions presented in this paper. The experiments were each performed on single node of a compute cluster with 2.00 GHz Intel Xeon X5675 processors. We always set a time limit of three hours (i.e., 10800 seconds) and the memory limit to 4 GB. Every single instance was processed once, but multiple instances were generated for each possible number of vertices with the same construction. We evaluated the following isomorphism solvers: `Bliss` version 0.72, `Nauty/Traces` version 25r9, `Saucy` version 3.0, and `Conauto` version 2.03.

All our instances consist of two graphs and the task for the algorithms was to check for isomorphism. For `Bliss` and `Nauty/Traces` this means that both graphs are canonized and the canonical forms are compared for equality. `Conauto` directly supports isomorphism testing and for `Saucy`, which only supports automorphism group computation, we compute the automorphism group of the disjoint union to check for isomorphism. In each series we performed the tests in increasing number of vertices with a time limit, which implied that once timeouts are reached repeatedly only a handful of further examples are computed.

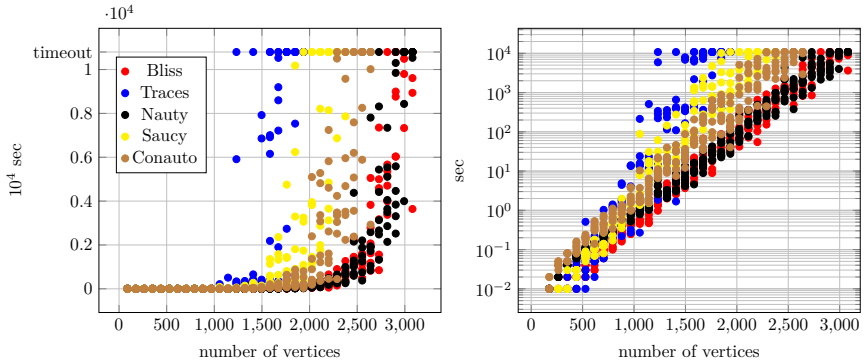


Figure 2 Performance of various algorithms on $R(B(G_n, \sigma))$ for random permutations σ in linear (left) and logarithmic scale (right).

We want to stress at this point that while the memory limit is irrelevant for `Bliss`, `Nauty`, `Saucy` and `Conauto`, it is significant for `Traces`. In fact, in many larger instances `Traces` reaches the memory limit before the time limit. For the sake of readability we do not distinguish between the two cases and in our figures runs that reached the memory limit are displayed as reaching the time limit. See [15] for details on the memory usage of `Traces`.

While all constructions pose difficult challenges for the solvers, those with more involved gadgets do not seem to yield more difficult examples. A reason for this effect could lie in the size of the gadgets. If the gadget is too large, asymptotic difficulty may emerge only for graphs with a number of vertices drastically larger than what could be tested. In any case, all of our constructions yield efficient methods to generate difficult isomorphism instances even with the size of the vertex set in regimes where no other difficult instances are available.

6.1 Shrunken Multipedes

Figure 2 shows the running times of the various isomorphism solvers on the multipede graphs described in Section 3 without any reductions. We observe that the multipede construction yields graphs which become infeasible for all solvers already for a few thousand vertices. Similarly, Figure 3 shows the running times on shrunken multipedes, i.e., the graphs to which the two node reductions of Section 4 have been applied. In comparison we observe that similar running times to the unreduced graphs are obtained already on graphs that have roughly half the number of vertices. While the shrunken version results in more difficult graphs one can also see that there is a significantly larger fluctuation (presumably depending on σ) among the graphs on a fixed number of vertices.

6.2 Comparison

We outline the main observations made in comparing our constructions with existing families of graphs. For a detailed comparison of the presented constructions among each other and to other families of difficult graphs we refer to the full version [19].

Reduction Techniques and other groups. We performed a series of experiments investigating the effect of the two reduction techniques from Section 4. Both reductions yield instances significantly more difficult and combining the two reductions yields the best results.

In experiments for the rigid base construction on other groups, we observe that the original construction based on $\mathbb{Z}_2$ yields the most difficult graphs.

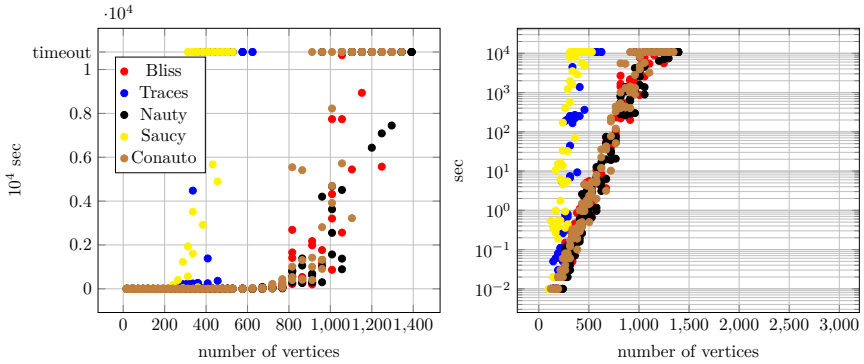


Figure 3 Performance of various algorithms on shrunken multipedes $R^*(B^*(G_n, \sigma))$ (both reductions applied) for random permutations σ in linear (left) and logarithmic scale (right).

Comparison with other benchmarks. Finally we compared the running times to benchmark graphs that previously existed. Since the existing benchmark graphs typically do not come in pairs of graphs we always take two copies, for each of which we randomly permute its vertices, and then perform an isomorphism test for the two copies.

We compared our graphs to the most difficult graphs from the library at [14]. We performed experiments for the families of Combinatorial graphs of Gordan Royle (combinatorial), Projective Planes of order 25 and 27 (pp), non-disjoint unions of tripartite graphs (tnn), Cai-Fürer-Immerman graphs (cfi), and Miyazaki graphs (mz-aug2). Starting from 1500 vertices our instances are several orders of magnitude more difficult on comparable input sizes.

7 Discussion

Variance. For each graph we tested each algorithm only once. Not all of the algorithms are deterministic and even for deterministic algorithms, there is also the question whether permuting the input graphs has any influence on the running times. We ran permutations of the same graph several times for each of the algorithms. Only saucy exhibits a non-negligible variance in some of the runs. However, even that variance is negligible in comparison to the exponential scaling of the running times on the benchmark graphs in the number of vertices.

Input size. The experiments we present all order the graphs according to the number of vertices. However, in practice one may be interested in sparse graphs, and the various algorithms are in particular tuned for this case. While all our graphs are sparse in that they only have a linear number of edges, average degrees vary. For shrunken multipedes we provided the coarse bound of 24 for the average degree. The multipedes have average degree 4.363, while the graphs obtained by only applying the linear algebra reduction ($R(B^*(G_n, \sigma))$) have average degree 4. It seems thus that when input size is measured in terms of the number of edges the graphs $R(B^*(G_n, \sigma))$ are the most difficult.

Conclusion. Overall we conclude that the new construction constitutes a simple algorithm that yields the most difficult benchmark graphs to date and experimentally we observe exponential behavior in terms of the running times of practical isomorphism solvers.

Acknowledgments. We thank Luis Núñez Chiroque, Tommi Junttila, Petteri Kaski, Brendan McKay, Adolfo Piperno, and José Luis López-Presa for numerous discussions related to graph isomorphism and for feedback on our benchmark instances.

References

1. László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In *STOC*, pages 684–697. ACM, 2016. doi:10.1145/2897518.2897542.
2. László Babai, Anuj Dawar, Pascal Schweitzer, and Jacobo Torán. The graph isomorphism problem (dagstuhl seminar 15511). *Dagstuhl Reports*, 5(12):1–17, 2015. doi:10.4230/DagRep.5.12.1.
3. László Babai, Paul Erdős, and Stanley M. Selkow. Random graph isomorphism. *SIAM Journal on Computing*, 9(3):628–635, 1980. doi:10.1137/0209047.
4. László Babai and Youming Qiao. Polynomial-time isomorphism test for groups with abelian sylow towers. In *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, February 29th – March 3rd, 2012, Paris, France*, volume 14 of *LIPIcs*, pages 453–464. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.STACS.2012.453.
5. Jin-yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identifications. *Combinatorica*, 12(4):389–410, 1992. doi:10.1007/BF01305232.
6. Paolo Codenotti, Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Conflict analysis and branching heuristics in the search for graph automorphisms. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*, pages 907–914. IEEE Computer Society, 2013. doi:10.1109/ICTAI.2013.139.
7. Anuj Dawar and David Richerby. The power of counting logics on restricted classes of finite structures. In *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2007. doi:10.1007/978-3-540-74915-8_10.
8. Yuri Gurevich and Saharon Shelah. On finite rigid structures. *J. Symb. Log.*, 61(2):549–562, 1996. doi:10.2307/2275675.
9. Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, pages 135–149. SIAM, 2007. doi:10.1137/1.9781611972870.13.
10. Ludek Kucera. Canonical labeling of regular graphs in linear average time. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 271–279. IEEE Computer Society, 1987. doi:10.1109/SFCS.1987.11.
11. José Luis López-Presa, Antonio Fernández Anta, and Luis Núñez Chiroque. Conauto-2.0: Fast isomorphism testing and automorphism group computation. *CoRR*, abs/1108.1060, 2011. URL: https://arxiv.org/abs/1108.1060.
12. Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. Syst. Sci.*, 25(1):42–65, 1982. doi:10.1016/0022-0000(82)90009-5.
13. Rudolf Mathon. Sample graphs for isomorphism testing. In *Proceedings of the ninth Southeastern Conference on Combinatorics, Graph Theory and Computing: Florida Atlanta University, Boca Raton, January 30-February 2, 1978*, Congressus Numerantium, 21. Winnipeg: Utilitas Mathematica Publish, 1978.
14. Brendan D. McKay and Adolfo Piperno. nautytraces software distribution web page. http://cs.anu.edu.au/~bdm/nauty/ and http://pallini.di.uniroma1.it.
15. Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014. doi:10.1016/j.jsc.2013.09.003.

16 Takunari Miyazaki. The complexity of mckay's canonical labeling algorithm. In *Groups and Computation*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 239–256. DIMACS/AMS, 1995.

17 Daniel Neuen and Pascal Schweitzer. Benchmark graphs for practical graph isomorphism. `https://www.lii.rwth-aachen.de/research/95-benchmarks.html`.

18 Daniel Neuen and Pascal Schweitzer. Subgroups of 3-factor direct products, 2016. arXiv:1607.03444. URL: `https://arxiv.org/abs/1607.03444`.

19 Daniel Neuen and Pascal Schweitzer. Benchmark graphs for practical graph isomorphism. *CoRR*, abs/1705.03686, 2017. URL: `http://arxiv.org/abs/1705.03686`.

20 Daniel Neuen and Pascal Schweitzer. An exponential lower bound for individualization-refinement algorithms for graph isomorphism. *CoRR*, abs/1705.03283, 2017. URL: `http://arxiv.org/abs/1705.03283`.

21 Yota Otachi and Pascal Schweitzer. Reduction techniques for graph isomorphism in the context of width parameters. In *SWAT*, volume 8503 of *Lecture Notes in Computer Science*, pages 368–379. Springer, 2014. `doi:10.1007/978-3-319-08404-6_32`.

22 Pascal Schweitzer. *Problems of unknown complexity: graph isomorphism and Ramsey theoretic numbers*. Phd. thesis, Universität des Saarlandes, Saarbrücken, Germany, 2009.

23 Jacobo Torán. On the hardness of graph isomorphism. *SIAM J. Comput.*, 33(5):1093–1108, 2004. `doi:10.1137/S009753970241096X`.

24 Faried Abu Zaid, Erich Grädel, Martin Grohe, and Wied Pakusa. Choiceless polynomial time on structures with small abelian colour classes. In *Mathematical Foundations of Computer Science 2014 – 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*, volume 8634 of *Lecture Notes in Computer Science*, pages 50–62. Springer, 2014. `doi:10.1007/978-3-662-44522-8_5`.

On the Tree Augmentation Problem

Zeev Nutov

The Open University of Israel, Ra'anana, Israel
nutov@openu.ac.il

Abstract

In the Tree Augmentation problem we are given a tree $T = (V, F)$ and a set $E \subseteq V \times V$ of edges with positive integer costs $\{c_e : e \in E\}$. The goal is to augment T by a minimum cost edge set $J \subseteq E$ such that $T \cup J$ is 2-edge-connected. We obtain the following results.

- Recently, Adjiashvili [SODA 17] introduced a novel LP for the problem and used it to break the 2-approximation barrier for instances when the maximum cost M of an edge in E is bounded by a constant; his algorithm computes a $1.96418 + \epsilon$ approximate solution in time $n^{(M/\epsilon^2)^{O(1)}}$. Using a simpler LP, we achieve ratio $\frac{12}{7} + \epsilon$ in time $2^{O(M/\epsilon^2)}$. This also gives ratio better than 2 for logarithmic costs, and not only for constant costs. In addition, we will show that (for arbitrary costs) the problem admits ratio 3/2 for trees of diameter ≤ 7.
- One of the oldest open questions for the problem is whether for unit costs (when $M = 1$) the standard LP-relaxation, so called Cut-LP, has integrality gap less than 2. We resolve this open question by proving that for unit costs the integrality gap of the Cut-LP is at most $28/15 = 2 - 2/15$. In addition, we will suggest another natural LP-relaxation that is much simpler than the ones in previous work, and prove that it has integrality gap at most 7/4.

1998 ACM Subject Classification G.2.2 [Graph Theory] Graph Algorithms

Keywords and phrases Tree augmentation, Logarithmic costs, Approximation algorithm, Half-integral extreme points, Integrality gap

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.61

1 Introduction

We consider the following problem:

Tree Augmentation
Input: A tree $T = (V, F)$ and an additional set $E \subseteq V \times V$ of edges with positive integer costs $c = \{c_e : e \in E\}$.
Output: A minimum cost edge set $J \subseteq E$ such that $T \cup J$ is 2-edge-connected.

The problem was studied extensively, cf. [11, 15, 3, 21, 8, 9, 4, 19, 5, 17, 2, 16]. For a long time the best known ratio for the problem was 2 for arbitrary costs [11] and 1.5 for unit costs [8, 17]; see also [9] for a simple 1.8-approximation algorithm. It is also known that the integrality gap of a standard LP-relaxation for the problem, so called Cut-LP, is at most 2 [11] and at least 1.5 [4]. Several other LP and SDP relaxations were introduced to show that the algorithm in [8, 9, 17] achieves ratio better than 2 w.r.t. these relaxations, cf. [2, 16]. For additional algorithms with ratio better than 2 for restricted versions see [5, 19].

Let M denote the maximum cost of an edge in E. Recently, Adjiashvili [1] introduced a novel LP for the problem – so called the k-Bundle-LP, and used it to break the natural 2-approximation barrier for instances when M is bounded by a constant. To introduce this result we need some definitions.

The edges of T will be called **T-edges** to distinguish them from the edges in E. Tree Augmentation can be formulated as a problem of covering the T-edges by paths. Let T_{uv}

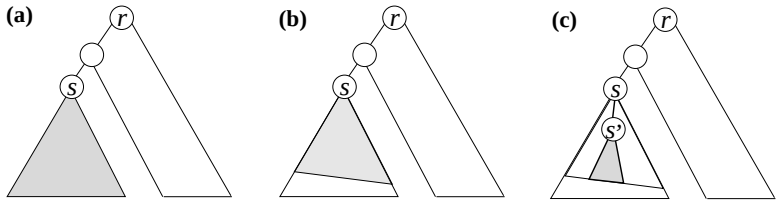


Figure 1 (a) complete rooted subtree; (b) full rooted subtree; (c) branch of a full rooted subtree.

denote the unique uv-path in T. We say that an **edge** uv **covers a** T**-edge** f if $f \in T_{uv}$. Then $T \cup J$ is 2-edge-connected if and only if J covers T. For a set $B \subseteq F$ of T-edges let $\psi(B)$ denote the set of edges in E that cover some $f \in B$, and $\tau(B)$ the minimum cost of an edge set in E that covers B. For $J \subseteq E$ let $x(J) = \sum_{e \in J} x_e$. The standard LP for the problem which we call the Cut-LP seeks to minimize $c^{\mathsf{T}}x = \sum_{e \in E} c_e x_e$ over the Cut-Polyhedron

$$\Pi^{Cut} = \left\{ x \in \mathbb{R}^E : x(\psi(f)) \geq 1 \ \forall f \in F, x \geq 0 \right\} .$$

The k-Bundle-LP of Adjiashvili [1] adds over the standard Cut-LP the constraints $\sum_{e \in \psi(B)} c_e x_e \geq \tau(B)$ for any forest B in T that has at most k leaves, where $k = \Theta(M/\epsilon^2)$. The algorithm of [1] computes a $1.96418 + \epsilon$ approximate solution w.r.t. the k-Bundle-LP in time $n^{k^{O(1)}}$. For unit costs, a modification of the algorithm achieves ratio $5/3 + \epsilon$.

Here we observe that it is sufficient to consider just certain subtrees of T instead of forests. Root T at some node r. The choice of r defines an ancestor/descendant relation on V. The **leaves of** T are the nodes in $V \setminus \{r\}$ that have no descendants. For any subtree S of T, the node s of S closest to r is the root of S, and the pair S, s is called a **rooted subtree** of T, r; we will not mention the roots of trees if they are clear from the context. We say that S is a **complete rooted subtree** if it contains all descendants of s in T, and a **full rooted subtree** if for any non-leaf node v of S the children of v in S and T coincide; see Fig. 1(a,b). A **branch of** S, or a **branch hanging on** s, is a rooted subtree B of S induced by the root s of S and the descendants in S of some child s' of s; see Fig. 1 (c). We say that a subtree B of T is a **branch** if it is a branch of a full rooted subtree, or if it is a full rooted subtree with root r. Equivalently, a branch is a union of a full rooted subtree and its parent T-edge.

Let $\mathcal{B}_k$ denote the set of all branches in T with less than k leaves. The k-Branch-LP seeks to minimize $c^{\mathsf{T}}x = \sum_{e \in E} c_e x_e$ over the k-Branch-Polyhedron $\Pi_k^{Br} \subseteq \mathbb{R}^E$ defined by the constraints:

$$\begin{aligned}
\sum_{e \in \psi(f)} x_e &\geq 1 && \forall f \in F , \\
\sum_{e \in \psi(B)} c_e x_e &\geq \tau(B) && \forall B \in \mathcal{B}_k , \\
x_e &\geq 0 && \forall e \in E .
\end{aligned}$$

The set of constrains of the k-Branch-LP is a subset of constraints of the k-Bundle-LP of Adjiashvili [1], hence the k-Branch-LP is both more compact and its optimal value is no larger than that of the k-Bundle-LP. The first main result in this paper is:

▶ **Theorem 1.** *For any* $1 \leq \lambda \leq k-1$, Tree Augmentation *admits a* $4^k \cdot poly(n)$ *time algorithm that computes a solution of cost at most* $\rho + \frac{8}{3}\frac{\lambda M}{k - \lambda M} + \frac{2}{\lambda}$ *times the optimal value of the* k-Branch-LP*, where* $\rho = \frac{12}{7}$ *for arbitrary costs and* $\rho = 1.6$ *for unit costs.*

For a given ϵ, choosing properly $\lambda = \Theta(1/\epsilon)$ and $k = \Theta(M/\epsilon^2)$ gives ratio $\rho + \epsilon$ in time $2^{O(M/\epsilon^2)} \cdot poly(n)$.

We note that in parallel to our work Fiorini, Groß, Könemann, and Sanitá [10] augmented the k-Bundle LP of [1] by additional constraints – $\{0, \frac{1}{2}\}$-Chvátal-Gomory Cuts, to achieve ratio $1.5 + \epsilon$ in $n^{(M/\epsilon^2)^{O(1)}}$ time, thus almost matching the best known ratio for unit costs [17]. Our result in Theorem 1, done independently, shows that already the k-Bundle LP has integrality gap closer to 1.5 than to 2. Our version of the algorithm of [1] is also simpler than the one in [10]. In fact, combining our approach with [10] enables to achieve ratio $1.5 + \epsilon$ in $2^{O(M/\epsilon^2)} \cdot poly(n)$ time. Note that this allows to achieve this ratio for logarithmic costs, and not only for constant costs.

Let $\mathsf{diam}(T)$ denote the diameter of T. Tree Augmentation admits a polynomial time algorithm when $\mathsf{diam}(T) \leq 3$. If $\mathsf{diam}(T) = 2$ then T is a star and we get the Edge-Cover problem, while the case $\mathsf{diam}(T) = 3$ is reduced to the case $\mathsf{diam}(T) = 2$ by "guessing" some optimal solution edge that covers the central T-edge. The problem is NP-hard when $\mathsf{diam}(T) = 4$ even for unit costs [11]. We prove that for arbitrary costs Tree Augmentation with trees of diameter ≤ 7 admits ratio 3/2.

Our second main result resolves one of the oldest open questions concerning the problem – whether for unit costs the integrality gap of the Cut-LP is less than 2. This was conjectured in the 90's by Cheriyan, Jordán & Ravi [3] for arbitrary costs, but so far there was no real evidence for this even for unit costs. Our second main result resolves this old open question.

▶ **Theorem 2.** *For unit costs, the integrality gap of the* Cut-LP *is at most* $28/15 = 2 - 2/15$.

In addition, we will suggest another natural LP-relaxation that is much simpler than the ones in previous work, and prove that for unit costs it has integrality gap at most 7/4.

2 The algorithm (Theorem 1)

The Theorem 1 algorithm is a modification of the algorithm of Adjiashvili [1]. We emphasize some differences. We use the k-Branch-LP instead of the k-Bundle-LP of [1]. But, unlike [1], we do not solve our LP at the beginning. Instead, we combine binary search with the ellipsoid algorithm as follows. We start with lower and upper bounds p and q on the value of the k-Branch-LP, e.g., $p = 0$ and q is the cost of some feasible solution to the problem. Given a "candidate" x with $q \leq c^{\mathsf{T}}x \leq p$, the outer iteration (see Algorithm 1) of the entire algorithm either returns a solution of cost at most $(\rho + \frac{8}{3}\frac{\lambda M}{k-\lambda M} + \frac{2}{\lambda})c^{\mathsf{T}}x$ or a constraint of the k-Branch-LP violated by x; we show that this can be done in time $4^k \cdot poly(n)$, rather than in time $n^{k^{O(1)}}$ as in [1]. We set $p \leftarrow \frac{p+q}{2}$ in the former case and $q \leftarrow \frac{p+q}{2}$ in the latter case and continue to the next iteration, terminating when $p - q$ is small enough. This essentially gives a $4^k \cdot poly(n)$ time separation oracle for the k-Branch-LP (if a violated k-branch constraint is found). Since the ellipsoid algorithm uses a polynomial number of calls to the separation oracle, the running time is $4^k \cdot poly(n)$. Note that checking whether $x \in \Pi^{Cut}$ is trivial, hence for simplicity of exposition we will assume that the "candidate" x is in Π^{Cut}.

For a set S of T-edges we denote by T/S the tree obtained from T by contracting every T-edge of S. This defines a new Tree Augmentation instance, where contraction of a T-edge uv leads to shrinking u, v into a single node in the graph (V, E) of edges. In the algorithm, we repeatedly take a certain complete rooted subtree $\hat{S}$, and either find a k-branch-constraint violated by some branch in $\hat{S}$, or a "cheap" cover J_S of a subset S of the T-edges of $\hat{S}$; in the latter case, we add J_S to our partial solution J, contract $\hat{S}$, and iterate on the instance $T \leftarrow T/\hat{S}$. At the end of the loop, the edges that are still not covered by the partial solution J are covered by a different procedure, by a total cost $\frac{2}{\lambda} \cdot c^{\mathsf{T}}x$, as follows.

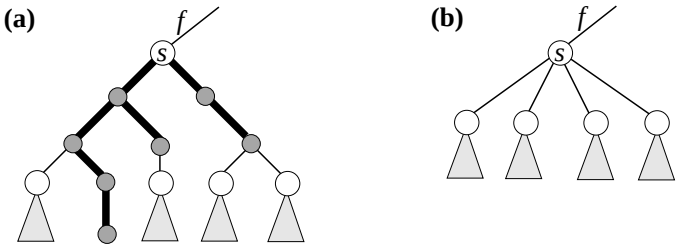


Figure 2 Branches hanging on s after contracting S'; λ-thick T-edges are shown by thick lines.

We call a T-edge $f \in F$ **λ-thin** if $x(\psi(f)) \leq \lambda$, and f is **λ-thick** otherwise. We need the following lemma from [1], for which we provide a proof for completeness of exposition.

▶ **Lemma 3** ([1]). *There exists a polynomial time algorithm that given $x \in \Pi^{Cut}$, $\lambda > 1$, and a set $F' \subseteq F$ of λ-thick T-edges computes a cover J' of F' of cost $\leq \frac{2}{\lambda} \cdot c^{\mathsf{T}} x$.*

Proof. Since all T-edges in F' are λ-thick, x/λ is a feasible solution to the Cut-LP for covering F'. Thus any polynomial time algorithm that computes a solution J' of cost at most 2 times the optimal value of the Cut-LP for covering F' has the desired property. There are several such algorithms, see [11, 12, 14]. ◀

We say that a complete rooted subtree S of T is a **(k, λ)-subtree** if S has at least k leaves and if either the parent T-edge f of S is λ-thin or $s = r$. For $\lambda = \Theta(1/\epsilon)$ and $k = \Theta(M/\epsilon^2)$ we choose $\hat{S}$ to be *an inclusionwise minimal (k, λ)-subtree*. Let us focus on the problem of covering such $\hat{S}$. Let S' be the set of T-edges of the inclusionwise maximal subtree of $\hat{S}$ that contains the root s of $\hat{S}$ and has only λ-thick T-edges (possibly $S = \emptyset$); see Fig. 2(a). We postpone covering the T-edges in S' to the end of the algorithm, so we contract S' into s and consider the tree $S \leftarrow \hat{S}/S'$; see Fig. 2(b). In S, every branch B hanging on s has less than k leaves, by the minimality of S, hence it has a corresponding constraint in the k-Branch-LP. We will show that for a k-branch B an optimal set of edges that covers B can be computed in time $4^k \cdot poly(n)$. If $\sum_{e \in \psi(B)} c_e x_e < \tau(B)$ for some branch B hanging on s in S, then we return the corresponding k-branch constraint violated by x; otherwise, we will show how to compute a "cheap" cover of S. More formally, in the next section we will prove:

▶ **Lemma 4.** *Suppose that we are given an instance of* Tree Augmentation *and $x \in \Pi^{Cut}$ such that any complete rooted proper subtree of the input tree has less than k leaves. Then there exists a $4^k \cdot poly(n)$ time algorithm that either finds a k-branch constraint violated by x, or computes a solution of cost $\leq \rho \sum_{e \in E \setminus R} c_e x_e + \frac{4}{3} \sum_{e \in R} c_e x_e$, where ρ is as in Theorem 1 and R is the set of edges in E incident to the root.*

To find a cheap covers of S, we consider the Tree Augmentation instance obtained from T/S' by contacting into s all nodes not in S. Note that every edge that was in $\psi(S) \cap \psi(f)$ is now incident to the root. Thus since $\rho \geq \frac{4}{3}$, Lemma 4 implies:

▶ **Corollary 5.** *There exists a $4^k \cdot poly(n)$ time algorithm that either finds a k-branch-constraint violated by x, or a cover J_S of S of cost $c(J_S) \leq \rho \sum_{e \in \gamma(S)} c_e x_e + \frac{4}{3} \sum_{e \in \psi(f)} c_e x_e$, where ρ is as in Theorem 1 and $\gamma(S)$ denotes the set of edges with both endnodes in S.*

The outer iteration of the algorithm is as follows:

Algorithm 1: OUTER-ITERATION$(T=(V,F),E,x,c,k,r,\lambda)$

1 $J \leftarrow \emptyset$, $F' \leftarrow \emptyset$
2 **while do**
3 $\quad$ T has at least 2 nodes
4 let $\hat{S}$ be an inclusionwise minimal (k,λ)-subtree of T
5 let S' be the edge-set of the inclusionwise maximal subtree of $\hat{S}$ that contains the root s of $\hat{S}$ and has only λ-thick edges
6 apply the algorithm from **Corollary 5** on $S \leftarrow \hat{S}/S'$
7 if the algorithm returns a cover J_S of S do: $F' \leftarrow F' \cup S'$, $J \leftarrow J \cup J_S$, $T \leftarrow T/\hat{S}$
8 else, return a k-branch constraint violated by x and STOP compute a cover J' of F' of cost $c(J') \leq \frac{2}{\lambda} \cdot c^{\mathsf{T}}x$ using the algorithm from **Lemma 3**
9 **return** $J \cup J'$

Note that at step 7 the T-edges in F' are all λ-thick and thus Lemma 3 applies. We will now analyze the performance of the algorithm assuming than no k-branch-constraint violated by x was found. Let $\delta(S)$ denote the set of edges with exactly one endnode in S and $\gamma(S)$ the set of edges with both endnodes in S. Let f be the parent T-edge of S. Since f is λ-thin

$$\sum_{e\in\psi(f)} c_e x_e \leq \sum_{e\in\psi(f)} M x_e \leq M \cdot x(\psi(f)) \leq M\lambda \ .$$

Since $x(\delta(v)) \geq 1$ for every leaf v of S, $c_e \geq 1$ for every $e \in E$, and since S is a (k,λ)-subtree

$$2\sum_{e\in\gamma(\hat{S})} c_e x_e = \sum_{v\in\hat{S}} \sum_{e\in\delta(v)} c_e x_e - \sum_{e\in\psi(f)} c_e x_e \geq \sum_{v\in\hat{S}\setminus\{s\}} x(\delta(v)) - \lambda M \geq k - \lambda M \ .$$

Consider a single iteration in the while-loop. Let $\Delta(c^{\mathsf{T}}x)$ denote the decrease in the LP-solution value as a result of contracting $\hat{S}$. Then

$$\Delta(c^{\mathsf{T}}x) = \sum_{e\in\gamma(\hat{S})} c_e x_e \geq \frac{k-\lambda M}{2} \ .$$

On the other hand, by Lemma 4, the partial solution cost increase is bounded by

$$c(J_S) \leq \rho \sum_{e\in\gamma(S)} c_e x_e + \frac{4}{3} \sum_{e\in\psi(f)} c_e x_e \leq \rho \sum_{e\in\gamma(\hat{S})} c_e x_e + \frac{4}{3}\lambda M \ .$$

Thus

$$\frac{c(J_S)}{\Delta(c^{\mathsf{T}}x)} \leq \rho + \frac{8}{3}\frac{\lambda M}{k-\lambda M} \ .$$

The while-loop terminates when the LP-solution value becomes 0, hence by a standard local-ratio/induction argument we get that at the end of the while-loop $c(J) \leq \left(\rho + \frac{8}{3}\frac{\lambda M}{k-\lambda M}\right) c^{\mathsf{T}}x$. At step 7 we add an edge set of cost $\leq \frac{2}{\lambda}c^{\mathsf{T}}x$, and Theorem 1 follows. It only remains to prove Lemma 4, which we will do in the subsequent sections.

2.1 Proof of Lemma 4

Assume that we are given an instance $T=(V,F),E,c,r$ of TREE AUGMENTATION and x as in Lemma 4. It is known that TREE AUGMENTATION instances when T is a path can be solved in polynomial time. This allows us to assume that the graph (V,E) is a complete graph and that $c_{uv} = \tau(T_{uv})$ for all $u,v \in V$. Note that we use this assumption only in the proof of Lemma 4, where the running time does not depend on the maximum cost M of an edge in E. Let us say that an edge $uv \in E$ is:

- a **cross-edge** if r is an internal node of T_{uv};
- an **in-edge** if r does not belong to T_{uv};
- an **r-edge** if $r = u$ or $r = v$;
- an **up-edge** if one of u, v is an ancestor of the other.

For a subset $E' \subseteq E$ of edges the **E'-up vector of** x is obtained from x as follows: for every non-up edge $e = uv \in E'$ increase x_{ua} and x_{va} by x_e and then reset x_e to 0, where a is the least common ancestor of u and v. The **fractional cost** of a set J of edges w.r.t. c and x is defined by $\sum_{e \in J} c_e x_e$. Let C^{in}_x, C^{cr}_x, and C^r_x denote the fractional cost of in-edges, cross-edges, and r-edges, respectively, w.r.t. c and x. We fix some $x^* \in \Pi^{Cut}$ and denote by C^{in}, C^{cr}, and C^r the fractional cost of in-edges, cross-edges, and r-edges, respectively, w.r.t. c and x^*. We give two rounding procedures, given in Lemmas 6 and 7. The rounding procedure in Lemma 6 is similar to that of Adjiashvili [1], for which we present an improved running time analysis.

▶ **Lemma 6.** *There exists a $4^k \cdot poly(n)$ time algorithm that either finds a k-branch inequality violated by x^*, or returns an integral solution of cost at most $C^{\mathsf{in}} + 2C^{\mathsf{cr}} + C^r$.*

Proof. Let $\mathcal{B}$ be the set of branches hanging on r. For every $B \in \mathcal{B}$ compute an optimal solution J_B. If for some $B \in \mathcal{B}$ we have $\tau(B) > \sum_{e \in \psi(B)} c_e x^*_e$ then a k-branch inequality violated by x^* is found. Else, the algorithm returns the union $J = \bigcup_{B \in \mathcal{B}} J_B$ of the computed edge sets. As every cross-edge has its endnodes in two distinct branches, while every in-edge or r-edge has its both endnodes in the same branch, we get

$$c(J) \leq \sum_{B \in \mathcal{B}} \tau(B) \leq \sum_{B \in \mathcal{B}} \sum_{e \in \psi(B)} c_e x^*_e = \sum_{B \in \mathcal{B}} \left(\sum_{e \in \delta(B)} c_e x^*_e + \sum_{e \in \gamma(B)} c_e x^*_e \right) = 2C^{\mathsf{cr}} + C^{\mathsf{in}} + C^r \ .$$

It remains to show that an optimal solution in each branch of r can be computed in time $4^k \cdot poly(n)$. More generally, we will show that Tree Augmentation instances with k leaves can be solved optimally within this time bound. Recall that we may assume that the graph (V, E) is a complete graph and that $c_{uv} = \tau(T_{uv})$ for all $u, v \in V$. We claim that then we can assume that T has no node v with $\deg_T(v) = 2$. This is a well known reduction, cf. [20]. In more details, we show that any solution J can be converted into a solution of no greater cost that has no edge incident to v, and thus v can be "shortcut". If J has edges uv, vw then it is easy to see that $(J \setminus \{uv, vw\}) \cup \{uw\}$ is also a feasible solution, of cost at most $c(J)$, since $c_{uw} \leq c_{uv} + c_{vw}$. Applying this operation repeatedly we may assume that $\deg_J(v) \leq 1$. If $\deg_J(v) = 0$, we are done. Suppose that J has a unique edge $e = vw$ incident to v. Let vu and vu' be the two T-edges incident to v, where assume that vu' is not covered by e. Then there is an edge $e' \in J$ that covers vu'. Since e' is not incident to v, it must be that e' covers vu. Replacing e by the edge wu gives a feasible solution without increasing the cost.

Consequently, we reduce our instance to an equivalent instance with at most $2k - 1$ tree edges. Now recall that Tree Augmentation is a particular case of the Min-Cost Set-Cover problem, where the set F of T-edges are the elements and $\{T_e : e \in E\}$ are the sets. The Min-Cost Set-Cover problem can be solved in $2^n \cdot poly(n)$ time via dynamic programming, where n is the number of elements; such an algorithm is described in [7, Sect. 6.1] for unit costs, but the proof extends to arbitrary costs [6]. Thus our reduced Tree Augmentation instance can be solved in $2^{2k-1} \cdot poly(n) \leq 4^k \cdot poly(n)$ time. ◀

For the second rounding procedure Adjiashvili [1] proved that for any $\lambda > 1$ one can compute in polynomial time an integral solution of cost at most $2\lambda C^{\mathsf{in}} + \frac{4}{3} \frac{\lambda}{\lambda - 1} C^{\mathsf{cr}}$. We prove:

▶ **Lemma 7.** *There exists a polynomial time algorithm that computes a solution of cost* $\frac{4}{3}(2C^{\mathsf{in}} + C^{\mathsf{cr}} + C^r)$*, and a solution of size* $2C^{\mathsf{in}} + \frac{4}{3}C^{\mathsf{cr}} + C^r$ *in the case of unit costs.*

Consider the case of arbitrary bounded costs. If $C^{\mathsf{in}} \geq \frac{2}{5}C^{\mathsf{cr}}$ we use the rounding procedure from Lemma 6 and the rounding procedure from Lemma 7 otherwise. In both cases we get $c(J) \leq \frac{12}{7}(C^{\mathsf{in}} + C^{\mathsf{cr}}) + \frac{4}{3}C^r$. In the case of unit costs, if $C^{\mathsf{in}} \geq \frac{2}{3}C^{\mathsf{cr}}$ we use the rounding procedure from Lemma 6, and the procedure from Lemma 7 otherwise. In both cases we get $c(J) \leq 1.6(C^{\mathsf{in}} + C^{\mathsf{cr}}) + C^r$.

Lemma 7 is proved in the next section. The proof relies on properties of extreme points of the Cut-Polyhedron Π^{Cut} that are of independent interest.

2.2 Properties of extreme points of the Cut-Polyhedron (Lemma 7)

W.l.o.g., we augment the Cut-LP by the constraints $x_e \leq 1$ for all $e \in E$, while using the same notation as before. Then (the modified) Cut-LP always has an optimal solution x that is an **extreme point** or a **basic feasible solution** of Π^{Cut}. Geometrically, this means that x is not a convex combination of other points in Π^{Cut}; algebraically this means that there exists a set of $|E|$ inequalities in the system defining Π^{Cut} such that x is the unique solution for the corresponding linear equations system. These definitions are known to be equivalent and we will use both of them, cf. [18].

A set family $\mathcal{L}$ is **laminar** if any two sets in the family are either disjoint or one contains the other. Note that Tree Augmentation is equivalent to the problem of covering the laminar family of the node sets of the complete rooted proper subtrees of T, where an edge covers a node set S if it has exactly one endnode in S. In particular, note that the constraint $\sum_{e\in\psi(f)} x_e \geq 1$ is equivalent to the constraint $x(\delta(S)) \geq 1$ where S is the node set of the complete rooted subtree with parent T-edge f.

▶ **Lemma 8.** *Let* (V, E) *be a graph,* $\mathcal{L}$ *a laminar family on* V*, and* $b \in \mathbb{N}^{\mathcal{L}}$*. Suppose that for every* $A \in \mathcal{L}$ *there is no edge between two distinct children of* S *and that the equation system* $\{x(\delta(S)) = b_S : S \in \mathcal{L}\}$ *has a unique solution* $0 < x^* < 1$*. Then* $x^*_e = 1/2$ *for all* $e \in E$*. Furthermore, each endnode of every* $e \in E$ *belongs to some* $S \in \mathcal{L}$*.*

Proof. For every $uv \in E$ put one token at u and one token at v. The total number of tokens is $2|E|$. For $S \in \mathcal{L}$ let $t(S)$ be the number of tokens placed at nodes in S that belong to no child of S. Since $\mathcal{L}$ is laminar, every token is placed in at most one set in $\mathcal{L}$, and thus $\sum_{S\in\mathcal{L}} t(S) \leq 2|E|$. Let $S \in \mathcal{L}$ and let $\mathcal{C}(S)$ be the set of children of S in $\mathcal{L}$. Let E_S be the set of edges in $\delta(S)$ that cover no child of S, and $E_{\mathcal{C}(S)}$ the set of edges not in $\delta(S)$ that cover some child of S. Note that no $e \in E_{\mathcal{C}(S)}$ connects two distinct children of S. Observe that

$$x^*(E_S) - x^*(E_{\mathcal{C}(S)}) = x^*(\delta(S)) - \sum_{C\in\mathcal{C}(S)} x^*(\delta(C)) = b_S - \sum_{C\in\mathcal{C}(A)} b_C \equiv b'_S \ .$$

Thus $x^*(E_S) - x^*(E_{\mathcal{C}(S)})$ is an integer. We cannot have $|E_S| = |E_{\mathcal{C}(S)}| = 0$ by linear independence, and we cannot have $|E_S| + |E_{\mathcal{C}(S)}| = 1$ by the assumption $0 < x < 1$. Thus $|E_S| + |E_{\mathcal{C}(S)}| \geq 2$. Since no $e \in E$ goes between children of S, $t(S) \geq |E_S| + |E_{\mathcal{C}(S)}|$. Consequently, since $\sum_{S\in\mathcal{L}} t(S) \leq 2|E|$, we get: $t(S) = |E_S| + |E_{\mathcal{C}(S)}| = 2 \ \forall S \in \mathcal{L}$. Moreover, if an endnode of some $e \in E$ belongs to no $S \in \mathcal{L}$, then we get the contradiction $\sum_{S\in\mathcal{L}} t(S) \geq 2|E| + 1$. Now we replace our equation system by an equivalent one $\{x(E_S) - x(E_{\mathcal{C}(S)}) = b'_S : S \in \mathcal{L}\}$ obtained by elementary operations on the rows of the coefficients matrix. Note that x^* is also a unique solution to this new equation system. Moreover, this equation system has exactly two variables in each equation and all its coefficients are integral. By [13], the solution of such systems is always half-integral. ◀

Let us say that Tree Augmentation instance is **spider-shaped** if every in-edge in E is an up-edge. By a standard "iterative rounding" argument (cf. [18]), and using the correspondence between rooted trees and laminar families, we get from Lemma 8:

▶ **Corollary 9.** *Suppose that we are given a spider-shaped* Tree Augmentation *instance and* $b \in \mathbb{N}^F$. *Let* x *be an extreme point of the polytope* $\{x \in \mathbb{R}^E : x(\psi(f)) \geq b_f \ \forall f \in F, 0 \leq x \leq 1\}$. *Then* x *is half-integral (namely,* $x_e \in \{0, \frac{1}{2}, 1\}$ *for all* $e \in E$*) and* $x_e \in \{0, 1\}$ *for every* $e \in \delta(r)$.

The algorithm that computes an integral solution of cost $\frac{4}{3}(2C^{\mathsf{in}} + C^{\mathsf{cr}} + C^r)$ is as follows. We obtain a spider-shaped instance by removing all non-up in-edges and compute an optimal extreme point solution x to the Cut-LP. By Corollary 9, x is half-integral and $x_e \in \{0, 1\}$ for every $e \in \delta(r)$. We take into our solution every edge e with $x_e = 1$ and round the remaining $1/2$ entries using the algorithm of Cheriyan, Jordán & Ravi [3], that showed how to round a half-integral solution to the Cut-LP to integral solution within a factor of $4/3$. Thus we can compute a solution J of cost at most $c(J) \leq \frac{4}{3}c^{\mathsf{T}}x \leq \frac{4}{3}c^{\mathsf{T}}x^*$. We claim that $c^{\mathsf{T}}x \leq 2C^{\mathsf{in}} + C^{\mathsf{cr}} + C^r$. To see this let E^{in} be the set of in-edges and let x' be the E^{in}-up vector of x^*. Then x' is a feasible solution to the Cut-LP of value $2C^{\mathsf{in}} + C^{\mathsf{cr}} + C^r$, in the obtained Tree Augmentation instance with all non-up in-edges removed. But since x is an optimal solution to the same LP, we have $c^{\mathsf{T}}x \leq c^{\mathsf{T}}x' = 2C^{\mathsf{in}} + C^{\mathsf{cr}} + C^r$. This concludes the proof of Lemma 7 for the case of arbitrary costs.

Note that Corollary 9 implies a 4/3-approximation algorithm for spider-shaped Tree Augmentation instances. Parallel to our work, a stronger result was proved in [10].

▶ **Lemma 10** (Fiorini, Groß, Könemann & Sanitá [10])**.** *Spider-shaped* Tree Augmentation *instances can be solved optimally in polynomial time.*

The following theorem illustrates an application of Lemma 10.

▶ **Corollary 11.** Tree Augmentation *admits ratio* $3/2$ *for trees of diameter* ≤ 7.

Proof. The case $\mathsf{diam}(T) = 7$ is reduced to the case $\mathsf{diam}(T) \leq 6$ by "guessing" some optimal solution edge that covers the central T-edge. So assume that $\mathsf{diam}(T) \leq 6$. Let r be a center of T. Fix some optimal solution and let C^{in} and C^{cr} denote the fractional cost of in-edges and cross-edges in this solution. As before, apply the following two procedures.

1. Each branch B hanging on r is a tree of diameter ≤ 3, hence an optimal cover J_B of B can be computed in polynomial time. The union of the edge sets J_B gives a solution of cost at most $C^{\mathsf{in}} + 2C^{\mathsf{cr}}$.
2. Compute an optimal solution of the spider-shaped instance obtained by removing all non-up in-edges using Lemma 10; the cost of this solution is $2C^{\mathsf{in}} + C^{\mathsf{cr}}$.

Choosing the better among the two computed solutions gives a solution of cost at most $\min\{C^{\mathsf{in}} + 2C^{\mathsf{cr}}, 2C^{\mathsf{in}} + C^{\mathsf{cr}}\}$, while the optimal solution cost is $C^{\mathsf{in}} + C^{\mathsf{cr}}$. It is easy to see that the approximation ratio is bounded by $3/2$; if $C^{\mathsf{in}} \leq C^{\mathsf{cr}}$ then $C^{\mathsf{in}} + 2C^{\mathsf{cr}} \leq \frac{3}{2}(C^{\mathsf{in}} + C^{\mathsf{cr}})$, while if $C^{\mathsf{in}} > C^{\mathsf{cr}}$ then $2C^{\mathsf{in}} + C^{\mathsf{cr}} < \frac{3}{2}(C^{\mathsf{in}} + C^{\mathsf{cr}})$. ◀

Corollary 11 can be used further to obtain ratio $9/5$ for trees of diameter ≤ 15. In a similar way, one can further obtain ratio better than 2 for trees of diameter ≤ 31, and so on, but the ratio approaches 2 when the diameter becomes higher.

In the rest of this section we consider the case of unit costs. For this case we prove:

▶ **Lemma 12.** *Let* x *be an extreme point of the polytope* $\Pi = \{x \in \Pi^{Cut} : C^{\mathsf{in}}_x = a, C^{\mathsf{cr}}_x = b\}$ *where* $a, b \geq 0$, *such that* $x_e > 0$ *for every cross-edge* e. *Then the graph* (V, E^{cr}) *of cross-edges has no even cycle and each one of its connected components has at most one cycle.*

The proof of Lemma 12 will be given in the full version. From Lemma 12 we also get:

▶ **Corollary 13.** *In the case of unit costs there exists a polynomial time algorithm that computes $x \in \Pi$ such that the graph (V, E^{cr}) of cross edges of positive x-value is a forest and such that and $C^{\mathsf{in}}_x = C^{\mathsf{in}}$, $C^r_x = C^r$, and $C^{\mathsf{cr}}_x \leq \frac{4}{3}C^{\mathsf{cr}}$.*

Proof. Let Π be as in Lemma 12 where $a = C^{\mathsf{in}}$ and $b = C^r$ and let x be an optimal extreme point solution to the LP $\min\{\sum_{e\in E} x_e : x \in \Pi\}$. Let Q be a cycle of cross-edges and e the minimum x-value edge in Q. We update x by adding x_e to each of $x_{e'}, x_{e''}$ and setting $x_e = 0$. The increase in the value of x is at most $\frac{1}{3}\sum_{e\in Q} x_e$, and it is easy to see that x remains a feasible solution. In this way we can eliminate all cycles, ending with $x \in \Pi$ as required. ◀

Let x be as in Corollary 13 and let x' be an E^{in}-up vector of x. Note that $x' \in \Pi^{Cut}$, since $x \in \Pi^{Cut}$. We will show how to compute a solution J of size $c(J) \leq x'(E) \leq 2C^{\mathsf{in}} + \frac{4}{3}C^{\mathsf{cr}} + C^r$. While there exists a pair of edges $e = uv$ and $e' = u'v'$ such that $x'_e, x'_{e'} > 0$ and $T_{u'v'} \subset T_{uv}$ we do $x'_e \leftarrow x'_e + x'_{e'}$ and $x'_{e'} \leftarrow 0$. Then x' remains a feasible solution to the Cut-LP without changing the value (since we are in the case of unit costs). Hence we may assume that there is no such pair of edges. Let E' be the support of x'. If every leaf of T has some cross-edge in E' incident to it, then by the assumption above there are no up-edges. In this case, since E' is a forest, $x_e \geq 1$ for every $e \in E'$ and E' is a solution as required.

Otherwise, there is a leaf v of T such that no cross-edge in E' is incident to v. Then there is a unique up-edge e incident to v, and $x'_e \geq 1$. We take such e into our partial solution, updating x' and E' accordingly. Note that some cross-edges may become r-edges, but no up-edge can become a cross-edge, and the set of cross-edges remains a forest. Applying this as long as such leaf v exists, we arrive at the previous case, where adding E' to the partial solution gives a solution as required. This concludes the proof of Lemma 7.

3 An upper bound on the integrality gap of the Cut-LP (Theorem 2)

Let us write the Cut-LP as well as its dual LP explicitly:

$$\begin{array}{lll} \min & \sum_{e\in E} x_e & \\ \text{s.t.} & \sum_{e\in\psi(f)} x_e \geq 1 & \forall f \in F \\ & x_e \geq 0 & \forall e \in E \end{array} \qquad \begin{array}{lll} \max & \sum_{f\in F} y_f & \\ \text{s.t.} & \sum_{\psi(f)\ni e} y_f \leq 1 & \forall e \in E \\ & y_f \geq 0 & \forall f \in F \end{array}$$

To prove that the integrality gap of the Cut-LP is at most $28/15 = 2 - 2/15$ we will show that a simplified version from [16] of the algorithm of [9] has the desired performance. For the analysis, we will use the dual fitting method. We will show how to construct a (possibly infeasible) dual solution $y \in \mathbb{R}^F_+$, that has the following two properties:

Property 1. y fully pays for the constructed solution J, namely, $|J| \leq \sum_{f\in F} y_f$.

Property 2. y may violate the dual constraints by a factor of at most $\rho = 28/15$.

From the second property we get that y/ρ is a feasible dual solution, hence by weak duality the value of y is at most ρ times the optimal value of the Cut-LP. Combining with the first property we get that $|J|$ is at most ρ times the optimal value of the Cut-LP.

The algorithm of [9, 16] iteratively finds a pair T', J' where T' is a subtree of the current tree and J' covers T', contracts T', and adds J' to J. We refer to nodes created by contractions as **compound nodes** and denote by C the set of non-leaf compound nodes of the current tree. Non-compound nodes are referred to as **original nodes**. For technical reasons, the root r is considered as a compound node, hence initially $C = \{r\}$.

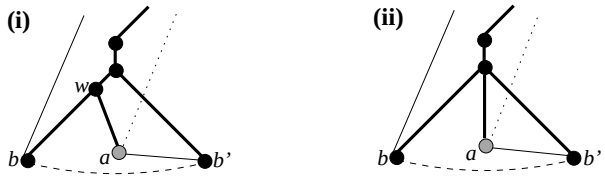


Figure 3 Dangerous trees. T-edges are shown by bold lines, edges in M by dashed lines, other existing edges by thin solid lines, and edges that cannot exist by dotted lines. Original nodes are shown by black circles, while nodes that may be compound are shown by gray circles. Some of the edges may be paths, possibly of length 0. A dangerous tree of type (i) has two nodes with 2 children each, and contracting the path between these two nodes results in a dangerous tree of type (ii).

To identify a pair T', J' as above, the algorithm maintains a matching M on the original leaves. We denote by U the leaves of the current tree unmatched by M. A subtree T' of T is **M-compatible** if for any $bb' \in M$ either both b, b' belong to T' or none of b, b' belongs to T'; in this case we will also say that a contraction of T' is M-compatible. Assuming that all compound nodes were created by M-compatible contractions, then the following type of contractions is also M-compatible.

▶ **Definition 14** (greedy contraction)**.** Adding to the partial solution J an edge e with both endnodes in U and contracting T_e is called a **greedy contraction**.

Given a complete rooted M-compatible subtree T' of T let us use the following notation:

- $M' = M(T')$ is the set of edges in M with both endnodes in T'.
- $U' = U(T')$ is the set of unmatched leaves of T'.
- $C' = C(T')$ is the set of non-leaf compound nodes of T'.

▶ **Definition 15** (semi-closed tree)**.** Let T' be a complete rooted subtree of T. For a subset A of nodes of T' we say that T' is **A-closed** if there is no edge from A to a node outside T', and T' is **A-open** otherwise. Given a matching M on the leaves of T, we say that T' is **semi-closed** if it is M-compatible and U'-closed.

The following definition characterizes semi-closed subtrees that we want to avoid. We will say that T' with 3 leaves is of type (i) if it has two nodes with exactly two children each (see the node w and its father in Fig. 3(i)) and T' is of type (ii) otherwise (see Fig. 3(ii)).

▶ **Definition 16** (dangerous semi-closed tree)**.** A semi-closed subtree T' of T is **dangerous** if it is as in Fig. 3. Namely, $|M'| = 1$, $|U'| = 1$, $|C'| = 0$, and if a is the leaf of T' unmatched by M then: T' is a-closed and there exists an ordering b, b' of the matched leaves of T' such that $ab' \in E$, the contraction of ab' does not create a new leaf, and T' is b-open.

In [9, 17] the following is proved:

▶ **Lemma 17** ([9, 17])**.** *Suppose that the current tree T was obtained from the initial tree by sequentially applying a greedy contraction or a semi-closed tree contraction, and that T has no greedy contraction. Then there exists a polynomial time algorithm that finds a non-dangerous semi-closed subtree T' of T and a cover J' of T' of size $|J'| = |M'| + |U'|$.*

▶ **Definition 18** (twin-edge, stem)**.** An edge on L is a **twin-edge** if its contraction results in a new leaf. The least common ancestor of the endnodes of a twin-edge is a **stem**.

Let $L(M)$ denote the set of leaves matched by M. The algorithm is as follows:

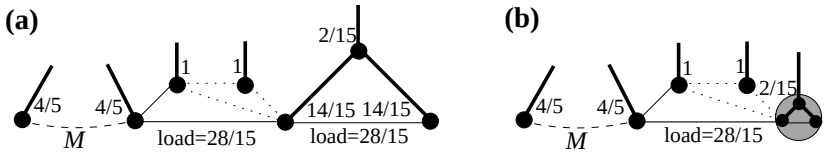


Figure 4 (a) Initial duals at step 1 of Algorithm 3 and the initial loads. Here there is one stem and $|M| = 1$. (b) After contracting the twin-edge, the new compound node has credit 1.

Algorithm 2: ITERATIVE-CONTRACTION$(T = (V, F), E)$

initialize: $M \leftarrow$ inclusionwise maximal matching on L among non-twin edges
$J \leftarrow$ inclusionwise maximal matching on $L \setminus L(M)$
while do
  T has at least 2 nodes
exhaust greedy contractions
if T has at least 2 nodes then for T, J' as in Lemma 17 do: $J \leftarrow J \cup J'$, $T \leftarrow T/T'$
return J

We now describe how to construct y satisfying Properties 1 and 2. For simplicity of exposition, we use the notation y_v and $y_{T'}$ to denote the dual variable of the parent edge of v and of T', respectively. With this notation, Algorithm 3 incorporates into Algorithm 2 the steps of the construction of the dual (possibly infeasible) solution y.

Algorithm 3: DUAL-CONSTRUCTION$(T = (V, F), E)$

Initialize: $M \leftarrow$ inclusionwise maximal matching on L among non-twin edges
$J \leftarrow$ inclusionwise maximal matching on $L \setminus L(M)$ (see Fig. 4)
- $y_v \leftarrow 1$ if $v \in L \setminus L(M \cup J)$
- $y_v \leftarrow 4/5$ if $v \in L(M)$
- $y_v \leftarrow 14/15$ if $v \in L(J)$
- $y_v \leftarrow 2/15$ if v is a stem of an edge in J

while do
  T has at least 2 nodes
exhaust greedy contractions
if T has at least 2 nodes then for T, J' as in Lemma 17 do: $J \leftarrow J \cup J'$, $T \leftarrow T/T'$
Case 1: $|C'| = 0$ and either: $|M'| = 0$ or $|M'| = 1, |U'| \geq 2$
- $y_{T'} \leftarrow 2/5$
- $y_v \leftarrow y_v + 2/5$ if $v \in U'$ and $y_v \leftarrow y_v - 2/5$ if $v \in L' \setminus U'$

Case 2: $|C'| = 0$ and $|M'| = |U'| = 1$
- update y as shown in Fig. 5 **return** J

We now define certain quantities that will help us to prove that at the end of the algorithm $|J| \leq \sum_{f \in F} y_f$ and that y violates the dual constraints by a factor of at most 28/15.

▶ **Definition 19** (load of an edge). Given $y \in \mathbb{R}^F_+$ and an edge $e = uv$, the **load** $\sigma(e)$ **of** e is the sum of the dual variables in the constraint of e in the dual LP, namely $\sigma_e = \sum_{\psi(f) \ni e} y_f$.

▶ **Definition 20** (credit of a node). Consider a constructed dual solution y and a node c of T/J during the algorithm, where c is obtained by contracting the subtree T' of T. The

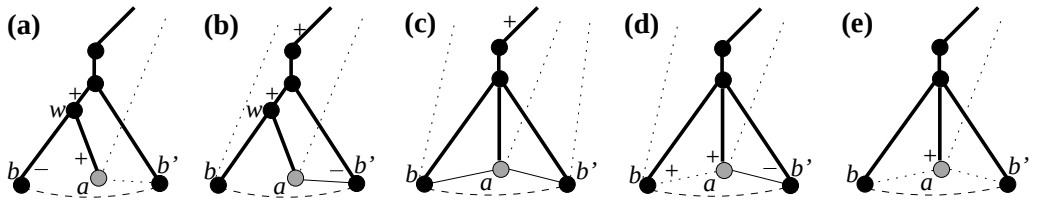


Figure 5 Non-dangerous trees with $|M'| = |U'| = 1$ and duals updates in Case 2 of Algorithm 3. Here "+" means increasing the dual variable by 2/5 and "−" means decreasing the dual variable by 2/5. All trees are a-closed. The trees in (a,b) are non-dangerous trees of type (i), and the trees in (c,d,e) are non-dangerous trees of type (ii). In (a) the edge ab' is missing and in (b) ab' is present and T' is b-closed. In (c) both ab and ab' are present, hence to be non-dangerous the tree must be both b'-closed and b-closed. In (d) ab' is present hence the tree must be b-closed; the case when ab present and the tree is b'-closed is identical. In (e) both ab and ab' are missing.

credit $\pi(c)$ is defined as follows. Let $\pi'(c)$ be the sum of the dual variables y of the edges of T' and the parent edge of v minus the number of edges used by the algorithm to contract T' into c. Then $\pi(c) = \pi'(c) + 1$ if $r \in T'$ and $\pi(c) = \pi'(c)$ otherwise.

We need to prove that at the end of the algorithm, $\sigma(e) \leq 28/15$ for all $e \in E$ and that the unique node of T has credit ≥ 1. For an edge $e = uv$ the **level** $\ell(e)$ of e is the number of endnodes of e in the leaves and compound nodes of T. Clearly, $\ell(e) \in \{0, 1, 2\}$ and if both endnodes of e lie in the same compound node then $\ell(e) = 2$. In the full version we prove:

▶ **Lemma 21.** *At the end of step 1 of the algorithm, and then at the end of every iteration in the "while" loop, the following holds.*

- *For any edge e: $\sigma(e) \leq \frac{28}{15}$ if $\ell(e) = 2$, $\sigma(e) \leq \frac{16}{15}$ if $\ell(e) = 1$, and $\sigma(e) = 0$ if $\ell(e) = 0$.*
- *$\pi(c) \geq 1$ if c is an unmatched leaf or a compound node of T.*

The following LP-relaxation was suggested by the author several years ago. We call an odd size set B of T-edges a **bunch** if no two edges of B lie on the same path in T. Let $\mathcal{B}$ denote the set of bunches in T. For any $B \in \mathcal{B}$ at least $w_B = (|B| + 1)/2$ edges are needed to cover B. The corresponding Bunch-LP and its dual LP are:

$$\begin{array}{lll} \min & \sum_{e \in E} x_e & \\ \text{s.t.} & \sum_{e \in \psi(B)} x_e \geq w_B & \forall B \in \mathcal{B} \\ & x_e \geq 0 & \forall e \in E \end{array} \qquad \begin{array}{lll} \max & \sum_{B \in \mathcal{B}} w_B y_B & \\ \text{s.t.} & \sum_{\psi(B) \ni e} y_B \leq 1 & \forall e \in E \\ & y_B \geq 0 & \forall B \in \mathcal{B} \end{array}$$

A **k-bunch** is a bunch of size k. Let k-Bunch-LP be the restriction of the Bunch-LP to bunches of size $\leq k$. Note that Theorem 1 says that the integrality gap of the 1-Bunch-LP is at most 28/15. We can easily prove a much netter bound for the 3-Bunch-LP.

▶ **Theorem 22.** *For unit costs, the integrality gap of the* 3-Bunch-LP *is at most* 7/4.

Proof. We use the same algorithm but define the dual variables slightly differently. In the initialization step we set $y_v \leftarrow 1$ if $v \in L \setminus L(M \cup J)$, $y_v \leftarrow 3/4$ if $v \in L(M)$, $y_v \leftarrow 1/2$ if $v \in L(J)$, and $y_B \leftarrow 1/2$ if B is the set of the 3 T-edges incident to a stem of an edge in J.

In Case 1 we update $y_{T'} \leftarrow 1/2$, $y_v \leftarrow y_v + 1/2$ if $v \in U'$ and $y_v \leftarrow y_v - 1/2$ if $v \in L' \setminus U'$.

In Case 2 we update y as shown in Fig. 5, where here "−" means decreasing the dual variable by 1/2 and:

- In (a), $y_B \leftarrow y_B + 1/2$ where B is the 3-bunch formed by the parent edges of a, b, w.
- In (b,c,d,e) "+" means increasing the dual variable by $1/2$.

The rest of the proof of Theorem 22 is similar to that of Theorem 2, and will be presented in the full version of the paper. ◀

Acknowledgment. The author thanks László Végh, Shoni Gilboa, Manor Mendel, Moran Feldman, and Gil Alon for several discussions.

References

1. D. Adjiashvili. Beating approximation factor two for weighted tree augmentation with bounded costs. In *SODA*, pages 2384–2399, 2017.
2. J. Cheriyan and Z. Gao. Approximating (unweighted) tree augmentation via lift-and-project, part II. Manuscript, 2015.
3. J. Cheriyan, T. Jordán, and R. Ravi. On 2-coverings and 2-packing of laminar families. In *ESA*, pages 510–520, 1999.
4. J. Cheriyan, H. Karloff, R. Khandekar, and J. Koenemann. On the integrality ratio for tree augmentation. *Operation Research Letters*, 36(4):399–401, 2008.
5. N. Cohen and Z. Nutov. A $(1 + \ln 2)$-approximation algorithm for minimum-cost 2-edge-connectivity augmentation of trees with constant radius. *Theoretical Computer Science*, 489-490:67–74, 2013.
6. M. Cygan. Private communication, 2016.
7. M. Cygan, F. Fomin, F. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2016.
8. G. Even, J. Feldman, G. Kortsarz, and Z. Nutov. A 3/2-approximation for augmenting a connected graph into a two-connected graph. In *APPROX*, pages 90–101, 2001.
9. G. Even, J. Feldman, G. Kortsarz, and Z. Nutov. A 1.8-approximation algorithm for augmenting edge-connectivity of a graph from 1 to 2. *ACM Transactions on Algorithms*, 5(2), 2009.
10. S. Fiorini, M. Groß, J.Könemann, and L. Sanitá. A $\frac{3}{2}$-approximation algorithm for tree augmentation via chvátal-gomory cuts. `https://arxiv.org/abs/1702.05567`, Feb 27, 2017.
11. G. Frederickson and J. Jájá. Approximation algorithms for several graph augmentation problems. *SIAM J. Computing*, 10:270–283, 1981.
12. M. Goemans, A. Goldberg, S. Plotkin, E. Tardos D. Shmoys, and D. Williamson. Improved approximation algorithms for network design problems. In *SODA*, pages 223–232, 1994.
13. D. Hochbaum, N. Megiddo, J. Naor, and A. Tamir. Tight bounds and 2-approximation algorithms for integer programs with two variables per inequality. *Math. Programming*, 62:69–83, 1993.
14. K. Jain. A factor 2 approximation algorithm for the generalized steiner network problem. *Combinatorica*, 21(1):39–60, 2001.
15. S. Khuller and R. Thurimella. Approximation algorithms for graph augmentation. *J. of Algorithms*, 14:214–225, 1993.
16. G. Kortsarz and Z. Nutov. LP-relaxations for tree augmentation. In *APPROX-RANDOM*, pages 13:1–13:16, 2016.
17. G. Kortsarz and Z. Nutov. A simplified 1.5-approximation algorithm for augmenting edge-connectivity of a graph from 1 to 2. *ACM Transactions on Algorithms*, 12(2):23, 2016.
18. L. C. Lau, R. Ravi, and M. Singh. *Iterative Methods in Combinatorial Optimization*. Cambridge University Press, 2011.
19. Y. Maduel and Z. Nutov. Covering a laminar family by leaf to leaf links. *Discrete Applied Mathematics*, 158(13):1424–1432, 2010.

20 D. Marx and L. Végh. Fixed-parameter algorithms for minimum-cost edge-connectivity augmentation. *ACM Transactions on Algorithms*, 11(4):27, 2015.

21 H. Nagamochi. An approximation for finding a smallest 2-edge connected subgraph containing a specified spanning tree. *Discrete Applied Mathematics*, 126:83–113, 2003.

Prize-Collecting TSP with a Budget Constraint

Alice Paul[1], Daniel Freund[2], Aaron Ferber[3], David B. Shmoys[4], and David P. Williamson[5]

1 **Operations Research and Information Engineering, Cornell University, Ithaca, NY, USA**
ajp336@cornell.edu
2 **Center for Applied Mathematics, Cornell University, Ithaca, NY, USA**
df365@cornell.edu
3 **Operations Research and Information Engineering, Cornell University, Ithaca, NY, USA**
amf272@cornell.edu
4 **Operations Research and Information Engineering, Cornell University, Ithaca, NY, USA**
dbs10@cornell.edu
5 **Operations Research and Information Engineering, Cornell University, Ithaca, NY, USA**
dw36@cornell.edu

Abstract

We consider constrained versions of the prize-collecting traveling salesman and the minimum spanning tree problems. The goal is to maximize the number of vertices in the returned tour/tree subject to a bound on the tour/tree cost. We present a 2-approximation algorithm for these problems based on a primal-dual approach. The algorithm relies on finding a threshold value for the dual variable corresponding to the budget constraint in the primal and then carefully constructing a tour/tree that is just within budget. Thereby, we improve the best-known guarantees from $3+\epsilon$ and $2+\epsilon$ for the tree and the tour version, respectively. Our analysis extends to the setting with weighted vertices, in which we want to maximize the total weight of vertices in the tour/tree subject to the same budget constraint.

1998 ACM Subject Classification G.2.2 [Mathematics of Computing] Graph Theory

Keywords and phrases Approximation Algorithms, Traveling Salesman Problem

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.62

1 Introduction

In the classical traveling salesman problem, we are given an undirected graph $G = (V, E)$ with edge costs $c_e \geq 0$ for all $e \in E$. The goal is to construct a tour visiting all vertices in the graph while minimizing the cost of edges in the tour. If, however, we are given a bound on the cost of the tour, then we may not be able to visit all vertices. In particular, suppose that we are given a budget $D \geq 0$. In the **budgeted prize-collecting traveling salesman problem**, a valid tour is a multiset of edges F such that (a) F specifies a tour on a subset $S \subseteq V$ and (b) the cost of the edges in F is at most D. The goal is to find a valid tour F that maximizes $|S|$, the number of vertices visited. Here, we do not require the graph to be complete and allow a tour to visit nodes more than once. Similarly, in the **budgeted prize-collecting minimum spanning tree problem**, a valid tree is a set of edges T such that (a) T specifies a spanning tree on a subset $S \subseteq V$ and (b) the cost of the edges in T is at most D. Again, the goal is to find a valid tree T that maximizes $|S|$.

The budgeted version of the traveling salesman problem arises naturally in many routing problems that have a distance or time constraint. For example, a bikeshare system has bike stations located around a city that may need repair. Throughout the day, the system operator wants to route a repairman over his work period while maximizing the number of stations that receive maintenance (in fact, this precise question emerged from our ongoing work with New York City Bikeshare [11]). We can represent this problem as a budgeted prize-collecting traveling salesman problem. Further, we can also capture the setting where stations have varying importance; we discuss in Section 6 how to extend our algorithm to a setting in which vertices have weights and the goal is to maximize the weight of vertices visited. In Section 7, we apply our algorithm to instances using Citi Bike data in New York City. The budgeted version of the minimum spanning tree also arises in a range of applications, including telecommunication network design problems where an infrastructure budget is weighed against the number of customers served.

In this paper, we present a 2-approximation algorithm for both problems. Our algorithm is based on a primal-dual subroutine which uses a linear programming relaxation of this problem. First, we search for a "good" value for the dual variable corresponding to the budget constraint in the primal. Having set this variable, we can then increase the other dual variables and form a forest of edges whose corresponding dual constraint is tight. For the tour problem, we then choose a tree in this forest and carefully prune it so that doubling this tree forms a tour that will be just within budget. For the tree problem, we prune edges such that the tree itself is just within budget. Lastly, we show that either our constructed tour/tree is within a factor of 2 of optimal or we can identify a subgraph to recurse on.

Literature Review

There have been many prize-collecting variants of both the traveling salesman problem (TSP) and the minimum spanning tree problem (MST) that seek to balance the number of vertices in the tree or tour with the cost of edges used. Johnson, Minkoff, and Phillips [16] characterize four main variants of prize-collecting MST problems: the Goemans-Williamson Minimization problem that minimizes the cost of edges plus a penalty for vertices not in the tree, the Net Worth Maximization problem that maximizes the weight of vertices in the tree minus the cost of used edges, the Quota problem that minimizes the cost of a tree containing at least Q vertices, and, finally, the Budget problem that maximizes the number of vertices in the tree subject to the cost of the tree being at most D. All of the variants above can be extended to a corresponding TSP version that constructs a tour rather than a tree.

Our algorithm is most similar to that of Garg [13], who presents a 2-approximation algorithm for the Quota problem for MST, improving upon the previous results of Garg [12], Arya and Ramesh [2], and Blum, Ravi, and Vempala [4]. Johnson et al. [16] observe that a 2-approximation algorithm to the Quota problem yields a $(3+\epsilon)$-approximation algorithm to the corresponding Budget problem. To our knowledge this was the previously best-known guarantee for the MST variant. Prior to this result, Levin [18] proved a $(4+\epsilon)$-approximation algorithm. Our 2-approximation algorithm for the budgeted prize-collecting MST thus improves upon the best known approximation ratio. While our algorithm is similar to that of Garg [13], our analysis differs in how we find the threshold value for the dual variable; further, our overall proof relies on more precise accounting.

For the Goemans-Williamson Minimization problem for MST, Archer et al. [1] obtain a $(2-\epsilon)$-approximation guarantee, improving upon the long-standing bound of 2 obtained by Goemans and Williamson [14] in 1995. Further, Archer et al. [1] successfully applied this algorithm to telecommunication network problems. Lastly, Feigenbaum et al. [9] show the Net Worth Maximization problem for MST is NP-hard to approximate within any constant.

To the best of our knowledge, the previous best approximation guarantee for the budgeted prize-collecting TSP arises from a special case of a result by Chekuri, Korula, and Pál [6]. Their work provides a $(2+\epsilon)$-approximation algorithm for the more general orienteering problem, where the goal is to find an $s-t$ path, where s and t are given, with bounded cost that maximizes the number of vertices visited on the path. By setting $s=t$ and iterating over all vertices, this yields a $(2+\epsilon)$-approximation algorithm for the budgeted prize-collecting TSP. The orienteering problem itself has attracted much attention within the combinatorial optimization community, with other variants studied by [21], [5], [8], [7], and [15].

There exist other adaptations of prize-collecting problems not discussed above. Specifically, Ausiello, Demange, Laura, and Paschos [3] present a 2-approximation algorithm for an on-line variant of the Quota problem for the TSP. Frederickson and Wittman [10] study the so-called traveling repairmen problem, in which each vertex can only be visited within a specific time window and the goal is to either maximize the number of vertices visited within a certain time period or to minimize the time visiting all vertices; they give constant-factor approximation algorithms for both variations of this problem. Lastly, Nagarajan and Ravi [19] study the problem of minimizing the number of tours to cover all vertices subject to each tour having bounded distance. They give a 2-approximation algorithm for tree metric distances.

The paper is structured as follows. In Section 2, we present the linear programming (LP) relaxation for the budgeted prize-collecting traveling salesman problem. In Section 3, we use this LP to present the primal-dual subroutine that will inform our decisions and develop some intuition behind what types of tours will be near optimal. In Section 4, we show how to set the dual variable corresponding to the budget constraint, and in Section 5, we show how to construct our proposed tour. In Section 6, we prove that our overall algorithm is a 2-approximation algorithm and present computational experiments in Section 7. For ease of presentation, we present only our result for the budgeted prize-collecting traveling salesman problem but the analysis extends easily to the corresponding MST case.

2 Notation

For each $S \subseteq V$, let $z_S \in \{0,1\}$ be a variable representing whether or not we choose to tour the vertices in S, and for each edge $e \in E$, let $x_e \in \mathbb{Z}^+$ be a variable representing how many copies of e to include in the tour. Then, the following is a linear programming relaxation for the budgeted prize-collecting traveling salesman problem.

$$\begin{aligned}
\text{maximize} \quad & \sum_{S \subseteq V} |S| z_S \\
\text{subject to} \quad & \sum_{e:e\in\delta(S)} x_e \geq 2 \sum_{T:S\subset T} z_T \quad \forall S \subset V \\
& \sum_{e\in E} c_e x_e \leq D \\
& \sum_{S\subseteq V} z_S \leq 1 \\
& z_S, x_e \geq 0
\end{aligned}$$

The first constraint states that if we choose to tour a subset T such that $S \subset T$ then we must have at least two edges across the cut S. The dual of this linear program is given by

the following.

$$\begin{aligned} &\text{minimize } \Lambda_1 D + \Lambda_2 \\ &\text{subject to } (2 \sum_{T:T\subset S} y_T) + \Lambda_2 \geq |S| \quad \forall S \subseteq V \\ &\qquad \sum_{S:e\in\delta(S)} y_S \leq \Lambda_1 c_e \quad \forall e \in E \\ &\qquad \Lambda_1, \Lambda_2, y_S \geq 0 \end{aligned}$$

In order to construct a tour, we will rely on a primal-dual subroutine. We first note in Theorem 2 that if we find $\Lambda_1 \geq 0$ and $y_S \geq 0$ that satisfy the dual constraint for every edge, then we can always set Λ_2 such that we have a full feasible dual solution. Suppose that we first set the value of Λ_1. The primal-dual subroutine will use this set value to construct a full dual solution and corresponding potential tours. These tours may or may not be feasible with respect to the budget constraint. Therefore, we will adjust Λ_1 to find a feasible solution with bounded approximation ratio.

3 Primal-Dual Subroutine

The primal-dual subroutine for a fixed Λ_1 is similar to the 2-approximation algorithm for the prize-collecting traveling salesman problem without a budget constraint presented by Goemans and Williamson [14]. Initially, we set all y_S to be 0 and set our collection of active sets to be all singleton nodes. Then, in each iteration, we increase y_S corresponding to all $S \subset V$ in the collection of active sets until either a dual constraint for an edge between two sets becomes tight, or a set becomes neutral.

▶ **Definition 1.** We say a subset $S \subseteq V$ is **neutral** if $2\sum_{T:T\subseteq S} y_T = |S|$.

If an edge becomes tight between two subsets S_1 and S_2, we add the edge to our solution and remove both S_1 and S_2 from the collection of active sets and add $S_1 \cup S_2$ to it. If a set becomes neutral, we mark the set as inactive and remove it from the collection of active sets. Once the collection of active sets is empty, we prune inactive sets of degree 1 and return the remaining edges in our solution (cf. Algorithm 1).

Algorithm 1 Primal-Dual Algorithm (PD(λ_1))

```
1:  procedure PD(λ1 ≥ 0)
2:      y_S ← 0, Λ1 ← λ1, T ← {}.
3:      mark all i ∈ V as active.
4:      while there exists an active subset do
5:          raise y_S uniformly for all active subsets S until either
6:          if an active set S becomes neutral then
7:              mark S as inactive.
8:          else if the dual constraint for edge e between S1 and S2 becomes tight then
9:              T ← T ∪ {e}.
10:             mark S = S1 ∪ S2 as active, remove S1 and S2 from the active subsets.
11:     T' ← T.
12:     while there exists a set S marked inactive such that |δ(S) ∩ T'| = 1 do
13:         remove all edges with at least one endpoint in S from T'.
        return two of each edge in T'.
```

Properties of the algorithm PD(λ_1) (by construction)

1. The algorithm terminates in polynomial time.
2. Throughout the algorithm, T is a forest, and by extension T' is a forest.
3. For all edges, the corresponding dual constraint is satisfied.
4. For all $e \in T$, the dual constraint for e is tight.

▶ **Theorem 2.** *Given $\lambda_1 \geq 0$, let y be as created by the algorithm. Then, there exists a value $\lambda_2 \geq 0$ such that $(y, \lambda_1, \lambda_2)$ is a feasible dual solution.*

Proof. Since all edge constraints are satisfied, we may set λ_2 to the maximum of zero and

$$\min_{S \subseteq V} \left[|S| - (2 \sum_{T:T \subset S} y_T) \right].$$

By construction, all dual constraints will be satisfied. ◀

3.1 Analysis

In this section, we assume that we have set $\Lambda_1 = \lambda_1$ in the primal-dual subroutine such that we produced a feasible dual solution $(y, \lambda_1, \lambda_2)$ (where we may not know the actual value of λ_2). We let $\mathcal{S}$ be the collection of sets that were active in some iteration of the algorithm and let $\mathcal{S}^+ = \mathcal{S} \cup \{V\}$. Since any set in $\mathcal{S}$ is either a single node or the union of other sets in $\mathcal{S}$, this is a laminar collection.

▶ **Lemma 3.** *For any $S \subseteq V$, $(2 \sum_{T:T \subseteq S} y_T) \leq |S|$.*

Proof. Any set S can be divided into maximal disjoint laminar sets $S_1, S_2, \ldots, S_c \in \mathcal{S}$. Therefore,

$$2 \sum_{T:T \subseteq S} y_T = 2 \sum_{i=1}^{c} \sum_{T:T \subseteq S_i} y_T \leq \sum_{i=1}^{c} |S_i| = |S|,$$

where the inequality comes from the fact that we make inactive any neutral subset. ◀

We first define a potential $\pi(S)$ for each subset $S \subseteq V$. These values will help us find an upper bound on the size of a feasible tour.

▶ **Definition 4.** For any subset $S \subseteq V$, we define the **potential** of S to be

$$\pi(S) := |S| - (2 \sum_{T:T \subset S} y_T).$$

For a set $S \in \mathcal{S}$, $\pi(S)$ is exactly equal to twice the amount that we could have increased y_S until S went neutral. In particular, if S was formed by the union of S_1 and S_2, then

$$\pi(S) = \pi(S_1) + \pi(S_2) - 2y_{S_1} - 2y_{S_2}.$$

If S_2 went inactive before merging with S_1, then this simplifies to $\pi(S) = \pi(S_1) - 2y_{S_1}$.

Given these potentials and our constructed dual solution, we give a bound on the size of an optimal solution.

▶ **Theorem 5.** *Let $O^\star$ be an optimal subset of vertices to tour and $F^\star$ be the edges in an optimal tour on $O^\star$. Further, let O be the minimal set in $\mathcal{S}^+$ that contains $O^\star$. Since $V \in \mathcal{S}^+$, such a set always exists. Then,*

$$|O^\star| \leq \lambda_1 D + \pi(O).$$

Proof. We provide the proof in the full version of the paper. ◀

Given the bound in Theorem 5, we argue that to construct a good tour we should try to find a tree $\bar{T}$ with cost close to $\frac{1}{2}D$ such that the set $\bar{S}$ of spanned vertices has high potential. Then, doubling this tree will give a feasible tour close to optimal. In order to find such a tree, we first rely on finding a good value of Λ_1.

4 Setting Λ_1

Our goal is to set Λ_1 so as to find a tree with cost very close to $\frac{1}{2}D$. Note that Λ_1 controls the cost of the edges, and as Λ_1 increases, edges become more expensive yielding smaller connected components in the primal-dual subroutine. In particular, for $\Lambda_1 = 0$ all edges go tight immediately and for $\Lambda_1 > n/(2\min_{e:c_e>0} c_e)$ all vertices go neutral before a single non-zero edge goes tight. When edges go tight and subsets go neutral at the same time, we may assume that subset events are considered first. Further, we assume that we break edge/subset ties using cost/size and then some known ordering (e.g. lexicographical).

If a minimum spanning tree on the graph has cost $\leq \frac{1}{2}D$, then we double this tree to get a feasible and optimal tour. Otherwise, suppose that we have found values l and r $(l < r)$ such that when we run PD(l^+) the largest component in T' has cost $\geq \frac{1}{2}D$ and when we run PD(r^-) the largest component in T' has cost $< \frac{1}{2}D$. Here, $x^- = x - \varepsilon$ and $x^+ = x + \varepsilon$ where ε is infinitesimally small.

▶ **Lemma 6.** *In polynomial time, we can find a threshold value λ_1 such that when we run PD(λ_1^-) the largest component in T' has cost $\geq \frac{1}{2}D$ and when we run PD(λ_1^+) the largest component in T' has cost $< \frac{1}{2}D$.*

Proof. We refer to an edge going tight during the primal-dual subroutine as an edge event and we refer to a subset going neutral as a subset event. Assume we have values l and r such that the first k events are the same when running the subroutine for any Λ_1 between l^+ and r^-. Further, assume that for each subset S we can find values α_S and β_S such that at the end of the first k events $y(S) = \Lambda_1\alpha_S + \beta_S$ for any Λ_1 between l^+ and r^-. Note that this is trivially true for the base case with l and r defined above and $k = 0$ since all y values will be zero.

To find the next event to occur, we need to find the time after the kth event that each subset will go neutral and each edge will go tight. Observe that an active set S will go neutral at time

$$\frac{1}{2}|S| - \sum_{T\subseteq S} y_T = \frac{1}{2}|S| - \sum_{T\subseteq S}[\Lambda_1\alpha_T + \beta_T],$$

an edge with exactly one endpoint in an active component will go tight at time

$$\Lambda_1 c_e - \sum_{T:e\in\delta(T)} y_T = \Lambda_1 c_e - \sum_{T:e\in\delta(T)}[\Lambda_1\alpha_T + \beta_T],$$

and an edge with both endpoints in different active components will go tight at time $\frac{1}{2}$ the above amount. The minimum of these values will determine the next event to occur. Since all these times are affine in Λ_1, we can divide the interval between l^+ and r^- into smaller subintervals such that the first $k+1$ events will be identical on these subintervals. See Figure 1.

By looking at these subintervals, either we identify a threshold point λ_1 or there exists a subinterval between l^+_{new} and r^-_{new} such that when we run PD(l^+_{new}) the largest component in T' has cost $\geq \frac{1}{2}D$ and when we run PD(r^-_{new}) the largest component in T' has cost $< \frac{1}{2}D$.

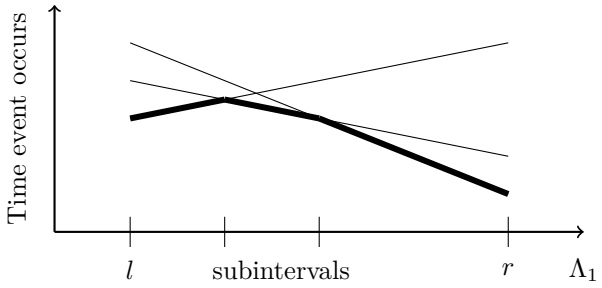


Figure 1 Finding the subintervals between l and r where the time of the next event is in bold.

Further, since the time of the $(k+1)$th event is an affine function in Λ_1, we can add this function to the affine function $y(S)$ for each active set S to get the new affine function for this y value, updating the α's and β's accordingly. Thus, the inductive hypothesis holds and eventually we can find a threshold point λ_1. ◀

We use this threshold point λ_1 to understand the subroutine for $\mathrm{PD}(\lambda_1)$. Consider running the subroutine for λ_1^+ and λ_1^- and comparing event by event. We let y^+ correspond to the y variables when running $\mathrm{PD}(\lambda_1^+)$ and y^- to the y variables when running $\mathrm{PD}(\lambda_1^-)$.

▶ **Lemma 7.** *Throughout the two subroutines, the following two properties hold:*
- *All active components in (V, T) are the same.*
- *For all $S \subseteq V$, the difference between y_S^+ and y_S^- is infinitesimally small.*

Proof. At the start of the subroutines this is true since all y^+ and y^- variables are zero. Now assume that this is true at some time t into the subroutines. As argued above, the next event to occur depends on the minimum of functions linear in Λ_1. Further, since the current active components are the same, the possible subset and edge events are the same.

In particular, the time for each subset to go neutral in $\mathrm{PD}(\lambda_1^+)$ is $\frac{1}{2}|S| - \sum_{T \subseteq S} y_T^+$, and is infinitesimally different from the time for that subset to go neutral in $\mathrm{PD}(\lambda_1^-)$. Similarly, the time for each edge to go tight is infinitesimally different between the two subroutines. Therefore, the next event to occur is only different between the two subroutines if two events occur at the same time for $\mathrm{PD}(\lambda_1)$.

If the next event is the same for the two subroutines, then the active components will remain the same and we raise all active components by an infinitesimally different amount. Therefore, the inductive properties will continue to hold. Otherwise, suppose the next event is different. We consider four cases:

1. Subset X goes neutral for $\mathrm{PD}(\lambda_1^-)$ and subset Y goes neutral for $\mathrm{PD}(\lambda_1^+)$.
2. Edge e goes tight for $\mathrm{PD}(\lambda_1^-)$ and edge f goes tight for $\mathrm{PD}(\lambda_1^+)$.
3. Edge e goes tight for $\mathrm{PD}(\lambda_1^-)$ and subset X goes neutral for $\mathrm{PD}(\lambda_1^+)$.
4. Subset X goes neutral for $\mathrm{PD}(\lambda_1^-)$ and edge e goes tight for $\mathrm{PD}(\lambda_1^+)$.

In the first case, the times for both X and Y to go neutral must be infinitesimally different and the other subset will go neutral immediately after the first. Therefore, after both X and Y go neutral, the amount that we have raised all y variables will be infinitesimally different and the current active components will be the same. Thus, the two inductive properties will continue to hold.

Similarly for the second case, if e and f are not between the same two components, the other edge will go tight immediately after, and the inductive properties will continue to hold. Otherwise, e and f are between the same components. Thus, when e goes tight, f is no

longer eligible to go tight but the newly merged active component will be the same for both subroutines. Again, the inductive properties will continue to hold.

In the third case, if edge e has an endpoint in an active component that is not X, then e will go tight immediately after X goes neutral for $\text{PD}(\lambda_1^+)$ and the components will remain the same, maintaining the inductive properties. Otherwise, one endpoint of e must be in X and the other endpoint of e is in an inactive component, and right after e goes tight for $\text{PD}(\lambda_1^-)$, the newly merged subset will have infinitesimally small remaining potential and will go inactive immediately. Again, this maintains the inductive properties.

Lastly, note that the time for a subset to go neutral has a negative slope in Λ_1 and the time for an edge to go tight has a positive slope in Λ_1. Since $\lambda_1^+ > \lambda_1^-$ and the y variables are infinitesimally different, the fourth case cannot occur. In all cases, the inductive properties continue to hold and the lemma holds. ◀

The proof of Lemma 7 exactly exhibits the differences between the two subroutines. First, there may be subsets that are neutral and marked inactive in $\text{PD}(\lambda_1^+)$ but have infinitesimally small potential in $\text{PD}(\lambda_1^-)$. Second, there may be pairs of edges that went tight between the same components. Lastly, there may be edges in $\text{PD}(\lambda_1^-)$ that do not exist in $\text{PD}(\lambda_1^+)$. However, these edges are between inactive components and components with infinitesimally small potential. Therefore, these edges will be pruned in $\text{PD}(\lambda_1^-)$ and will not contribute to the component of size $\geq \frac{1}{2}D$.

Since we assume we break ties by considering subsets before edges and lower weight edges first, $\text{PD}(\lambda_1)$ will behave the same as $\text{PD}(\lambda_1^+)$. Therefore, the largest component in T' when running $\text{PD}(\lambda_1)$ has cost $< \frac{1}{2}D$. However, we can think about reversing these ties one by one. In particular, consider breaking the first i ties according to $\text{PD}(\lambda_1^-)$ and then the rest by $\text{PD}(\lambda_1^+)$. By the analysis in Lemma 7, reversing these ties will not change the y variables or active components. The only difference will be going into the pruning phrase.

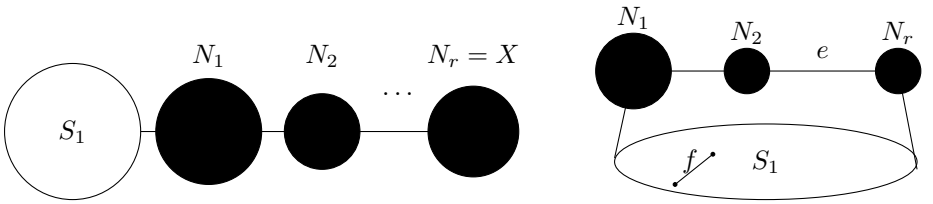
Thus, eventually we find the smallest k such that breaking the first k ties according to $\text{PD}(\lambda_1^-)$ yields a component of size $\geq \frac{1}{2}D$. In other words, we have either identified a neutral subset S such that marking S active rather than inactive changes the largest component to have size $\geq \frac{1}{2}D$ or we have identified two edges e and f that tie such that adding e instead of f changes the largest component to have size $\geq \frac{1}{2}D$. From here on, we assume that we always run $\text{PD}(\lambda_1)$ according to these tie-breaking rules.

5 Constructing a Tour

Let y be all of the dual variables for $\text{PD}(\lambda_1)$, let T' be the set of edges after the pruning phase, and let $\mathcal{S}$ be defined as before. Lastly, let $\pi(S)$ be the potential of $S \subseteq V$ given y. By construction, the largest component returned by $\text{PD}(\lambda_1)$ has size $\geq \frac{1}{2}D$. Recall from Section 4 that either

1. there exists a neutral subset $X \in \mathcal{S}$ such that if X is marked inactive then the largest component in T' has cost $< \frac{1}{2}D$ or
2. there exist tight edges $e \in T$ and $f \notin T$ such that if we swap e with f in T then the largest component has size $< \frac{1}{2}D$.

In the first case, when X is marked inactive, then a path of neutral subsets $N_1, N_2, \ldots, N_r = X$ is pruned yielding a component S_1 with cost $< \frac{1}{2}D$. Similarly, in the second case, having the edge e prevented some neutral subsets $N_1, N_2, \ldots, N_r$ from being pruned that had degree > 1. However, by removing e and replacing it with f, these subsets are pruned and we are left with component S_1 with cost $< \frac{1}{2}D$. See Figures 2a and 2b.



(a) Case 1: Marking X as inactive.

(b) Case 2: Replacing e with f.

Figure 2 Neutral subsets pruned in each case to yield component S_1 with cost $< \frac{1}{2}D$.

For both cases, we will use this threshold event to produce a tree T_A on a subset of vertices S_A of cost $\leq \frac{1}{2}D$. In doing so, we will also find another tree $\bar{T}$ on a subset of vertices $\bar{S}$ of cost $\geq \frac{1}{2}D$ such that $|S_A| \geq |\bar{S}| - 1$. Then, doubling T_A will yield a feasible tour F_A that visits almost as many vertices as in $\bar{S}$. The tree $\bar{T}$ will be helpful in obtaining a lower bound for $|S_A|$.

We start by setting T_A to be the edges in T' that span S_1. By construction, these edges have cost $< \frac{1}{2}D$. We will then try to grow T_A as much as possible along the path from S_1 to $N_1, N_2, \ldots, N_r$. First, suppose that we can add this full path and the edges that span each N_i to T_A without going over cost $\frac{1}{2}D$. Then, we set T_A to be this expanded tree and $S_A = S_1 \cup N_1 \cup \ldots \cup N_r$. Further, we set $\bar{T}$ to be the edges in T' in the corresponding component at the end of $\mathrm{PD}(\lambda_1)$. By construction, the cost of $\bar{T}$ is $\geq \frac{1}{2}D$ and $|S_A| \geq |\bar{S}|$.

Otherwise, we continue to add N_1, N_2, ... to our tree until we reach a component $\bar{X} \in \{N_1, N_2, \ldots, N_r\}$ such that adding the edges that span $\bar{X}$ to T_A implies that $\sum_{e \in T_A} c_e > \frac{1}{2}D$. In other words, we cannot add this whole subset to our tree without going over budget. Let $e = (u, v)$ be the edge that connects $\bar{X}$ to T_A in T'. If adding e to T_A already brings the cost of T_A strictly over $\frac{1}{2}D$, then we stop growing T_A and set $\bar{T} = T_A \cup \{e\}$. Otherwise, we add e to T_A and run a procedure $\mathbf{pick}(\bar{X}, v, \bar{T})$ that will pick a subset of the edges spanning $\bar{X}$ including v.

Specifically, the procedure $\mathbf{pick}(X, w, T_A)$ adds to T_A a set of edges in T' that span a subset of component X including w. We denote by $X_1, X_2 \in \mathcal{S}$ the two components that merged to form X and by $e' = (u, v)$ the edge that connects X_1 and X_2 in T'. Without loss of generality, $u \in X_1, v \in X_2$. Further, let T'_1 and T'_2 be the edges in T' with both endpoints in X_1 and X_2, respectively. See Figure 3.

If the total cost of edges in $T_A \cup T'_1$ is greater than $\frac{1}{2}D$, then we know we should only add edges in this subtree to T_A and we recursively invoke $\mathbf{pick}(X_1, w, T_A)$. If instead the total cost of edges in $T_A \cup T'_1 \cup \{e'\}$ is less than $\frac{1}{2}D$, then we can feasibly add all edges in T'_1 and e' without going over budget. Thus, the procedure adds all these edges to T_A and recursively invokes $\mathbf{pick}(X_2, v, T_A)$ to pick the remaining edges in T'_2. Finally, if the cost of edges in $T_A \cup T'_1$ is less than or equal to $\frac{1}{2}D$, but greater than $\frac{1}{2}D - c_{e'}$, then we cannot quite make it to T'_2 without going over budget. In this case, the procedure adds all edges in T'_1 to T_A and sets $\bar{T} = T_A \cup \{e'\}$.

At the end of the procedure, we produce a tree T_A of cost $\leq \frac{1}{2}D$ that spans a subset S_A along with a tree $\bar{T}$ of cost $\geq \frac{1}{2}D$ that spans a subset $\bar{S}$ where $|\bar{S}| \leq |S_A| + 1$. Further, if $|\bar{S}| = |S_A| + 1$, then $\bar{T}$ has cost $> \frac{1}{2}D$.

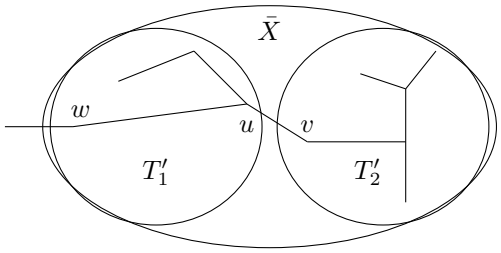


Figure 3 Illustration of the pick procedure.

5.1 Properties of $\bar{T}$

We have now constructed a tree T_A of cost $\leq \frac{1}{2}D$ that spans a subset S_A along with a tree $\bar{T}$ of cost $\geq \frac{1}{2}D$ that spans a subset $\bar{S}$ containing at most one more vertex than S_A. Further, if $|\bar{S}| = |S_A| + 1$, then $\bar{T}$ has cost $> \frac{1}{2}D$. We will use $\bar{T}$ to prove a bound on $|\bar{S}|$, which in turn will give a bound on $|S_A|$.

Let $Q \in \mathcal{S}$ be a subset containing $\bar{S}$. Since $\bar{S}$ is a subset of an active set, such a set will always exist. Our goal will be to show that

$$|S_A| \geq \frac{1}{2}\lambda_1 D + \pi(Q) - 1.$$

Let $\bar{v} = \bar{S} - S_A$ (possibly equal to $\emptyset$). We first state the following useful lemma. Since the proof closely resembles that of Goemans and Williamson [14] for the Prize-Collecting Steiner Tree Problem, we defer the proof to the full version of the paper.

▶ **Lemma 8.**

$$\sum_{e \in \bar{T}} \sum_{S : e \in \delta(S)} y_S \leq 2 \sum_{\substack{T : T \cap \bar{S} \neq \emptyset \\ \bar{v} \notin T}} y_T. \tag{1}$$

▶ **Theorem 9.** *Let Q be any set in $\mathcal{S}$ containing $\bar{S}$. Then,*

$$|S_A| > \frac{1}{2}\lambda_1 D + \pi(Q) - 1.$$

Proof. Vertices in $Q - \bar{S}$ are either in a neutral subset N (the combination of pruned subsets and N_i not reached) or are in the set $\bar{X} \in \{N_1, N_2, \ldots, N_r\}$ that we started our pick routine on. Let $\mathcal{S}_N$ be all subsets in $\mathcal{S}$ that are subsets of N. By the definition of neutral subsets,

$$|N| = 2 \sum_{T : T \in \mathcal{S}_N} y_T.$$

Similarly, let $\mathcal{S}_X$ be all subsets in $\mathcal{S}$ that are subsets of $\bar{X}$ and contain vertices in $\bar{X} - \bar{S}$. These are all the previously active subsets T such that $y_T > 0$ and T contains vertices in $\bar{X} - \bar{S}$ before the set $\bar{X}$ went neutral. Thus,

$$|\bar{X} - \bar{S}| \leq 2 \sum_{T : T \in \mathcal{S}_X} y_T.$$

Any subset in $\mathcal{S}$ that contains vertices in $\bar{S}$ and $\bar{X} - \bar{S}$ must contain v. Therefore, the only subsets of Q that are not in $\mathcal{S}_N$ or $\mathcal{S}_X$ are those that contain a subset of $\bar{S}$ but do not

contain $\bar{v}$. In other words,

$$|Q| = 2 \sum_{T:T\subseteq Q} y_T + \pi(Q)$$
$$\geq 2 \sum_{\substack{T:T\cap\bar{S}\neq\emptyset \\ \bar{v}\notin T}} y_T + 2 \sum_{T\in\mathcal{S}_N} y_T + 2 \sum_{T:T\in\mathcal{S}_X} y_T + \pi(Q) \quad \geq 2 \sum_{\substack{T:T\cap\bar{S}\neq\emptyset \\ \bar{v}\notin T}} y_T + |Q - \bar{S}| + \pi(Q)$$

Rearranging,

$$|\bar{S}| \geq 2 \sum_{\substack{T:T\cap\bar{S}\neq\emptyset \\ \bar{v}\notin T}} y_T + \pi(Q) \geq \sum_{e\in\bar{T}} \sum_{S:e\in\delta(S)} y_S + \pi(Q) = \lambda_1 \cdot \sum_{e\in\bar{T}} c_e + \pi(Q).$$

The second inequality follows from Lemma 8 and the third from property 4 of the algorithm. If $|\bar{S}| = |S_A|$, then we are done. Otherwise, suppose that $|\bar{S}| = |S_A|+1$. Then, $\sum_{e\in\bar{T}} c_e > \frac{1}{2}D$. In either case, the theorem holds. ◀

6 Approximation Ratio

The previous sections show that we can produce a feasible tour F_A on a subset S_A such that $|S_A| > \frac{1}{2}\lambda_1 D + \pi(Q) - 1$, where Q is the set in $\mathcal{S}$ of maximum potential that contains $\bar{S}$. Recall from Theorem 5, that for an optimal subset of vertices $O^\star$, $|O^\star| \leq \lambda_1 D + \pi(O)$, where O is the minimal subset in $\mathcal{S}^+$ that contains $O^\star$. Suppose that $\pi(Q) \geq \pi(O)$. In this case,

$$|S_A| + 1 > \frac{1}{2}[\lambda_1 D + \pi(O)] \geq \frac{1}{2}|O^\star|.$$

Without loss of generality, assume $|O^\star|$ is even (we can always make a copy of each vertex that has an edge of cost zero incident to the original). This implies that $|S_A| \geq \frac{1}{2}|O^\star|$.

On the other hand, suppose that $\pi(Q) < \pi(O)$. By the definition of Q, $Q \not\subseteq O$ since Q was the set of maximum potential that contained $\bar{S}$. Thus, either O is contained in a laminar set that is a strict subset of Q (and does not contain all vertices in $\bar{S}$) or O is disjoint from Q. By looking at the maximal sets with potential higher than $\pi(Q)$, we can recurse on each disjoint subgraph and return the best solution found. Overall, this shows that we can find a feasible tour F_A on a subset S_A such that $|S_A| \geq \frac{1}{2}|O^\star|$.

▶ **Theorem 10.** *The described algorithm is a 2-approximation for the budgeted prize-collecting traveling salesman problem.*

To see that this algorithm extends to the weighted version, imagine creating copies of each vertex v with zero cost edges to v. Since all these edges will go tight instantaneously in the primal-dual subroutine, we can actually just begin the algorithm with these weighted "clusters" as our initial active sets with potential equal to the weight of v.

7 Computational Experiments

In this section, we complete computational experiments in order to better understand the performance of our algorithm in practice. The primal-dual algorithm as detailed in this paper was implemented in C++11 using binary search to find λ_1. The experiments were conducted on a Dell R620 with two Intel 2.70GHz 8-core processors and 96GB of RAM.

The first set of graphs we used for the experiments are the 37 symmetric TSP instances with at most 400 nodes in the TSPLIB data set [20]. The second set of graphs are 37 weighted

Table 1 Graph statistics for each group of graphs averaged over all instances.

Instance Type	$\|V\|$	$\|E\|$	Total Vertex Weight
TSPLIB	158.14	15658.43	158.14
Bike	319.54	4634.77	1302.51

Table 2 Computational results of the primal-dual algorithm for each group of graphs and budget with results averaged over all instances.

Instance Type	f	Time (s)	# Recursions	% Opt. Gap	% Weight	% Budget
TSPLIB	0.25	74.16	0.59	46.67	33.06	77.38
TSPLIB	0.5	72.61	0.14	41.89	58.08	69.89
TSPLIB	0.75	71.24	0.22	18.62	81.38	68.80
Bike	0.25	25.15	0.28	45.74	43.37	66.90
Bike	0.5	33.21	0.28	25.89	74.01	67.13
Bike	0.75	30.46	0.05	8.29	91.68	67.37

instances constructed using the Citi Bike network of bikesharing stations in New York City. Each instance corresponds to a week of usage data at these stations, and the weight of a vertex corresponds to the number of broken docks at that station during that week. The number of broken docks was estimated from the usage data using a similar probabilistic method to that of Kaspi, Raviv, and Tzur [17]. Details about both types of constructed instances are given in Table 1.

For each test graph G, we first found an upper bound on the cost of a tour by computing 2 times the cost of a minimum spanning tree in G. We then set the budget for our tour to be $f = 25\%$, 50%, or 75% of this upper bound. W denotes the total weight of the vertices, for TSPLIB instances, the number of vertices. After finding our solution of weight A, we compute an upper bound on the weight of visited vertices $U = \min(\lambda_1 D + \max_{S \in \mathcal{S}} \pi(S), W)$ and record the percent optimality gap as $100 \times (U - A)/U$. Results are given in Table 2. Column 6 gives the percentage of the total weight W captured by the constructed tour, and Column 7 gives the percentage of the distance budget used after shortcutting the tree.

We report several interesting structural results. First, the average time seems to be heavily influenced by the number of edges; the bike instances were quicker to complete even though the average number of nodes was higher. However, the average time does not seem to grow with the budget (and hence with the size of the outputted solution) since most of the time is spent finding the value of Λ_1. The average optimality gap, on the other hand, does improve with the budget. This is likely due to the fact that for larger budgets the upper bound is given by W. Also of interest is that $\max_{S \in \mathcal{S}} \pi(S)$ contributed little to our upper bound U. As a result, our optimality gaps depend mostly on the value of Λ_1, rather than the potentials, and may be far from tight. However, the fact that on average we only use around $2/3$ of the distance budget implies that the solutions could be improved as well. To ensure that we use a larger part of the budget, we ran further experiments on the Citi Bike instances; in these, we ran binary search over possible *virtual budgets* in the input until finding one with which the resulting tour uses at least 90% of the actual budget. This reduced our optimality gaps from 45.74%, 25.89%, and 8.29% to 27.96%, 11.87%, and 0.17%, respectively. Lastly, it is interesting that the algorithm rarely ever needs to recurse on a subgraph.

8 Conclusion and Future Work

In this paper, we provide a 2-approximation algorithm for the budgeted prize-collecting traveling salesman problem that has at its base a classic primal-dual approach. The key insights are to use constructed potentials to evaluate potential subsets to tour and to identify the structure of a good tour. In particular, we construct a tree that closely follows the structure of the laminar collection of subsets with positive dual value. Further, we ensure this tree is just within budget in that adding one extra edge will make doubling the tree an infeasible tour. An obvious open question seeks to improve the approximation guarantee or prove the current guarantee is the best possible. Specifically, it would be interesting to know whether or not a (3/2)-approximation algorithm is possible given that that is the current best guarantee for the unconstrained traveling salesman problem. Another interesting direction would be to see if one can avoid recursing by inferring more from the potentials.

References

1 Aaron Archer, MohammadHossein Bateni, MohammadTaghi Hajiaghayi, and Howard Karloff. Improved approximation algorithms for prize-collecting Steiner tree and TSP. *SIAM Journal on Computing*, 40(2):309–332, 2011.

2 Sunil Arya and Hariharan Ramesh. A 2.5-factor approximation algorithm for the k-MST problem. *Information Processing Letters*, 65(3):117–118, 1998.

3 G. Ausiello, M. Demange, L. Laura, and V. Paschos. Algorithms for the on-line quota traveling salesman problem. *Information Processing Letters*, 92(2):89–94, 2004.

4 Avrim Blum, Ramamurthy Ravi, and Santosh Vempala. A constant-factor approximation algorithm for the k-MST problem. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing (STOC)*, pages 442–448. ACM, 1996.

5 Chandra Chekuri and Nitish Korula. Approximation algorithms for orienteering with time windows. *arXiv preprint arXiv:0711.4825*, 2007.

6 Chandra Chekuri, Nitish Korula, and Martin Pál. Improved algorithms for orienteering and related problems. *ACM Transactions on Algorithms (TALG)*, 8(3):23, 2012.

7 Chandra Chekuri and Martin Pal. A recursive greedy algorithm for walks in directed graphs. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 245–253. IEEE, 2005.

8 Ke Chen and Sariel Har-Peled. The orienteering problem in the plane revisited. In *Proceedings of the Twenty-Second Annual Symposium on Computational Geometry*, pages 247–254. ACM, 2006.

9 Joan Feigenbaum, Christos H. Papadimitriou, and Scott Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 63(1):21–41, 2001.

10 Greg N. Frederickson and Barry Wittman. Approximation algorithms for the traveling repairman and speeding deliveryman problems. *Algorithmica*, 62(3-4):1198–1221, 2012.

11 Daniel Freund, Ashkan Norouzi-Fard, Alice Paul, Shane G. Henderson, and David B. Shmoys. Data-driven rebalancing methods for bike-share systems. Working Paper, 2017.

12 N. Garg. A 3-approximation for the minimum tree spanning k vertices. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, FOCS'96, pages 302–, Washington, DC, USA, 1996. IEEE Computer Society.

13 Naveen Garg. Saving an epsilon: a 2-approximation for the k-MST problem in graphs. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing (STOC)*, pages 396–402. ACM, 2005.

14 Michel X. Goemans and David P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.

15 Anupam Gupta, Ravishankar Krishnaswamy, Viswanath Nagarajan, and R. Ravi. Approximation algorithms for stochastic orienteering. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1522–1538. SIAM, 2012.

16 David S. Johnson, Maria Minkoff, and Steven Phillips. The prize collecting Steiner tree problem: theory and practice. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 760–769. Society for Industrial and Applied Mathematics, 2000.

17 Mor Kaspi, Tal Raviv, and Michal Tzur. Detection of unusable bicycles in bike-sharing systems. *Omega*, 65:10–16, 2016.

18 Asaf Levin. A better approximation algorithm for the budget prize collecting tree problem. *Operations Research Letters*, 32(4):316–319, 2004.

19 Viswanath Nagarajan and R. Ravi. Approximation algorithms for distance constrained vehicle routing problems. *Networks*, 59(2):209–214, 2012.

20 Gerhard Reinelt. TSPLIB – a traveling salesman problem library. *ORSA Journal on Computing*, pages 376–384, 1991.

21 Shalabh Vidyarthi and Kaushal K. Shukla. Approximation algorithms for P2P orienteering and stochastic vehicle routing problem. *arXiv preprint arXiv:1501.06515*, 2015.

Counting Restricted Homomorphisms via Möbius Inversion over Matroid Lattices*†

Marc Roth

Saarland University and Cluster of Excellence (MMCI), Saarbrücken, Germany
mroth@mmci.uni-saarland.de

Abstract

We present a framework for the complexity classification of parameterized counting problems that can be formulated as the summation over the numbers of homomorphisms from small pattern graphs $H_1, \ldots, H_\ell$ to a big host graph G with the restriction that the coefficients correspond to evaluations of the Möbius function over the lattice of a graphic matroid. This generalizes the idea of Curticapean, Dell and Marx [STOC 17] who used a result of Lovász stating that the number of subgraph embeddings from a graph H to a graph G can be expressed as such a sum over the lattice of partitions of H.

In the first step we introduce what we call graphically restricted homomorphisms that, inter alia, generalize subgraph embeddings as well as locally injective homomorphisms. We provide a complete parameterized complexity dichotomy for counting such homomorphisms, that is, we identify classes of patterns for which the problem is fixed-parameter tractable (FPT), including an algorithm, and prove that all other pattern classes lead to #W[1]-hard problems. The main ingredients of the proof are the complexity classification of linear combinations of homomorphisms due to Curticapean, Dell and Marx [STOC 17] as well as a corollary of Rota's NBC Theorem which states that the sign of the Möbius function over a geometric lattice only depends on the rank of its arguments.

We apply the general theorem to the problem of counting locally injective homomorphisms from small pattern graphs to big host graphs yielding a concrete dichotomy criterion. It turns out that – in contrast to subgraph embeddings – counting locally injective homomorphisms has "real" FPT cases, that is, cases that are fixed-parameter tractable but not polynomial time solvable under standard complexity assumptions. To prove this we show in an intermediate step that the subgraph counting problem remains #P-hard when both the pattern and the host graphs are restricted to be trees. We then investigate the more general problem of counting homomorphisms that are injective in the r-neighborhood of every vertex. As those are graphically restricted as well, they can also easily be classified via the general theorem.

Finally we show that the dichotomy for counting graphically restricted homomorphisms readily extends to so-called linear combinations.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Counting Problems, G.2.2 Graph Theory

Keywords and phrases homomorphisms, matroids, counting complexity, parameterized complexity, dichotomy theorems

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.63

* A full version of the paper is available at https://arxiv.org/abs/1706.08414.
† Part of this work was done while the author was visiting the Simons Institute for the Theory of Computing

25th Annual European Symposium on Algorithms (ESA 2017).
Editors: Kirk Pruhs and Christian Sohler; Article No. 63; pp. 63:1–63:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In his seminal work about the complexity of computing the permanent Valiant [29] introduced counting complexity which has since then evolved into a well-studied subfield of computational complexity. Despite some surprising positive results like polynomial time algorithms for counting perfect matchings in planar graphs by the FKT method [27, 17], counting spanning trees by Kirchhoff's Matrix Tree Theorem or counting Eulerian cycles in directed graphs using the "BEST"-Theorem (see e.g. [2]), most of the interesting problems turned out to be intractable. Therefore, several relaxations such as restrictions of input classes [33] and approximate counting [16, 11] were introduced. Another possible relaxation, the one this work deals with, is to consider parameterized counting problems as introduced by Flum and Grohe [14]. Here, problems come with an additional parameter k and a problem is fixed-parameter tractable (FPT) if it can be solved in time $g(k) \cdot \text{poly}(n)$ where n is the input size and g is a computable function, which yields fast algorithms for large instances with small parameters. On the other hand, a problem is considered intractable if it is #W[1]-hard. This stems from the fact that #W[1]-hard problems do not allow an FPT algorithm unless standard assumptions such as the exponential time hypothesis (ETH) are wrong.

When investigating a family of related (counting) problems one could aim to simultaneously solve the complexity of as many problems as possible, rather than tackling a (possibly infinite) number of problems by hand. For example, instead of proving that counting paths in a graph is hard, then proving that counting cycles is hard and then proving that counting stars is easy, one should, if possible, find a criterion that allows a classification of those problems in hard and easy cases. Unfortunately, there are results like Ladner's Theorem [18], stating that there are problems neither in P nor NP-hard (assuming P $\neq$ NP), which give a negative answer to that goal in general. However, there are families of problems that have enough structure to allow so-called dichotomy results. One famous example, and to the best of the authors knowledge this was the first such result, is Schaefer's dichotomy [25], stating that every instance of the generalized satisfiability problem is either polynomial time solvable or NP-complete. Since then much work has been done to generalize this result, culminating in recent announcements ([3],[34],[23]) of a proof of the Feder-Vardi-Conjecture [12]. This question was open for almost twenty years and indicates the difficulty of proving such dichotomy results, at least for decision problems. In counting complexity, however, it seems that obtaining such results is less cumbersome. One reason for this is the existence of some powerful techniques like polynomial interpolation [28], the Holant framework [30, 31, 4] as well as the principle of inclusion-exclusion which all have been used to establish very revealing dichotomy results such as [5, 9].

Examples of dichotomies in parameterized counting complexity are the complete classifications of the homomorphism counting problem due to Dalmau and Jonsson [10][1] and the subgraph counting problem due to Curticapean and Marx [9]. For the latter, one is given graphs H and G and wants to count the number of subgraphs of G isomorphic to H, parameterized by the size of H. It is known that this problem is polynomial time solvable if there is a constant upper bound on the size of the largest matching of H and #W[1]-hard otherwise[2]. The first step in this proof was the hardness result of counting matchings of size k of Curticapean [6], which turned out to be the "bottleneck" problem and was then reduced to the general problem.

[1] Ultimately, the results of [7] and this work rely on the dichotomy for counting homomorphisms.

[2] On the other hand the complexity of the decision version of this problem, that is, finding a subgraph of G isomorphic to H, is still unresolved. Only recently it was shown in a major breakthrough that finding bicliques is hard [19].

This approach, first finding the hard obstructions and then reducing to the general case, seemed to be the canonical way to tackle such problems. However, recently Curticapean, Dell and Marx [7] discovered that a result of Lovász [20] implies the existence of parameterized reductions that, inter alia, allow a far easier proof of the general subgraph counting problem. Lovász result states that, given simple graphs H and G, it holds that

$$\#\mathsf{Emb}(H,G) = \sum_{\rho \geq \emptyset} \mu(\emptyset,\rho) \cdot \#\mathsf{Hom}(H/\rho, G)\,, \tag{1}$$

where the sum is over the elements of the partition lattice of $V(H)$, $\mathsf{Emb}(H,G)$ is the set of embeddings[3] from H to G and $\mathsf{Hom}(H/\rho,G)$ is the set of homomorphisms from the graph H/ρ obtained from H by identifying vertices along ρ to G. Furthermore μ is the Möbius function. In their work Curticapean, Dell and Marx showed in a general theorem that a summation $\sum_{i=1}^{\ell} c_i \cdot \#\mathsf{Hom}(H_i,G)$ for *pairwise non-isomorphic graphs* H_i is #W[1]-hard if there is no upper bound on the treewidth of the pattern graphs H_i and fixed-parameter tractable otherwise, using a dichotomy for counting homomorphisms due to Dalmau and Jonsson [10]. Having this, one only has to show two properties of (1) to obtain the dichotomy for #Emb. First, one has to show that a high matching number of H implies that one of the graphs H/ρ has high treewidth and second, that two (or more) terms with high treewidth and isomorphic graphs H/ρ and H/σ do not cancel out (note that the Möbius function can be negative). As there is a closed form for the Möbius function over the partition lattice it was possible to show that whenever H/ρ and H/σ are isomorphic the sign of the Möbius function is equal.

1.1 Our results

The motivation of this work is the question whether the result of Curticapean, Dell and Marx can be generalized to construct a framework for the complexity classification of counting problems that can be expressed as the summation over homomorphisms and it turns out that this is possible whenever the summation is over a the lattice of a graphic matroid and the coefficients are evaluations of the Möbius function over the lattice, capturing not only embeddings but also locally injective homomorphisms.

In Section 3 we introduce what we call *graphically restricted homomorphisms*: Intuitively, a graphical restriction $\tau(H)$ of a graph H is a set of forbidden binary vertex identifications of H, modeled as a graph with vertex set $V(H)$ and edges along the binary constraints. We write τ-$\mathcal{M}(H)$ as the set of all graphs obtained from H by contracting vertices along edges in $\tau(H)$ and deleting multiedges, excluding those that contain selfloops. Now a graphically restricted homomorphism from H to G with respect to τ is a homomorphism from H to G that maps every pair of vertices $u,v \in V(H)$ that are adjacent in $\tau(H)$ to different vertices in G. We write $\mathsf{Hom}_\tau(H,G)$ for the set of all graphically restricted homomorphisms w.r.t. τ from H to G and provide a complete complexity classification for counting graphically restricted homomorphisms:

▶ **Theorem 1** (Intuitive version). *Computing* $\#\mathsf{Hom}_\tau(H,G)$ *is fixed-parameter tractable when parameterized by* $|V(H)|$ *if the treewidth of every graph in* τ-$\mathcal{M}(H)$ *is small. Otherwise the problem is* #W[1]*-hard.*

[3] Note that embeddings and subgraphs are equal up to automorphisms, that is, counting embeddings and counting subgraphs are essentially the same problem.

In particular, we obtain the following algorithmic result:

▶ **Theorem 2.** *There exists a deterministic algorithm that computes* $\#\mathsf{Hom}_\tau(H,G)$ *in time* $g(|V(H)|)\cdot|V(G)|^{\mathsf{tw}(\tau\text{-}\mathcal{M}(H))+1}$*, where* g *is a computable function and* $\mathsf{tw}(\tau\text{-}\mathcal{M}(H))$ *is the maximum treewidth of every graph in* $\tau\text{-}\mathcal{M}(H)$*.*

Having established the general dichotomy we observe that there exist graphical restrictions τ_{clique} and τ_{Li} such that $\mathsf{Hom}_{\tau_{\mathsf{clique}}}(H,G)$ is the set of all subgraph embeddings from H to G and $\mathsf{Hom}_{\tau_{\mathsf{Li}}}(H,G)$ is the set of all *locally injective homomorphisms* from H to G.

As a consequence we obtain a full complexity dichotomy for counting locally injective homomorphisms from small pattern graphs H to big host graphs G. To the best of the author's knowledge, this is the first result about the complexity of counting locally injective homomorphisms.

▶ **Corollary 3** (Intuitive version)**.** *Computing the number of locally injective homomorphisms from* H *to* G *is fixed-parameter tractable when parameterized by* $|V(H)|$ *if the treewidth of every graph in* $\tau_{\mathsf{Li}}\text{-}\mathcal{M}(H)$ *is small. Otherwise the problem is* #W[1]*-hard.*

Furthermore, there exists a deterministic algorithm that computes this number in time $g(|V(H)|)\cdot|V(G)|^{\mathsf{tw}(\tau_{\mathsf{Li}}\text{-}\mathcal{M}(H))+1}$*, where* g *is a computable function and* $\mathsf{tw}(\tau_{\mathsf{Li}}\text{-}\mathcal{M}(H))$ *is the maximum treewidth of every graph in* $\tau_{\mathsf{Li}}\text{-}\mathcal{M}(H)$*.*

We then observe that – in contrast to subgraph embeddings – counting locally injective homomorphisms has "real" FPT cases, that is, cases that are fixed-parameter tractable but not polynomial time solvable under standard assumptions. We show this by restricting the pattern graph to be a tree:

▶ **Corollary 4.** *Computing the number of locally injective homomorphisms from a tree* T *to a graph* G *can be done in deterministic time* $g(|V(T)|)\cdot|V(G)|^2$*, that is, the problem is fixed-parameter tractable when parameterized by* $|V(T)|$*. On the other hand, the problem is* #P*-hard.*

To prove #P-hardness, we prove in an intermediate step that the subgraph counting problem remains hard when both graphs are restricted to be trees, which may be of independent interest:

▶ **Lemma 5.** *The problem of, given trees* T_1 *and* T_2*, computing the number of subtrees of* T_2 *that are isomorphic to* T_1 *is* #P*-hard.*

After that we generalize locally injective homomorphisms to homomorphisms that are injective in the r-neighborhood of every vertex and observe that those are also graphically restricted and consequently obtain a counting dichotomy as well. Due to space constraints, the corresponding section is deferred to the full version of the paper.

Finally, it turns out that all results can easily be extended to so-called *linear combinations* of graphically restricted homomorphisms. Here one gets as input graphs $H_1,\dots,H_\ell$ together with positive coefficients $c_1,\dots,c_\ell$ and a graph G and the goal is to compute

$$\sum_{i=1}^{\ell} c_i\cdot\#\mathsf{Hom}_{\tau_i}(H_i,G)\,,$$

for graphical restrictions $\tau_1,\dots,\tau_\ell$. This generalizes for example problems like counting all trees of size k in G or counting all locally injective homomorphisms from all graphs of size k to G or a combination thereof. We find out that, under some conditions, the dichotomy criteria transfer immediately to linear combinations:

▶ **Theorem 6** (Intuitive version). *Computing* $\sum_{i=1}^{\ell} c_i \cdot \#\mathsf{Hom}_{\tau_i}(H_i, G)$ *is fixed-parameter tractable when parameterized by* $\max_i\{|V(H_i)|\}$ *if the maximum treewidth of every graph in* $\bigcup_i^{\ell} \tau_i\text{-}\mathcal{M}(H_i)$ *is small. Otherwise, if additionally* $|V(H_i)|$ *has the same parity for every* $i \in [\ell]$*, the problem is* #W[1]*-hard.*

Furthermore we observe that this theorem is not true on the #W[1]-hardness side if we omit the parity condition. Due to space constraints, the section dealing with linear combinations is deferred to the full version as well.

1.2 Techniques

The main ingredients of the proofs of Theorem 1 and Theorem 2 are the complexity classification of linear combinations of homomorphisms due to Curticapean, Dell and Marx (see Lemma 3.5 and Lemma 3.8 in [7]) as well as a corollary of Rota's NBC Theorem (see e.g. Theorem 4 in [24]). In the first step we prove the following identity for the number of graphically restricted homomorphisms via Möbius inversion:

$$\#\mathsf{Hom}_{\tau}(H, G) = \sum_{\rho \geq \emptyset} \mu(\emptyset, \rho) \cdot \#\mathsf{Hom}(H/\rho, G)\,,$$

where the sum is over elements of the lattice of flats of the graphical matroid given by $\tau(H)$ and H/ρ is the graph obtained by contracting the vertices of H along the flat ρ. After that we use Rota's Theorem to prove that none of the terms cancel out[4], despite the fact that the Möbius function can be negative. More precisely we show that whenever $H/\rho \cong H/\sigma$, we have that $\mathsf{rk}(\rho) = \mathsf{rk}(\sigma)$ and therefore, by Rota's Theorem, $\mathsf{sgn}(\mu(\emptyset, \rho)) = \mathsf{sgn}(\mu(\emptyset, \sigma))$.

The dichotomies for locally injective homomorphisms and homomorphisms that are injective in the r-neighborhood of every vertex are mere applications of the general theorem. For #P-hardness of the subgraph counting problem restricted to trees, we adapt the idea of the "skeleton graph" by Goldberg and Jerrum [15] and reduce directly from computing the permanent. To transfer this result to locally injective homomorphisms we use the well-known observation that locally injective homomorphisms from a tree to a tree are embeddings.

Finally, we prove the dichotomy for linear combinations of graphically restricted homomorphisms by taking a closer look at the proof of Theorem 1. Here, the parity constraint of the vertices of the graphs in the linear combination assures that there are no graphs H_i and H_j and elements ρ_i and ρ_j of the matroid lattices of $\tau_i(H_i)$ and $\tau_j(H_j)$ such that H_i/ρ_i and H_j/ρ_j are isomorphic but ρ_i and ρ_j have ranks of different parities. Using this observation, Theorem 6 can be proven in the same spirit as Theorem 1.

2 Preliminaries

First we will introduce some basic notions: Given a finite set S, we write $|S|$ or $\#S$ for the cardinality of S. Given a natural number ℓ we let $[\ell]$ be the set $\{1, \ldots, \ell\}$. Given a real number r we define the *sign* $\mathsf{sgn}(r)$ of r to be 1 if $r > 0$, 0 if $r = 0$ and -1 if $r < 0$.

A *poset* is a pair $(P, \leq)$ where P is a set and $\leq$ is a binary relation on P that is reflexive, transitive and anti-symmetric. Throughout this paper we will write $y \geq x$ if $x \leq y$. A *lattice* is a poset $(L, \leq)$ such that every pair of elements $x, y \in L$ has a *least upper bound* $x \vee y$ and a *greatest lower bound* $x \wedge y$ that satisfy:

[4] Here "cancel out" means that it could be possible that H/ρ and H/σ are isomorphic, but $\mu(\emptyset, \rho) = -\mu(\emptyset, \sigma)$ and all other H/ρ' are not isomorphic to H/ρ. In this case, the term $\#\mathsf{Hom}(H/\rho, G)$ would vanish in the above identity.

- $x \vee y \geq x$, $x \vee y \geq y$ and for all z such that $z \geq x$ and $z \geq y$ it holds that $z \geq x \vee y$.
- $x \wedge y \leq x$, $x \wedge y \leq y$ and for all z such that $z \leq x$ and $z \leq y$ it holds that $z \leq x \wedge y$.

Given a finite set S, a *partition* of S is a set ρ of pairwise disjoint subsets of S such that $\dot{\bigcup}_{s\in\rho} s = S$. We call the elements of ρ *blocks*. For two partitions ρ and σ we write $\rho \leq \sigma$ if every element of ρ is a subset of some element of σ. This binary relation is a lattice and called the *partition lattice* of S. We will in particular encounter lattices of graphic matroids in our proofs.

2.1 Matroids

We will follow the definitions of Chapt. 1 of the textbook of Oxley [22].

▶ **Definition 7.** A *matroid* M is a pair $(E, \mathcal{I})$ where E is a finite set and $\mathcal{I} \subseteq \mathcal{P}(E)$ such that
(1) $\emptyset \in \mathcal{I}$,
(2) if $A \in \mathcal{I}$ and $B \subseteq A$ then $B \in \mathcal{I}$, and
(3) if $A, B \in \mathcal{I}$ and $|B| < |A|$ then there exists $a \in A \setminus B$ such that $B \cup \{a\} \in \mathcal{I}$.
We call E the *ground set* and an element $A \in \mathcal{I}$ an *independent set*. A maximal independent set is called a *basis*. The *rank* $\mathsf{rk}(M)$ of M is the size of its bases[5].

Given a subset $X \subseteq E$ we define $\mathcal{I}|X := \{A \subseteq X \mid A \in \mathcal{I}\}$. Then $M|X := (X, \mathcal{I}|X)$ is also a matroid and called the restriction of M to X. Now the *rank* $\mathsf{rk}(X)$ of X is the rank of $M|X$. Equivalently, the rank of X is the size of the largest independent set $A \subseteq X$.

Furthermore we define the *closure* of X as follows:

$$\mathsf{cl}(X) := \{e \in E \mid \mathsf{rk}(X \cup \{e\}) = \mathsf{rk}(X)\}\,.$$

Note that by definition $\mathsf{rk}(X) = \mathsf{rk}(\mathsf{cl}(X))$. We say that X is a *flat* if $\mathsf{cl}(X) = X$. We denote $L(M)$ as the set of flats of M. It holds that $L(M)$ together with the relation of inclusion is a lattice, called the *lattice of flats* of M. The least upper bound of two flats X and Y is $\mathsf{cl}(X \cup Y)$ and the greatest lower bound is $X \cap Y$. It is known that the lattices of flats of matroids are exactly the geometric lattices[6] and we denote the set of those lattices as $\mathcal{L}$.

In Section 3 we take a closer look at (lattices of flats of) graphic matroids:

▶ **Definition 8.** Given a graph $H = (V, E) \in \mathcal{G}$, the *graphic matroid* $M(H)$ has ground set E and a set of edges is independent if and only if it does not contain a cycle.

If H is connected then a basis of H is a spanning tree of H. If H consists of several connected components then a basis of $M(H)$ induces spanning trees for each of those. Every subset X of E induces a partition of the vertices of H where the blocks are the vertices of the connected components of $H|_X$ and it holds that

$$\mathsf{rk}(X) = |V(H)| - c(H|_X)\,. \tag{2}$$

In particular, the flats of $M(H)$ correspond bijectively to the partitions of vertices of H into connected components as adding an element to X such that the rank does not change will not change the connected components, too. For convenience we will therefore abuse notation

[5] This is well-defined as every maximal independent set has the same size due to (3).
[6] For the purpose of this paper we do not need the definition of geometric lattices but rather the equivalent one in terms of lattices of flats and therefore omit it. We recommend e.g. Chapt. 3 of [32] and Chapt. 1.7 of [22] to the interested reader.

and say, given an element ρ of the lattice of flats of $M(H)$, that ρ partitions the vertices of H where the blocks are the vertices of the connected components of $H|_\rho$. The following observation will be useful in Section 3:

▶ **Lemma 9.** *Let $\rho, \sigma \in L(M(H))$ for a graph $H \in \mathcal{G}$. If the number of blocks of ρ and σ are equal then $\mathsf{rk}(\rho) = \mathsf{rk}(\sigma)$.*

Proof. Immediately follows from Equation (2). ◀

We denote H/ρ as the graph obtained from H by contracting the vertices of H that are in the same component of ρ and deleting multiedges (but keeping selfloops). As the vertices of H/ρ partition the vertices of H, we think of the vertices of H/ρ as subsets of vertices of H and call them *blocks*. Furthermore we write $[v]$ for the block containing v.

2.2 Graphs and homomorphisms

In this work all graphs are considered unlabeled and simple but may allow selfloops unless stated otherwise. We denote the set of all those graphs as $\mathcal{G}^\circ$. Furthermore we denote $\mathcal{G}$ as the set of all unlabeled and simple graphs without selfloops.

For a graph G we write n for the number of vertices $V(G)$ of G and m for the number of edges $E(G)$ of G. We denote $c(G)$ as the number of connected components of G. Furthermore, given a subset X of edges, we denote $G|_X$ as the graph with vertices $V(G)$ and edges X. Given a partition of vertices ρ of a graph H, we write H/ρ as the graph obtained from H by contracting the vertices of H that are in the same component of ρ and deleting multiedges (but keeping selfloops). As the vertices of H/ρ partition the vertices of H, we think of the vertices as subsets of vertices of H and call them *blocks*. Furthermore we write $[v]$ for the block containing v.

Given graphs H and G, a *homomorphism* from H to G is a mapping $\varphi : V(H) \to V(G)$ such that $\{u, v\} \in E(H)$ implies that $\{\varphi(u), \varphi(v)\} \in E(G)$. We denote $\mathsf{Hom}(H, G)$ as the set of all homomorphisms from H to G. A homomorphism is called *embedding* if it is injective and we denote $\mathsf{Emb}(H, G)$ as the set of all embeddings from H to G. An embedding from H to H is called an *automorphism* of H. We denote $\mathsf{Aut}(H)$ as the set of all automorphisms of H. Furthermore we let $\mathsf{Sub}(H, G)$ be the set of all subgraphs of G that are isomorphic to H. Then it holds that $\#\mathsf{Aut}(H) \cdot \#\mathsf{Sub}(H, G) = \#\mathsf{Emb}(H, G)$ (see e.g. [20]).

Given a set S and a function $\alpha : S \to \mathbb{Q}$, we define the *support* of α as follows:

$$\mathsf{supp}(\alpha) := \{s \in S \mid \alpha(s) \neq 0\}.$$

A graph parameter that will be of quite some importance to define the dichotomy criteria is the *treewidth* of a graph, capturing how "tree-like" a graph is. We do not need the explicit definition of treewidth which is therefore, together with some examples, deferred to the full version. However, throughout this paper we will often say that a set C of graphs has *bounded treewidth* meaning that there is a constant B such that the treewidth of every graph $H \in C$ is bounded by B.

2.3 Parameterized counting

We will mainly follow the definitions of Chapt. 14 of the textbook of Flum and Grohe [14]. A *parameterized counting problem* is a function $F : \{0,1\}^* \to \mathbb{N}$ together with a polynomial-time computable *parameterization* $k : \{0,1\}^* \to \mathbb{N}$. A parameterized counting problem is *fixed-parameter tractable* if there exists a computable function g such that it can be solved

in time $g(k(x)) \cdot |x|^{O(1)}$ for any input x. A *parameterized Turing reduction* from (F, k) to (F', k') is an FPT algorithm w.r.t. parameterization k with oracle (F', k') that on input x computes $F(x)$ and additionally satisfies that there exists a function g' such that for every oracle query y it holds that $k'(y) \leq g(k(x))$. A parameterized counting problem (F, k) is #W[1]-hard if there exists a parameterized Turing reduction from $\#k$-clique to (F, k), where $\#k$-clique is the problem of, given a graph G and a parameter k, computing the number of cliques of size k in G[7]. Under standard assumptions (e.g. under the exponential time hypothesis) #W[1]-hard problems are not fixed-parameter tractable.
The following two parameterized counting problems will be of particular importance in this work: Given a class of graphs $C \subseteq \mathcal{G}$, $\#\mathsf{Hom}(C)$ ($\#\mathsf{Emb}(C)$) is the problem of, given a graph $H \in C$ and a graph $G \in \mathcal{G}$, computing $\#\mathsf{Hom}(H, G)$ ($\#\mathsf{Emb}(H, G)$). Both problems are parameterized by $\#V(H)$. Their complexity has already been classified:

▶ **Theorem 10** ([10]). *Let C be a recursively enumerable class of graphs. If C has bounded treewdith then $\#\mathsf{Hom}(C)$ can be solved in polynomial time. Otherwise $\#\mathsf{Hom}(C)$ is* #W[1]*-hard.*

▶ **Theorem 11** ([9]). *Let C be a recursively enumerable class of graphs. If C has bounded matching number then $\#\mathsf{Emb}(C)$ can be solved in polynomial time. Otherwise $\#\mathsf{Emb}(C)$ is* #W[1]*-hard.*

Recall that "bounded treewidth (matching number)" means that there is a constant B such that the treewidth (size of the largest matching) of any graph in C is bounded by B.

2.4 Linear combinations of homomorphisms and Möbius inversion

Curticapean, Dell and Marx [7] introduced the following parameterized counting problem:

▶ **Definition 12** (Linear combinations of homomorphisms). Let $\mathcal{A}$ be a set of functions $a : \mathcal{G} \to \mathbb{Q}$ with finite support[8]. We define the parameterized counting problem $\#\mathsf{Hom}(\mathcal{A})$ as follows:

Given $a \in \mathcal{A}$ and $G \in \mathcal{G}$, compute

$$\sum_{H \in \mathrm{supp}(a)} a(H) \cdot \#\mathsf{Hom}(H, G)\,, \quad \text{parameterized by } \max_{H \in \mathrm{supp}(a)} \#V(H)\,.$$

Note that this problem generalizes $\#\mathsf{Hom}(C)$. The following theorem will be the foundation of all complexity results in this paper:

▶ **Theorem 13** ([7], Lemma 3.5 and Lemma 3.8). *If $\mathcal{A}$ has bounded treewidth then $\#\mathsf{Hom}(\mathcal{A})$ can be solved in time $g(|\mathrm{supp}(\alpha)|) \cdot n^{O(1)}$ on input (α, G) where $n = |V(G)|$ and g is a computable function. Otherwise the problem is* #W[1]*-hard.*

In their paper, the authors show how this result can be used to give a much simpler proof of Theorem 11. The idea is that every problem $\#\mathsf{Emb}(C)$ is equivalent to a problem $\#\mathsf{Hom}(\mathcal{A})$. As all proofs in this work are in the same flavour, we will outline the technique here, using $\#\mathsf{Emb}(C)$ as an example. Therefore, we first need to introduce the so called Möbius inversion (we recommend reading [26] for a more detailed introduction):

[7] For a more detailed introduction to #W[1] we recommend [14] to the interested reader.
[8] We can also think of $\mathcal{A}$ being a set of lists.

▶ **Definition 14.** Let $(P, \leq)$ be a poset and $h : P \to \mathbb{C}$ be a function. Then the *zeta transformation* ζh is defined as follows:

$$\zeta h(\sigma) := \sum_{\rho \geq \sigma} h(\rho) \,.$$

▶ **Theorem 15** (Möbius inversion, see [26] or [24])**.** *Let $(P, \leq)$ and h as in Definition 14. Then there is a function $\mu_P : P \times P \to \mathbb{Z}$ such that for all $\sigma \in P$ it holds that*

$$h(\sigma) = \sum_{\rho \geq \sigma} \mu_P(\sigma, \rho) \cdot \zeta h(\rho) \,.$$

μ_P is called the Möbius function*.*

The following identity is due to Lovász [20]:

$$\#\mathsf{Hom}(H/\sigma, G) = \sum_{\rho \geq \sigma} \#\mathsf{Emb}(H/\rho, G) \,,$$

where σ and ρ are partitions of vertices of H and $\geq$ is the partition lattice of H. Now Möbius inversion yields the following identity [20]:

$$\#\mathsf{Emb}(H, G) = \sum_{\rho \geq \emptyset} \mu(\emptyset, \rho) \cdot \#\mathsf{Hom}(H/\rho, G) \,,$$

where μ is the Möbius function over the partition lattice. Therefore, for every class of graphs C, there is a family of functions with finite support $\mathcal{A}$ such that $\#\mathsf{Emb}(C)$ and $\#\mathsf{Hom}(\mathcal{A})$ are the same problems. Now Curticapean, Dell and Marx show that C has unbounded matching number if and only if $\mathcal{A}$ has unbounded treewidth. The critical point in this proof was to show that the sign of $\mu(\emptyset, \rho)$ only depends on the number of blocks of ρ, which implies that for two isomorphic graphs H_1 and H_2, the terms $\#\mathsf{Hom}(H_1, G)$ and $\#\mathsf{Hom}(H_2, G)$ have the same sign in the above identity and therefore do not cancel out in the homomorphism basis. As there is a closed form for $\mu(\emptyset, \rho)$[9], the information about the sign could easily be extracted.

The motivation of this work is the question whether this can be made more general and it turns out that a corollary of Rota's NBC Theorem [24] (see also [1]) captures exactly what we need:

▶ **Theorem 16** (See e.g. Theorem 4 in [24])**.** *Let L be a geometric lattice with unique minimal element $\perp$ and let ρ be an element of L. Then it holds that*

$$\mathsf{sgn}(\mu_L(\perp, \rho)) = (-1)^{\mathsf{rk}(\rho)} \,.$$

In the following we will show that combining Rota's Theorem and the dichotomy for counting linear combinations of homomorphisms yields complete complexity classifications for the problems of counting those restricted homomorphisms that induce a Möbius inversion over the lattice of a graphic matroid, which are known to be geometric, when transformed into the homomorphism basis. Those include embeddings as well as locally injective homomorphisms.

[9] Here it is crucial that μ is the Möbius function over the (complete) partition lattice.

3 Graphically restricted homomorphisms

In the following we write $\emptyset$ for the minimal element of a matroid lattice.

▶ **Definition 17.** A *graphical restriction* is a computable mapping τ that maps a graph $H \in \mathcal{G}$ to a graph $H' \in \mathcal{G}$ such that $V(H) = V(H')$, that is, τ only modifies edges of H. We denote the set of all graphical restrictions as T. Given graphs H and G and a graphical restriction τ, we define the set of *graphically restricted homomorphisms* w.r.t. τ from H to G as follows:

$$\mathsf{Hom}_\tau(H,G) := \{\varphi \in \mathsf{Hom}(H,G) \mid \forall u,v \in V(H) : \{u,v\} \in E(\tau(H)) \Rightarrow \varphi(u) \neq \varphi(v)\}\,.$$

Given a recursively enumerable class of graphs $C \subseteq \mathcal{G}$, we define the parameterized counting problem $\#\mathsf{Hom}_\tau(C)$ as follows: Given a graph $H \in C$ and a graph $G \in \mathcal{G}$, we parameterize by $|V(H)|$ and wish to compute $\#\mathsf{Hom}_\tau(H,G)$.

Assume for example that τ_{clique} maps a graph H to the complete graph with vertices $V(H)$. Then one can easily verify that $\mathsf{Hom}_{\tau_{\mathsf{clique}}}(H,G) = \mathsf{Emb}(H,G)$.

The following lemma is an application of Möbius inversion (and slightly generalizes [20]). Due to space constraints, the proof is deferred to the full version.

▶ **Lemma 18.** *Let τ be a graphical restriction. Then for all graphs $H \in \mathcal{G}^\circ$ and $G \in \mathcal{G}$ it holds that*

$$\#\mathsf{Hom}_\tau(H,G) = \sum_{\rho \geq \emptyset} \mu(\emptyset,\rho) \cdot \#\mathsf{Hom}(H/\rho, G)\,, \tag{3}$$

where $\leq$ and μ are the relation and the Möbius function of the lattice $L(M(\tau(H)))$.

Intuitively, we will now show that counting graphically restricted homomorphisms from H to G is hard if we can "glue" vertices of H together along edges of $\tau(H)$ such that the resulting graph has no selfloops and high treewidth. We will capture this intuition formally:

▶ **Definition 19.** Let $H \in \mathcal{G}$ be a graph and let τ be a graphical restriction. A graph $H' \in \mathcal{G}^\circ$ obtained from H by contracting pairs of vertices u and v such that $\{u,v\} \in E(\tau(H))$ and deleting multiedges (but keeping selfloops) is called a *τ-contraction* of H. If additionally $H' \in \mathcal{G}$, that is, the contraction did not yield selfloops, we call H' a *τ-minor* of H. We denote the set of all τ-minors of H as $\tau\text{-}\mathcal{M}(H)$ and given a class of graphs $C \subseteq \mathcal{G}$ we denote the set of all τ-minors of all graphs in C as $\tau\text{-}\mathcal{M}(C)$.

Finally, we can classify the complexity of counting graphically restricted homomorphisms along the treewidth of their τ-minors:

▶ **Theorem 20** (Theorem 1 and Theorem 2, restated)**.** *Let τ be a graphical restriction and let $C \subseteq \mathcal{G}$ be a recursively enumerable class of graphs. Then $\#\mathsf{Hom}_\tau(C)$ is FPT if $\tau\text{-}\mathcal{M}(C)$ has bounded treewidth and $\#\mathrm{W}[1]$-hard otherwise. Furthermore, given $H, G \in \mathcal{G}$, there exists a deterministic algorithm that computes $\#\mathsf{Hom}_\tau(H,G)$ in time*

$$g(|V(H)|) \cdot |V(G)|^{\mathsf{tw}(\tau\text{-}\mathcal{M}(H))+1}\,,$$

where g is a computable function.

Proof. By Lemma 18 we have that

$$\#\mathsf{Hom}_\tau(H,G) = \sum_{\rho \geq \emptyset} \mu(\emptyset,\rho) \cdot \#\mathsf{Hom}(H/\rho, G)\,.$$

Now, as G has no selfloops, a term $\#\mathsf{Hom}(H/\rho, G)$ is zero whenever H/ρ has a selfloop. Consequently, for every non-zero term $\#\mathsf{Hom}(H/\rho, G)$, it holds that $H/\rho \in \tau\text{-}\mathcal{M}(H)$. Therefore, by Lemma 3.5 in [7], we obtain an algorithm computing $\#\mathsf{Hom}_\tau(H,G)$ in time $g(|V(H)|) \cdot |V(G)|^{\mathsf{tw}(\tau\text{-}\mathcal{M}(H))+1}$, for a computable function g. This immediately implies that the problem $\#\mathsf{Hom}_\tau(C)$ is fixed-parameter tractable if $\tau\text{-}\mathcal{M}(C)$ has bounded treewidth. It remains to show that $\#\mathsf{Hom}_\tau(C)$ is #W[1]-hard otherwise. By condensing all terms $\#\mathsf{Hom}(H/\rho, G)$ and $\#\mathsf{Hom}(H/\sigma, G)$ where H/ρ and H/σ are isomorphic, it follows that there exist coefficients $c_H[H']$ for every $H' \in \tau\text{-}\mathcal{M}(H)$ such that

$$\#\mathsf{Hom}_\tau(H,G) = \sum_{H' \in \tau\text{-}\mathcal{M}(H)} c_H[H'] \cdot \#\mathsf{Hom}(H', G)\,.$$

We will now show that none of the $c_H[H']$ is zero: It holds that

$$c_H[H'] = \sum_{\substack{\rho \geq \emptyset \\ H' \cong H/\rho}} \mu(\emptyset,\rho)\,. \tag{4}$$

Consider ρ and ρ' such that $H/\rho \cong H/\rho' \cong H'$. It follows that $\mathsf{rk}(\rho) = |V(H)| - c(H/\rho) = |V(H)| - c(H') = |V(H)| - c(H/\rho') = \mathsf{rk}(\rho')$. Now, as the lattice of $M(\tau(H))$ is geometric, we can apply the corollary of Rota's NBC Theorem (Theorem 16) and obtain that $\mathsf{sgn}(\mu(\emptyset,\rho)) = (-1)^{\mathsf{rk}(\rho)} = (-1)^{\mathsf{rk}(\rho')} = \mathsf{sgn}(\mu(\emptyset,\rho'))$. Consequently every term in Equation (4) has the same sign and therefore $c_H[H'] \neq 0$. Now we define a function $a_H : \mathcal{G} \to \mathbb{Q}$ as follows

$$a_H(F) := \begin{cases} c_H[F] & \text{if } F \in \tau\text{-}\mathcal{M}(H) \\ 0 & \text{otherwise} \end{cases}$$

and we set $\mathcal{A}_C = \{a_H \mid H \in C\}$. Then the problems $\#\mathsf{Hom}(\mathcal{A}_C)$ and $\#\mathsf{Hom}_\tau(C)$ are equivalent w.r.t. parameterized turing reductions. As $c_H[H'] \neq 0$ for every $H' \in \tau\text{-}\mathcal{M}(H)$ it follows that $\mathcal{A}_C$ has unbounded treewidth if and only if $\tau\text{-}\mathcal{M}(C)$ has unbounded treewidth. We conclude by Theorem 13 that $\#\mathsf{Hom}_\tau(C)$ is #W[1]-hard in this case. ◀

4 Locally injective homomorphisms

In this section we are going to apply the general dichotomy theorem to the concrete case of counting locally injective homomorphisms. A homomorphism φ from H to G is *locally injective* if for every $v \in V(H)$ it holds that $\varphi|_{N(v)}$ is injective. We denote $\mathsf{Li\text{-}Hom}(H,G)$ as the set of all locally injective homomorphisms from H to G and we define the corresponding counting problem $\#\mathsf{Li\text{-}Hom}(C)$ for a class of graphs $C \subseteq \mathcal{G}$ as follows: Given graphs $H \in C$ and $G \in \mathcal{G}$, compute $\#\mathsf{Li\text{-}Hom}(H,G)$. The parameter is $|V(H)|$. Locally injective homomorphisms have already been studied by Nešetřil in 1971 [21] and were applied in the context of distance constrained labelings of graphs (see [13] for an overview). As well as subgraph embeddings, locally injective homomorphisms are graphically restricted homomorphisms. Due to space constraints, the proofs of this section are deferred to the full version.

▶ **Lemma 21.** *Let $H \in \mathcal{G}$ be a graph and let $\tau_{\mathsf{Li}}(H) = (V(H), E_{\mathsf{Li}}(H))$ be a graphical restriction defined as follows: $E_{\mathsf{Li}}(H) = \{\{u,w\} \mid u \neq w \wedge \exists v : \{u,v\},\{w,v\} \in E(H)\}$. Then for all $G \in \mathcal{G}$ it holds that $\mathsf{Hom}_{\tau_{\mathsf{Li}}}(H,G) = \mathsf{Li\text{-}Hom}(H,G)$.*

We continue by stating the dichotomy for counting locally injective homomorphisms.

▶ **Corollary 22** (Corollary 3, restated). *Let $C \subseteq \mathcal{G}$ be a recursively enumerable class of graphs. Then $\#\mathsf{Li\text{-}Hom}(C)$ is FPT if $\tau_{\mathsf{Li}}\text{-}\mathcal{M}(C)$ has bounded treewidth and $\#\mathrm{W}[1]$-hard otherwise. Furthermore, there exists a computable function g and a deterministic algorithm that computes $\#\mathsf{Li\text{-}Hom}(H, G)$ in time*

$$g(|V(H)|) \cdot |V(G)|^{\mathsf{tw}(\tau_{\mathsf{Li}}\text{-}\mathcal{M}(H))+1}.$$

We give an example for a hard instance of the problem: Let W_k be the "windmill" graph of size k, i.e., the graph with vertices $a, v_1, \ldots, v_k, w_1, \ldots, w_k$ and edges $\{a, v_i\}, \{v_i, w_i\}$ and $\{w_i, a\}$ for each $i \in [k]$. Furthermore we let $\mathcal{W}$ be the set of all W_k for $k \in \mathbb{N}$.

▶ **Corollary 23.** *$\#\mathsf{Li\text{-}Hom}(\mathcal{W})$ is $\#\mathrm{W}[1]$-hard.*

In contrast to embeddings where every FPT case is also polynomial time solvable, there are "real" FPT cases when it comes to locally injective homomorphisms. Let $\mathcal{T} \subseteq \mathcal{G}$ be the class of all trees. Counting locally injective homomorphisms from those graphs is fixed-parameter tractable:

▶ **Corollary 24.** *$\#\mathsf{Li\text{-}Hom}(\mathcal{T})$ is FPT. In particular, there is a deterministic algorithm that computes $\#\mathsf{Li\text{-}Hom}(T, G)$ for a tree T in time*

$$g(|V(T)|) \cdot |V(G)|^2,$$

where g is a computable function.

On the other hand $\#\mathsf{Li\text{-}Hom}(\mathcal{T})$ is unlikely to have a polynomial time algorithm.

▶ **Lemma 25.** *$\#\mathsf{Li\text{-}Hom}(\mathcal{T})$ is $\#\mathrm{P}$-hard.*

The proof of this theorem as well as a brief introduction to classical counting complexity can be found in the full version. Corollary 4 follows then from Corollary 24 and Lemma 25.

5 Conclusion and further work

We have shown that various parameterized counting problems can be expressed as a linear combination of homomorphisms over the lattice of graphic matroids, implying immediate complexity classifications along with fixed-parameter tractable algorithms for the positive cases. These results can be obtained without using often cumbersome tools like "gadgeting" or interpolation and relies only on the knowledge of the problem of counting homomorphisms and the comprehension of the cancellation behaviour when transforming a problem into this "homomorphism basis". The latter, in turn, was nothing more than a question about the sign of the Möbius function, which was answered by Rota's Theorem.

This framework, however, still has limits: It seems that, e.g., neither induced subgraphs nor edge-injective homomorphisms [8] are graphically restricted. Indeed, both can be expressed as a sum of homomorphisms over (non-geometric) lattices but the problem is that there are isomorphic terms with different signs in both cases. This suggests that a better understanding of the Möbius function over those lattices could yield even more general complexity classifications of parameterized counting problems.

Acknowledgements. The author is very grateful to Holger Dell and Radu Curticapean for fruitful discussions. Furthermore the author thanks Cornelius Brand for saying "Tutte Polynomial" every once in a while.

References

1 Andreas Blass and Bruce E. Sagan. Möbius functions of lattices. *Advances in mathematics*, 127(1):94–123, 1997.

2 Graham R. Brightwell and Peter Winkler. Note on counting eulerian circuits. *CoRR*, cs.CC/0405067, 2004. URL: http://arxiv.org/abs/cs.CC/0405067.

3 Andrei A. Bulatov. A dichotomy theorem for nonuniform CSPs. *CoRR*, abs/1703.03021, 2017. URL: http://arxiv.org/abs/1703.03021.

4 Jin-yi Cai, Pinyan Lu, and Mingji Xia. Holographic algorithms by fibonacci gates and holographic reductions for hardness. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 644–653, 2008. doi:10.1109/FOCS.2008.34.

5 Jin-yi Cai, Pinyan Lu, and Mingji Xia. Computational complexity of holant problems. *SIAM J. Comput.*, 40(4):1101–1132, 2011. doi:10.1137/100814585.

6 Radu Curticapean. Counting matchings of size k is #W[1]-hard. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming, ICALP, Part I*, pages 352–363, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-39206-1_30.

7 Radu Curticapean, Holger Dell, and Dániel Marx. Homomorphisms are a good basis for counting small subgraphs. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 210–223, 2017. doi:10.1145/3055399.3055502.

8 Radu Curticapean, Holger Dell, and Marc Roth. Counting edge-injective homomorphisms and matchings on restricted graph classes. In *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany*, pages 25:1–25:15, 2017. doi:10.4230/LIPIcs.STACS.2017.25.

9 Radu Curticapean and Dániel Marx. Complexity of counting subgraphs: Only the boundedness of the vertex-cover number counts. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 130–139, 2014. doi:10.1109/FOCS.2014.22.

10 Víctor Dalmau and Peter Jonsson. The complexity of counting homomorphisms seen from the other side. *Theor. Comput. Sci.*, 329(1-3):315–323, 2004. doi:10.1016/j.tcs.2004.08.008.

11 Martin E. Dyer, Leslie Ann Goldberg, and Mark Jerrum. An approximation trichotomy for Boolean #CSP. *J. Comput. Syst. Sci.*, 76(3-4):267–277, 2010. doi:10.1016/j.jcss.2009.08.003.

12 Tomás Feder and Moshe Y. Vardi. The computational structure of monotone monadic SNP and constraint satisfaction: A study through datalog and group theory. *SIAM J. Comput.*, 28(1):57–104, 1998. doi:10.1137/S0097539794266766.

13 Jirí Fiala and Jan Kratochvíl. Locally constrained graph homomorphisms – structure, complexity, and applications. *Computer Science Review*, 2(2):97–111, 2008. doi:10.1016/j.cosrev.2008.06.001.

14 Jörg Flum and Martin Grohe. The parameterized complexity of counting problems. *SIAM J. Comput.*, 33(4):892–922, 2004. doi:10.1137/S0097539703427203.

15 Leslie Ann Goldberg and Mark Jerrum. Counting unlabelled subtrees of a tree is #P-complete. 1999.

16 Mark Jerrum and Alistair Sinclair. Approximating the permanent. *SIAM J. Comput.*, 18(6):1149–1178, 1989. doi:10.1137/0218077.

17 Pieter W. Kasteleyn. Graph theory and crystal physics. In *Graph Theory and Theoretical Physics*, pages 43–110. Academic Press, 1967.

18 Richard E. Ladner. On the structure of polynomial time reducibility. *J. ACM*, 22(1):155–171, 1975. doi:10.1145/321864.321877.

19 Bingkai Lin. The parameterized complexity of k-biclique. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 605–615, 2015. doi:10.1137/1.9781611973730.41.

20 László Lovász. Operations with structures. *Acta Mathematica Hungarica*, 18(3-4):321–328, 1967.

21 Jaroslav Nesetril. Homomorphisms of derivative graphs. *Discrete Mathematics*, 1(3):257–268, 1971. doi:10.1016/0012-365X(71)90014-8.

22 James G. Oxley. *Matroid theory*. Oxford University Press, 1992.

23 Arash Rafiey, Jeff Kinne, and Tomás Feder. Dichotomy for digraph homomorphism problems. *CoRR*, abs/1701.02409, 2017. URL: http://arxiv.org/abs/1701.02409.

24 Gian-Carlo Rota. On the foundations of combinatorial theory I. Theory of möbius functions. *Probability theory and related fields*, 2(4):340–368, 1964.

25 Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, pages 216–226, 1978. doi:10.1145/800133.804350.

26 Richard P Stanley. Enumerative combinatorics: Volume 1. 2011.

27 Harold N. V. Temperley and Michael E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1478–6435, 1961.

28 Salil P. Vadhan. The complexity of counting in sparse, regular, and planar graphs. *SIAM J. Comput.*, 31(2):398–427, 2001. doi:10.1137/S0097539797321602.

29 Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979. doi:10.1016/0304-3975(79)90044-6.

30 Leslie G. Valiant. Accidental algorithms. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 509–517, 2006. doi:10.1109/FOCS.2006.7.

31 Leslie G. Valiant. Holographic algorithms. *SIAM J. Comput.*, 37(5):1565–1594, 2008. doi:10.1137/070682575.

32 Dominic J.A. Welsh. *Matroid theory*. Courier Corporation, 2010.

33 Mingji Xia, Peng Zhang, and Wenbo Zhao. Computational complexity of counting problems on 3-regular planar graphs. *Theor. Comput. Sci.*, 384(1):111–125, 2007. doi:10.1016/j.tcs.2007.05.023.

34 Dmitriy Zhuk. The proof of CSP dichotomy conjecture. *CoRR*, abs/1704.01914, 2017. URL: http://arxiv.org/abs/1704.01914.

Clustering in Hypergraphs to Minimize Average Edge Service Time*

Ori Rottenstreich[1], Haim Kaplan[2], and Avinatan Hassidim[3]

1 **Princeton University, Princeton, NJ, USA**
orir@cs.princeton.edu
2 **Tel Aviv University, Tel Aviv, Israel**
haimk@post.tau.ac.il
3 **Bar-Ilan University, Ramat Gan, Israel**
avinatan@cs.biu.ac.il

Abstract

We study the problem of clustering the vertices of a weighted hypergraph such that on average the vertices of each edge can be covered by a small number of clusters. This problem has many applications such as for designing medical tests, clustering files on disk servers, and placing network services on servers. The edges of the hypergraph model groups of items that are likely to be needed together, and the optimization criteria which we use can be interpreted as the average delay (or cost) to serve the items of a typical edge. We describe and analyze algorithms for this problem for the case in which the clusters have to be disjoint and for the case where clusters can overlap. The analysis is often subtle and reveals interesting structure and invariants that one can utilize.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Clustering, average cover time, hypergraphs, set cover

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.64

1 Introduction

Between 15% and 20% of the population suffers from some form of allergic contact dermatitis [26]. One of the most common ways to treat this is to find the allergen, and avoid it. In order to find the allergen the doctor applies *patch tests* to the patient. Each patch test is applied by attaching a patch containing a cluster of several different allergens to the patient's back. The doctor first decides which allergens to test based on anamnesis. Then she picks a set of clusters that contains all suspected allergens and applies the corresponding patch tests. The study of this paper answers the question how to cluster different allergens together such that common anamnesis require a small number of patch tests. This is in order to reduce the cost and patient's discomfort. Such an abstraction is relevant of course in any scenario in which tests (medical or other) are performed in clusters and one has to design the clusters.[1]

* Work by Haim Kaplan has been supported by Grant 1161/2011 from the German-Israeli Science Foundation, by Grant 1841-14 from the Israel Science Foundation, and by the Israeli Centers for Research Excellence (I-CORE) program (center no. 4/11).

[1] This medical setting may remind the reader of *group testing*. In group testing we want to locate individuals who have a certain property by testing the individuals against groups of properties, rather than against individual ones, and we want to minimize the number of groups. Here we also group properties into tests, but we may have different properties that we try to locate among different subsets and our objective is different.

A similar clustering problem arises in several other application areas. For example, in network design when a network operator has to apply a subset of functions (services such as Deep Packet Inspection, Network Address Translation, etc.) to the packets of each flow. In networks supporting Network Function Virtualization (NFV) these functions are implemented in software on general-purpose servers, where each server can run a limited number of functions [10, 27]. Here we need to assign the functions to servers (a cluster is a set of functions assigned to the same server) such that heavy flows can be served by a small number of servers to minimize delay. Unlike the medical setting, for NFV it is often the case that we need to apply the functions to the packets within a prescribed order.[2]

For a different application consider the task of assigning papers to sessions in a conference with a single track. We would like to construct the program such that attendees interested in particular topics can hear all talks on these topics by attending a small number of sessions.

Another application is in disk servers where one would like to cluster on the same server files that are often read together for minimizing the number of servers that have to be accessed.

1.1 Formal definition of our clustering problem

Our input is a hypergraph $G = (V, E)$ where $|V| = n$. Each edge $e \in E$ has a positive *weight ("frequency")* $w(e)$ satisfying $\sum_e w(e) = 1$. Our goal is to partition V into a collection P of $\alpha = \lceil n/c \rceil$ disjoint clusters, $P = \{B(1), B(2), \ldots, B(\alpha)\}$, $B(i) \subseteq V$, where each cluster is of size no larger than c. For each edge $e \in E$ we define the service time of e to be $t(e) = |\{B \in P \mid B \cap e \neq \emptyset\}|$. Our objective is to compute a clustering that minimizes[3] the average service time $\sum_e w(e)t(e)$.

We also consider the variant of this problem in which the clusters can overlap. For a given number of clusters α and cluster size c such that $\alpha \geq \lceil n/c \rceil$ we want to compute a collection of clusters $P = \{B(1), B(2), \ldots, B(\alpha)\}$ such that each cluster is of size at most c and $\cup_{B \in P} B = V$. In this case we may be able to cover e with a subset of the clusters $\{B \in P \mid B \cap e \neq \emptyset\}$. So, our clustering algorithm is also required to compute a small cover $P(e) \subseteq \{B \in P \mid B \cap e \neq \emptyset\}$, such that $e \subseteq \cup_{B \in P(e)} B$, for each edge e. We define $t(e) = |P(e)|$ and our goal is to compute a clustering and edge covers $P(e)$ that minimize the average service time $\sum_e w(e)t(e)$.

Last we consider the version of this problem in which each edge $e \in E$ is an *ordered* tuple of vertices (rather than a subset of the vertices), say $e = (v_1, \ldots, v_{|e|})$. In this case $P(e)$ has to be a sequence of clusters $B_1, B_2, \ldots, B_{t(e)}$, possibly with repetitions[4] such that there exist $i_1, i_2, \ldots i_{t(e)}$ where $\{v_1, \ldots, v_{i_1}\} \subseteq B_1$, $\{v_{i_1+1}, \ldots, v_{i_2}\} \subseteq B_2$, etc.

We denote the optimal average service time by T_{OPT}. In the above applications, the average time can describe the number of patch tests that have to be applied on average for an anamnesis, the average number of servers a flow has to visit to implement its required functions, the number of sessions one has to attend or the number of disk servers required to be accessed. The maximal allowed cluster size c models a restriction on the number of examined allergens in a patch test, the maximal number of functions that can be implemented in a server, the number of papers in a conference session or the number of files a disk server can save.

---

[2] In this setting the maximum load on a processor is also a relevant metric, which is not a part of this treatment.

[3] We assume without loss of generality that $|e| > 1$ for every $e \in E$ since edges of size 1 just contribute their weight to the cost of any clustering.

[4] That is we may have $B_i = B_j$ for $i \neq j$, $t(e)$ is the size of $P(e)$ counting repetitions.

Table 1 Summary of the results.

# Clusters α	Cluster size c	Hypergraph edges e	Result
Unordered edges			
n/c	2	–	Optimal algo.
–	2	$\lvert e\rvert = 2$	Optimal algo.
n/c	–	$\lvert e\rvert = 2$	$\frac{2c+1}{c+2}$ Approx. algo.
n/c	3	$\lvert e\rvert \leq 3$	5/3 Approx. algo.
–	2	$\lvert e\rvert = 3$	NP-hardness
n/c	3	$\lvert e\rvert = 2$	NP-hardness
n/c	$n/2$	$\lvert e\rvert = 2$	NP-hardness
–	–	–	Bi-criteria approx. algo.
Ordered edges			
n/c	–	–	Approx. algo.

1.2 Our results

We give an algorithm that computes an optimal clustering for the case of disjoint clusters of size $c = 2$. We also give an optimal algorithm for the case of overlapping clusters of size 2 when our hypergraph is in fact a graph (all edges are of size 2). These algorithms compute the optimal clustering by finding maximum matchings in related graphs. In case of disjoint clusters the construction of the graph is relatively straightforward whereas for overlapping clusters the reduction is more sophisticated and requires solving multiple matching problems (see Section 2 and Section 3.1).

In contrast with these positive results we show that when clusters are allowed to overlap, $c = 2$, and the edges of the hypergraph are of size 3 the problem is already NP-hard. Moreover, when the clusters are required to be disjoint the problem becomes NP-hard for $c = 3$ even if $|e| = 2$ for all $e \in E$, so we cannot hope for polynomial algorithms that compute the optimal clustering in a more general setting. This motivates the design of approximation algorithms.

To understand which approximation ratios we are targeting, notice that for any $e \in E$, $t(e) \leq |e|$ and since each cluster is of size at most c, $t(e) \geq \lceil |e|/c \rceil$. This implies that the average service time of any two clusterings is within a factor of c from each other. In particular an arbitrary clustering gives a c-approximation. (Clearly an approximation ratio of $\max\{|e| \mid e \in E\}$ is also achieved by an arbitrary clustering.) Getting an approximation ratio strictly better than c is not trivial.

For disjoint clusters and hypergraphs with edges of size 2 or 3 we describe and analyze a greedy strategy. If all edges are of size 2 we show that this algorithm obtains a clustering of cost at most $(2c+1)/(c+2)$ times the cost of the optimal clustering. When edges are of size 2 or 3 and $c = 3$ we prove that the approximation ratio is at most 5/3. We can generalize the greedy algorithm (in several ways) for hypergraphs with larger edges and for larger values of c but these variants are more complicated to analyze (see Section 2.1 and Section 2.2).

Our analysis (for hypergraphs of edges of size 2 and 3) is subtle and relies on the fact that a clustering has to pay more for edges that cannot be covered by a single cluster. We show that when the optimal clustering is much better than the greedy one, then the subsets of the edges that they cover are almost disjoint. Since they are almost disjoint, it must be that the optimal clustering covers many edges by more than a single cluster (those that are covered by a single cluster in the greedy clustering), and hence it has to pay for them. Interestingly, we observe that for hyperedges with edges of size 3 or more, a stronger phenomena occurs: If

the optimal clustering covers many more edges than the greedy clustering then it must be the case that there are edges which are not covered by neither the greedy nor the optimal clustering. We do not know exactly how to exploit this phenomenon, and leave it as an open question. We hope that this observation would lead to a tighter analysis of a generalization of the greedy for hypergraphs with larger edges.

We give a bi-criteria approximation algorithm to compute overlapping clusters in any hypergraph for any value of c. This algorithm produces a clustering of $O(\alpha \log M \log c)$ clusters whose average service time is larger than the optimal service time with α clusters by a factor of $O(\log M \log c)$. Here M denotes the maximum cardinality of an edge (see Section 3.2).

We use our approximation algorithm for disjoint clusters in the case where all edges are of size 2 to develop an approximation algorithm for the case of disjoint clustering in an ordered hypergraph with edges of arbitrary sizes (see Section 3.3).

Our results are summarized in Table 1.

1.3 Related work

Clustering has always been an important problem, and a lot of research has been done on this topic. Clustering has applications in many different fields, including machine learning, vision, information retrieval and bioinformatics. Different applications have different metrics for the quality of the clustering, and consequently use different algorithms [16]. Some of the more common quality measures include various distance metrics (e.g., the Davies-Bouldin index [12] or the Dunn index [14]), spectral properties [25, 31, 29], and correlation (in correlation clustering [5]).

Clustering is usually a partition of the data (often represented as a graph), but overlapping clusters have also been studied for at least 45 years [11]. Recent applications include solving partial differential equations [6, 18], analysis of social networks [24, 3], wireless networks [1] and solving algorithmic problems on large graphs [7, 4]. One of the challenges in this case is to define the right measure for the quality of the clustering. Taking the standard measure (e.g., the sum of the weights of the edges crossing clusters over the sum of the weight of edges inside clusters) does not give any benefit to overlapping clusters.

One way to measure the quality of a partition is to perform a random walk, and see how long it stays in the same cluster (equivalently how often the random walk crosses clusters). Anderson et al. generalize this metric to overlapping clusters [4]. Their clustering is composed of overlapping clusters that cover all the vertices in the graph, and in addition, each vertex has its primary cluster (one of the clusters it belongs to). To evaluate a clustering and a choice of primary clusters, they start a random walk at a random vertex v_1. Let t_1 denote the number of steps the walk stays in the primary cluster of v_1. Let v_2 denote the first vertex outside that cluster the walk visits. Let t_2 denote the number of steps the walk stays in the primary cluster of v_2, etc. The clustering is good if the expected value of $t_1 + t_2 + t_3 + \ldots$ is large.

There has been many works that deal with non overlapping hypergraph clustering problems. Motivated by applications in computer vision, Agarwal et al. [2] proposed a two phased approach, which first projects the hypergraph to a weighted graph, and then uses graph clustering techniques. Zhou et al. [32] generalize the spectral techniques to work directly on hypergraphs, without the projection stage. Shashua et al. [28] use tensor factorization instead of spectral methods. Lately, Leordeanu and Sminchisescu [22] used an iterative method based on solving a series of LPs, to obtain faster clustering algorithms. Finally, Bulò and Pelillo [8] apply a game theoretic approach, in which every cluster is being

controlled by an agent who tries to maximize the size of her cluster, and the equilibrium status determines the partition. We note that the classical work on hypergraph clustering deal with non overlapping clusters, and that the used metrics differ from ours.

After the selection of clusters, the decision which of them to use to cover each of the edges is an instance of the minimum set cover problem. In the set cover problem the input is a universe U of $n = |U|$ elements and k sets $S_1, \ldots, S_k \subseteq U$. The goal is to find a collection with a minimal number of sets such that its union equals the universe U. Set cover is NP-hard as shown in Karp's seminal paper [21]. A greedy algorithm, selecting as the next set one that covers a maximal number of elements that have not been covered, gives a $\ln n$ approximation. Feige showed that assuming $P \neq NP$, no polynomial-time algorithm can obtain an approximation ratio better than $\ln n$ [17]. Furthermore, if the cardinality of each set is at most c, the greedy algorithm obtains roughly a $1 + \ln c$ approximation [20, 23, 9]. Trevisan [30] adjusted the parameters in Feige's reduction, to show that if the largest set is of size c, no polynomial-time algorithm can obtain an approximation ratio better than $\ln c - O(\ln \ln c)$, assuming $P \neq NP$.

2 Disjoint clusters

In this section we study the case that $\alpha = n/c$, i.e., the clusters are disjoint.

We start with the case of $c = 2$ for which we can find the optimal clustering as follows. We construct a weighted complete (undirected) simple graph $\Lambda = (V, F)$ over the vertices of our input hypergraph $G = (V, E)$ and set the weight $c(u, v)$ of an edge $(u, v) \in F$ to be $\sum_{e \in E | \{u,v\} \subseteq e} w(e)$. We claim that a maximum perfect matching in Λ gives an optimal clustering. Correctness of this algorithm follows from the observation that the average service time of a clustering $P = \{B(1), B(2), \ldots, B(n/c)\}$ is exactly

$$\sum_{e \in E} w(e) \left(|e| - |\{B \in P \mid B \subseteq e\}| \right) = \sum_{e \in E} w(e)|e| - \sum_{B \in P} \sum_{e | B \subseteq e} w(e).$$

This holds since we can trivially serve an edge e with $|e|$ clusters – one per vertex. Each cluster B_i with $|B_i| = 2$ and $B_i \subseteq e$ can be used to serve two of the vertices of e and thereby reduces by one the total number of required clusters. Since clusters are disjoint their contributions add up. We note that in the special case where $|e| = 2$ for each $e \in E$ and E connects every pair of vertices from V then Λ and G are identical, and the optimal solution is given by a maximum matching in G.

2.1 The greedy algorithm for a graph

We start with the case where $|e| = 2$ for every $e \in E$, meaning that our input hypergraph $G = (V, E)$ is in fact a graph. The value of c is arbitrary, and assume for simplicity that $|V|$ is a multiple of c.

A *solution* is a partition of V into disjoint clusters of c vertices. We refer to a partition of the vertices in which every cluster is of size *at most* c as a *partial solution*. We define the *score* $s(B)$ of a cluster B (of size at most c) to be the sum of the weights of the edges contained in B, i.e., $s(B) = \sum_{e \in E | e \subseteq B} w(e)$.

We analyze a simple greedy algorithm that at every step, chooses as the next cluster the set B of maximum score among all possible clusters of size c consisting of uncovered vertices.

▶ **Theorem 1.** *The greedy algorithm results in an average service time of at most* $\frac{2c+1}{c+2} T_{OPT}$.

For $c = 3$ the approximation ratio is $7/5$, and for $c = 4$ it is $3/2$.

We need the following definitions for the proof of Theorem 1. Given a solution or a partial solution X, we define its score $s(X)$ as the sum of the scores of its clusters. In particular, we consider the optimal and the greedy partitions denoted by OPT and GREEDY with scores of $s(OPT)$ and $s(GREEDY)$, respectively. Finally, let $\mathcal{W}$ denote the sum of the weights of the edges in the graph G. We begin with the following lemma that relates $s(GREEDY)$, $s(X)$ for some solution X, and $\mathcal{W}$.

▶ **Lemma 2.** *For every graph $G = (V, E)$ and every solution X and $c \geq 2$, the following relations hold:*
(i) $s(GREEDY) \geq s(X)/c$,
(ii) $(c-1) \cdot s(GREEDY) + \mathcal{W} \geq 2s(X)$.

Proof Outline. The proof is by induction on the number of vertices in the graph. The basis is the case where $|V| \leq c$. In this case $GREEDY$ selects one cluster containing all vertices of G so $s(GREEDY) = \mathcal{W}$. It follows that $s(GREEDY) \geq s(X) \geq s(X)/c$ and $(c-1)s(GREEDY) + \mathcal{W} \geq s(GREEDY) + \mathcal{W} \geq 2 \cdot s(X)$.

Induction step: We assume the lemma holds for any graph with less than $|V|$ vertices. Let $B_1 \subseteq V$ be the first cluster of size c that GREEDY chooses in G. Let $s(B_1)$ be the score of this cluster, that is $s(B_1) = \sum_{v,v' \in B_1, (v,v') \in E} w(v, v')$.

Let $G' = (V', E')$ be the graph generated by deleting from G the vertices in B_1 and all the edges incident to these vertices. That is, $V' = V \setminus B_1$ and $E' = \{(u, v) \in E \mid \{u, v\} \cap B_1 = \emptyset\}$.

We derive from the clustering X of G, a clustering X' of G', by removing from each cluster in X the vertices in B_1, and keeping only clusters with at least two remaining elements following the removal. Formally, the clusters of X' are $\{A \setminus B_1 \mid |A \setminus B_1| \geq 2, A \in X\}$. Let $GREEDY'$ be the solution obtained by running the greedy algorithm in G', which is the same as $GREEDY \setminus B_1$. We have that $s(GREEDY) = s(B_1) + s(GREEDY')$.

The inductive hypothesis applied to G' gives that $s(GREEDY') \geq s(X')/c$, and that $(c-1)s(GREEDY') + \mathcal{W}' \geq 2s(X')$, where $\mathcal{W}'$ is the total weight of the edges in G'.

Let E_X^- be the set of edges in E that are covered by a cluster of X but not covered by a cluster of X' and let $w(E_X^-)$ be the sum of the weights of the edges in E_X^-. Clearly, $E_X^- \subseteq E \setminus E'$ and we have that $s(X) = w(E_X^-) + s(X')$ and

$$\mathcal{W} \geq \mathcal{W}' + w(E_X^-) \ . \tag{1}$$

Let $X_1 \subseteq X$ be the set of clusters in X covering the edges in E_X^-. Since $|B_1| = c$ we must have that $|X_1| \leq c$. Since B_1 was selected by GREEDY it follows that $s(B_1) \geq s(A)$ for every cluster $A \in X$ and in particular for every cluster $A \in X_1$. So it follows that

$$s(B_1) \geq \frac{s(X_1)}{|X_1|} \geq \frac{w(E_X^-)}{|X_1|} \geq \frac{w(E_X^-)}{c} \ . \tag{2}$$

We can now show that $s(GREEDY) \geq s(X)/c$ by combining the induction hypothesis with Equation (2) as follows.

$$s(GREEDY) = s(B_1) + s(GREEDY') \geq w(E_X^-)/c + s(X')/c = s(X)/c \ .$$

To show that $(c-1)s(GREEDY) + \mathcal{W} \geq 2s(X)$, we distinguish between the cases where
(i) $|X_1| \leq c-1$ and
(ii) $|X_1| = c$.
We establish case (ii) by a stronger version of inequality (1) that holds in case and says that $\mathcal{W} \geq \mathcal{W}' + w(E_X^-) + s(B_1)$. ◀

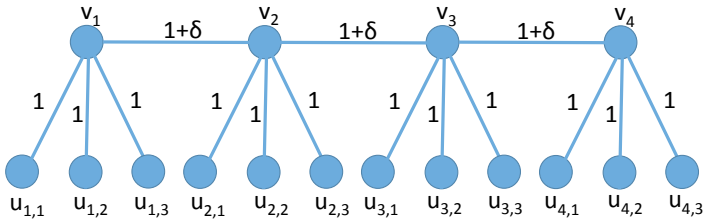


Figure 1 Illustration of the graph $G_{c,\epsilon}$ for $c = 4$ in Lemma 3. It consists of $n = c^2$ vertices $\{v_1, \ldots, v_c\} \cup \{u_{i,j} \mid 1 \le i \le c, 1 \le j \le c-1\}$ and $c^2 - 1$ edges of two types. For every $1 \le i \le c-1$ we have an edge (v_i, v_{i+1}) of weight $1 + \delta$ where $\delta = \epsilon/c(c-1)$. For every $1 \le i \le c$, and every $1 \le j \le c-1$ there is an edge $(v_i, u_{i,j})$ of weight 1.

We now use Lemma 2 to prove Theorem 1.

Proof Outline of Theorem 1. The service time for an edge is 1 if it is contained in one of the clusters of the partition, and 2 otherwise. So we can bound the ratio of the average service time of GREEDY, denoted by T_{GREEDY}, and the average service time of OPT by

$$\frac{T_{GREEDY}}{T_{OPT}} = \frac{2\mathcal{W} - s(GREEDY)}{2\mathcal{W} - s(OPT)} \le \frac{2c+1}{c+2}.$$

The last inequality follows from the bounds in Lemma 2 applied with $X = OPT$ and some algebraic manipulations. ◀

Lemma 2 is tight for any fixed value of c and therefore the approximation ratio of Theorem 1 is also tight. To show this we use the graph $G_{c,\epsilon}$ (for any $\epsilon > 0$ and c) illustrated in Figure 1 for which we prove the following lemma.

▶ **Lemma 3.** *In the graph $G_{c,\epsilon}$, we have $(c-1)s(GREEDY) + \mathcal{W} \le 2s(OPT) + \epsilon$, and simultaneously $s(GREEDY) \le s(OPT)/c + \epsilon$. In particular for this graph $T_{GREEDY} / T_{OPT} \ge (2c+1)/(c+2) - \epsilon$.*

2.2 The case of a hypergraph

Having established tight bounds on the approximation ratio for the case $|e| = 2$, we now move to the more difficult case where $|e| \le 3$ and $c = 3$. We again describe a simple greedy algorithm and bound its approximation ratio.

For $G = (V, E)$, let $E_2 \subseteq E$ and $E_3 \subseteq E$ be the subsets of the edges of size 2 and 3, respectively. A *solution* is a partition of V into triplets. We also refer to a partition of the elements in which every part is of size at most 3 vertices as a *partial solution*. Given a solution or a partial solution X, we denote by X_3 the set of edges $e \in E_3$ that are also triplets in X; by X_2 the set of edges $e \in E_2$ which are contained in a triplet or pair of X; and by $X_{2,3}$ the set of edges $e \in E_3$ such that $|e \cap B| = 2$ for some pair or triplet $B \in X$.

We define the *score*, $s(X)$, of a solution (or a partial solution) X to be $s(X) = 2w(X_3) + w(X_2) + w(X_{2,3})$. In particular, we define the score $s(B)$ of a pair or a triplet $B \in X$ to be twice the weight of B if $B \in E_3$, plus the sum of the weights of the edges $e \in E$ such that $|e \cap B| = 2$. Intuitively, this is the contribution of B to the reduction in the average service time. The average service time T_X of a solution X equals $3w(E_3) + 2w(E_2) - s(X)$.

We consider a greedy algorithm, denoted by $GREEDY$ that at each step picks a triplet B with maximum score and then removes the vertices of B, and all the edges whose restriction

to the remaining graph is of size at most one. The following analog of Lemma 2 is the main technical lemma of this subsection.

▶ **Lemma 4.** *For any solution X the following relations hold*
(i) $s(GREEDY) \geq s(X)/3$.
(ii) $3w(E_3) + 1.5w(E_2) + 3s(GREEDY) \geq 3s(X)$.

Applying this Lemma to $X = OPT$ we prove the following bound on the approximation ratio of $GREEDY$.

▶ **Theorem 5.** *For hypergraphs with $|e| \leq 3$ for all e and $c = 3$, the average service time of the greedy algorithm is at most $\frac{5}{3} \cdot T_{OPT}$.*

▶ Remark. Our upper bound in Section 2.1 for the case where the input graph G is a graph is tight as Lemma 3 shows. This follows since both parts of Lemma 2 are tight for the graph $G_{c,\epsilon}$. On the other hand, we believe that our result in Theorem 5 for the case in which G is a multigraph is not tight as we suspect that there is no graph for which both parts of Lemma 4 are tight. Lemma 4 can be extended for the case of hyperedges of size even larger than 3, but we believe that this approach is unlikely to provide tight bounds. An obvious open problem is to find a way to strengthen Lemma 4 and improve our bounds for the case where G is a hypergraph.

3 Overlapping clusters

In this section we study the scenario of a general number $\alpha \geq n/c$ of clusters that can overlap and are not necessarily disjoint. In the first part of the section we focus on the case where $c = 2$, that is each cluster can include two vertices from V. We give a polynomial-time algorithm that finds an optimal clustering for the case where $|e| = 2$ for all $e \in E$, i.e., the input is a graph. (We recall that without loss of generality we assumed $|e| > 1$ for every $e \in E$.) In the full version of this paper we show that the problem is NP-hard for hypergraphs in which $|e| = 3$ for all $e \in E$ (and $c = 2$). This motivates the second part of this section in which we describe an approximation algorithm that applies to a general instance of the problem.

3.1 Optimal algorithm for a graph

We consider the case where $c = 2$ and $|e| = 2$ for all $e \in E$. Notice that in Section 2, we gave an algorithm that finds an optimal clustering for the case where $c = 2$ and $\alpha = n/c$ but without any assumption on $|e|$. When $c = 2$, $\alpha = n/c$ and $|e| = 2$ for all e, the algorithm we give here and the algorithm of Section 2 are identical.

To simplify the presentation we assume that the input graph G contains all the edges (some may have weight 0) and thereby a clustering P is just a subset of E. Edges of P are served by a single cluster and each other edge is served by two clusters. Let $w(P) = \sum_{e \in P} w(e)$. It follows that the service time of P is $2 \cdot \sum_{e \in E} w(e) - w(P)$. The following characterization of an an optimal clustering now easily follows.

▶ **Theorem 6.** *Let $c = 2$ and α be an arbitrary value. For any weighted graph $G = (V, E)$, a clustering $P = \{e(1), e(2), \ldots, e(\alpha)\}$ minimizes the average service time if and only if P maximizes $w(P)$ while satisfying $\bigcup_{i \in [1,\alpha]} e(i) = V$.*

Assume without loss of generality that $w(e) \neq w(e')$ by some consistent tie breaking scheme. Let $e_1, e_2, \ldots, e_{\binom{n}{2}}$ be the edges of G in decreasing order of weight (by our tie

breaking). Let P be an optimal clustering that includes the longest prefix of $e_1, e_2, \ldots, e_{\binom{n}{2}}$ (among all optimal clusterings). Let x be the length of this prefix and let $e(1), e(2), \ldots, e(\alpha)$ be the edges of P in non-increasing order of weight. By the choice of P, $e(i) = e_i$ for $1 \leq i \leq x$ and $e(x+1) \neq e_{x+1}$ if $x < \alpha$. Here are a few observations about P that follow from Theorem 6.

We say that a cluster $e(j) \in P$ *first covers* a vertex v, if it is the first (according to the order defined above) among the clusters of P that contains v. Each vertex is first covered by exactly one of the clusters and each of the clusters $e(x+1), \ldots, e(\alpha)$ must first cover either one or two vertices. Indeed, if, say, $e(i)$ for some $x+1 \leq i \leq \alpha$, does not first cover any of its vertices, we can replace it in P by e_{x+1} and get a clustering P' such that, $w(P') \geq w(P)$, P' covers all vertices, and P' contains a longer prefix of edges in the sequence $e_1, e_2, \ldots, e_{\binom{n}{2}}$, contradicting the choice of P.

Let y_1 (resp. y_2) be the number of clusters among $e(x+1), \ldots, e(\alpha)$ that first cover a single vertex (resp. two vertices). Clearly we have that $\alpha = x + y_1 + y_2$. Let α_x be the number of distinct vertices in the first x pairs, i.e. $\alpha_x = |\bigcup_{i=1}^{x} e(i)|$. Since there are n vertices and each is first covered exactly once, then $n = \alpha_x + y_1 + 2y_2$. The two equalities imply that

$$y_2 = n - \alpha_x - \alpha + x \;\text{ and }\; y_1 = \alpha - x - y_2 = 2\alpha - 2x - n + \alpha_x. \tag{3}$$

Consider a vertex v that is first covered by a cluster $e(j)$, which first covers only v. We claim that $e(j)$ is the edge of largest weight (first in $e_1, e_2, \ldots, e_{\binom{n}{2}}$) that covers v, as otherwise we can replace it in P by an edge of larger weight, while still covering all vertices of G, contradicting the maximality of P.

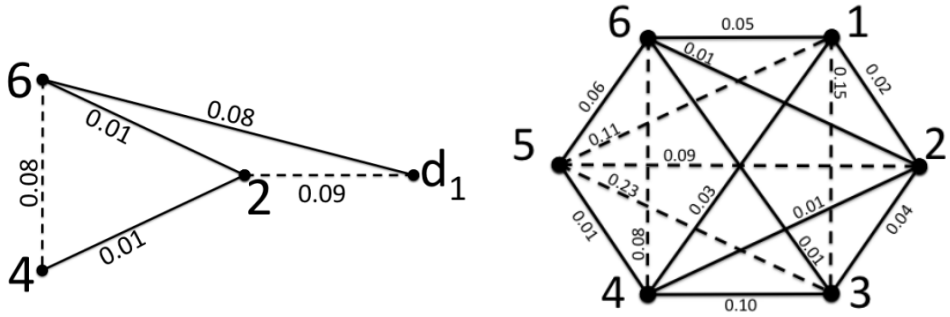
We now turn to describe the algorithm. We first sort the pairs of vertices by their weight and compute the order $e_1, e_2, \ldots, e_{\binom{n}{2}}$. Then we iterate over all possible values of $x \in [0, \alpha]$. For each value x, we construct a clustering $P = \{e(1), e(2), \ldots, e(\alpha)\}$ with $e(i) = e_i$ for $1 \leq i \leq x$ that maximizes $w(P)$ among all such clusterings. We compute α_x and the values of y_1 and y_2 given by Equation (3). To find the $y_1 + y_2 = \alpha - x$ additional clusters, we consider the induced subgraph Λ of G on the $n - \alpha_x$ vertices not covered by the largest x edges of G. We add to this induced subgraph, y_1 additional dummy vertices $d_1, \ldots, d_{y_1}$, and obtain a graph Λ' with $n - \alpha_x + y_1 = 2(\alpha - x) = 2(y_1 + y_2)$ vertices. In Λ' we add an edge (d_i, v) for each $1 \leq i \leq y_1$ if the edge of largest weight covering v is e_j for some $j > x + 1$. We set $w(d_i, v) = w(e_j)$. A maximum perfect matching in Λ' gives us the $n - x$ remaining clusters as argued by the following lemma.

▶ **Lemma 7.** *A perfect matching in Λ' corresponds to the $y_1 + y_2$ edges among $e_{x+2}, \ldots, e_{\binom{n}{2}}$ of largest weight containing the $n - \alpha_x$ vertices of G that are not in $e_1, \ldots, e_x$. An edge of Λ corresponds to itself and an edge (d_i, v), where d_i is a dummy vertex, corresponds to the edge incident to v of largest weight in G.*

The total weight of the clustering defined by a perfect matching in Λ' is given by the sum of the weights of $e_1, \ldots, e_x$ and the weight of the matching. The optimal clustering is selected as the one with the maximum total weight among the clusterings that we get for the various values of x. An example demonstrating the algorithm is illustrated in Figure 2. Finding a maximum weight perfect matching in a general (not necessarily bipartite) graph is a classical combinatorial optimization problem that can be solved in polynomial-time by various implementations of Edmond's algorithm [15] (see also [13] and the references there). The current best strongly polynomial bound is $O(qr + r^2 \log r)$ by Gabow [19] (here q denotes the number of edges and r denotes the number of vertices). Since the number of vertices in

{3,5} 0.23, **{1,3} 0.15**, **{1,5} 0.11**, {3,4} 0.10, {2,5} 0.09, {4,6} 0.08, {5,6} 0.06, {1,6} 0.05, {2,3} 0.04, {1,4} 0.03, {1,2} 0.02, {2,4} 0.01, {4,5} 0.01, {2,6} 0.01, {3,6} 0.01.

(a) The graph edges, sorted by weight.



(b) The corresponding constructed graph Λ'. **(c)** The complete weighted graph.

Figure 2 Illustration of the optimal algorithm for $c = 2$ in a graph (edges of two vertices) (for the described edge weights with $\alpha = 5, n = 6$). (a) shows the graph edges sorted in a non-increasing order of their edge weights, shown next to each edge. The first $x = 3$ pairs appear in bold with $\alpha_x = |\{1, 3, 5\}| = 3$ and $n - \alpha_x = 3$. (b) presents the constructed graph Λ' with vertices $2, 4, 6$ that do not appear in the $x = 3$ pairs and a single additional dummy vertex. (c) shows the complete weighted graph for all vertices in which dashed edges represent clusters in an optimal clustering.

any of the graphs in which we compute a perfect matching is $O(\alpha)$, each maximum matching problem is solved in $O(\alpha^3)$ time. In total we solve at most α matching problems in $O(\alpha^4)$ time. Since $\alpha \geq n/2$, this clearly dominates the time it take to sort the $O(n^2)$ edges. The following theorem summarizes the result of this section.

▶ **Theorem 8.** *The algorithm described above computes an optimal assignment for the case of $c = 2$, $|e| \leq 2$, and runs in $O(\alpha^4)$ time.*

In the special case where $\alpha = n/c = n/2$ the optimal solution corresponds to a maximum perfect matching in G. Indeed, for $x = 0$ we have $\alpha_x = 0$, $y_2 = n - \alpha_x - \alpha + x = n - 0 - \alpha = n/2$, and $y_1 = \alpha - x - y_2 = 0$. So for this value of x there are no dummy vertices in Λ' and an optimal assignment is given by a maximum matching in G. There is no need to try other values of x.

3.2 A Bi-Criteria Approximation Algorithm

We describe an approximation algorithm that applies to *a general instance* of the problem. Let c denote the maximum cluster size and α the number of clusters as before, and let $M = \max_{e \in E} |e|$. We assume that the algorithm has an estimate β of T_{OPT}, the optimal average service time with α clusters, such that $T_{OPT} \leq \beta < 2T_{OPT}$. In case such an estimate is not available, we can run the algorithm with $\beta = M/2^i$ for $i = 0, 1 \ldots, \lfloor \log M \rfloor$. Since $1 \leq T_{OPT} \leq M$, one of these values of β must be in the required range. Our approximation algorithm is bi-criteria, it computes a clustering with an average service time $O(T_{OPT} \log M \log c)$ and $O(\alpha \log M \log c)$ clusters.

Our algorithm adds a single cluster per iteration. At each iteration we compute a *score* for each tentative cluster D, denoted by $\phi(D)$, and we pick the cluster of maximum score. To compute $\phi(D)$ for each cluster D, we maintain for each edge e, the subset

$A(e)$ of the uncovered (by clusters picked at previous iterations) vertices of e. For each edge e we compute the fraction of e covered by D. We define this fraction to be *large* if it is at least $\frac{1}{4\beta}$. The score $\phi(D)$, is the weighted sum of the large fractions that D covers. That is, $\phi(D) = \sum_e w(e)|A(e) \cap D|/|A(e)|$, where the sum is over all e such that $|A(e) \cap D|/|A(e)| \geq \frac{1}{4\beta}$.

We prove that after a *phase* consisting of at most $8\alpha(\log M + 1)$ iterations we completely cover edges of total weight at least a $1/4$ (using $8\alpha(\log M + 1)$ clusters), where each edge which is completely covered is covered by $\leq 8T_{OPT}(\log M + 1)$ clusters. The optimal service time of the remaining (not completely covered) edges (of total weight at most $3/4$) is at most $4/3T_{OPT}$. In the next phase we apply the same procedure to these remaining edges with their new value of T_{OPT} (and estimate β). By the same argument, in the next phase we cover edges whose weights sum to $1/4$ of the total weight of the remaining edges (that were not covered in the first phase) using $\leq 8 \cdot (4/3T_{OPT})(\log M + 1)$ clusters. It follows that each phase adds $O(\alpha \log M)$ clusters and increases the average service time by $O(T_{OPT} \log M)$.

After $O(\log c)$ phases the leftover uncovered edges are of total weight $\leq 1/c$. We cover these remaining $1/c$ fraction of the edges arbitrarily using at most $\lceil n/c \rceil \leq \alpha$ additional clusters including all functions. The following theorem summarizes our result.

▶ **Theorem 9.** *The algorithm described above computes a solution with an average service time $O(T_{OPT} \log M \log c)$ that uses $O(\alpha \log M \log c)$ clusters.*

The running time of our algorithm is exponential in c since we have to compute the scores of all (unused) clusters of size c in each iteration.

3.3 Directed Hypergraphs

In some applications demands have to be served according to a specific order. In this case our input is a *directed* hypergraph meaning that each edge is an *ordered* tuple of vertices. We show how to use our approximation algorithm for graphs (all edges are of size 2 and unordered) specified in Theorem 1 to obtain a clustering with small average service time for directed hypergraphs. We only consider the case where $\alpha = n/c$ (disjoint clusters).

Recall that in the directed case which we consider here $P(e)$ is a sequence of clusters (possibly with repetitions) that cover the vertices of e in order consistent with the order of e. The service time $t(e)$ is the length of the sequence $P(e)$.

For an ordered hyperedge $e = (v_1, v_2, \ldots, v_k)$ let $U(e)$ be the set of the $k-1$ edges $\{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}\}$. We have the following lemma.

▶ **Lemma 10.** *Let $\alpha = n/c$ and let B be a clustering of the vertices of a directed hypergraph H into α disjoint clusters. Let $t(e)$ be the service time for serving an ordered edge e by B. Consider the set $U(e)$ of $|e|-1$ (unordered) edges, each equals to a consecutive pair of vertices in e as defined above. Then the total service time of serving $U(e)$ by B is $t' = t(e) + |e| - 2$.*

Consider for example the edge $e = (8, 2, 1, 7, 3)$ and the clusters $B(1) = \{1, 2, 3\}, B(2) = \{4, 5, 6\}, B(3) = \{7, 8, 9\}$. With this clustering, the edge e must be covered first by cluster 3 (to cover vertex 8), then cluster 1 (covering vertices 2 and 1), then cluster 3 again (to cover vertex 7), and finally cluster 1 again (to cover vertex 3), so $t(e) = 4$. The set $U(e)$ consists of the four edges $\{8, 2\}$, $\{2, 1\}$, $\{1, 7\}$, $\{7, 3\}$. Each of these edges but $\{2, 1\}$ requires two clusters to be covered for a total of 7 which is indeed $t(e) + |e| - 2$.

Lemma 10 suggests the following reduction from the ordered problem to an unordered problem in which $|e| = 2$ for all e.

Given a directed hypegraph $H = (V, E)$ we construct a graph $G = (V, E')$ on the same vertex set V and with $E' = \cup_{e \in E} U(e)$. We set the weight of an edge $e' \in U(e)$ to be $w(e') = w(e)/(W-1)$ where $W = \sum_{e \in E} w(e)|e|$. Note that $W - 1 = \sum_{e \in E} w(e)(|e| - 1)$ is a normalization factor that makes the weights of the edges in U sum to 1.[5]

Consider a clustering B of n vertices into $\alpha = n/c$ clusters. Then by Lemma 10

$$\sum_{e \in E'} w(e)t(e) = \sum_{e \in E} \frac{w(e)}{W-1}(t(e) + |e| - 2).$$

So if T' is the average service time of G by B and T is the average service time of H by B then $T' = (T + W - 2)/(W - 1)$. By rearranging we get that $T = (W-1)T' - W + 2$.

Since all edges in G are of size 2 then we can apply to G the approximation algorithm of Theorem 1 which guarantees an approximation ratio of $\frac{2c+1}{c+2}$. The resulting clustering has the following guarantee for H.

▶ **Theorem 11.** *The clustering obtain for G by the algorithm of Theorem 1 has an average service time $T \leq \frac{2c+1}{c+2} T_{OPT(H)}$ when applied to serve the hyperedges of H, where $T_{OPT(H)}$ is the smallest possible average service time for H.*

Proof. The average service time T of the obtained clustering satisfies that $T \leq (W-1) \cdot \frac{2c+1}{c+2} \cdot T_{OPT(G)} - W + 2$, where $T_{OPT(G)}$ is the average service time of the optimal solution to G. The optimal solution of H induces a solution of G with a service time T' such that $T' = (T_{OPT(H)} + W - 2)/(W-1)$. It follows that $T_{OPT(G)} \leq T' = \frac{T_{OPT(H)} + W - 2}{W-1}$. By substituting the last equation into the previous we get $T \leq \frac{2c+1}{c+2} T_{OPT(H)}$. ◀

4 Conclusions and Open Problems

We introduce the problem of clustering vertices of a weighted hypergraph to minimize the average service time of its edges. For disjoint clusters we described a natural greedy algorithm and analyzed it in two cases: when edges are of size 2, and when edges are of size at most 3 and clusters are of size 3. The latter analysis is subtle and uses an interesting set of invariants. This greedy algorithm can be naturally generalized for larger edges and cluster sizes. Our analysis, however, gets complicated and the number of cases seems to get out of control. Is there an alternative simpler way to analyze such a generalization? One can also try to deal with larger clusters via an hierarchical approach: First cluster the vertices into smaller clusters, then contract these small clusters, and cluster the contracted hypergraph into small clusters again. Finding a way to analyze this hierarchical approach is an open problem. An interesting problem for an experimental research is to compare the performance of the hierarchical and non-hierarchical approaches on some interesting data sets. To conclude, finding a general algorithm for disjoint clusters which is amenable to analysis, and has good approximation ratio is a very interesting challenge (or alternatively proving inapproximability results). Such an algorithm, if practical, will find numerous applications. For overlapping clusters we gave a bicriteria approximation algorithm. A natural open question is to find an algorithm with a guaranteed approximation ratio with respect to the optimal clustering with α clusters that does not require more than α clusters.

[5] Notice that in the constructed unordered instance U we may have identical edges.

References

1 Ameer Ahmed Abbasi and Mohamed Younis. A survey on clustering algorithms for wireless sensor networks. *Computer communications*, 30(14):2826–2841, 2007.

2 Sameer Agarwal, Jongwoo Lim, Lihi Zelnik-Manor, Pietro Perona, David J. Kriegman, and Serge J. Belongie. Beyond pairwise clustering. In *IEEE CVPR*, 2005.

3 Yong-Yeol Ahn, James P. Bagrow, and Sune Lehmann. Link communities reveal multiscale complexity in networks. *Nature*, 466(7307):761–764, 2010.

4 Reid Andersen, David F. Gleich, and Vahab Mirrokni. Overlapping clusters for distributed computation. In *ACM Web search and data mining*, 2012.

5 Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Machine Learning*, 56(1-3):89–113, 2004.

6 Rafael Bru, Francisco Pedroche, and Daniel B. Szyld. Additive Schwarz iterations for Markov chains. *SIAM Journal on Matrix Analysis and Applications*, 27(2):445–458, 2005.

7 Rafael Bru, Francisco Pedroche, and Daniel B. Szyld. Cálculo del vector PageRank de Google mediante el método aditivo de Schwarz. In *Congreso de Métodos Numéricos en Ingeniería*, 2005.

8 Samuel Rota Bulò and Marcello Pelillo. A game-theoretic approach to hypergraph clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(6):1312–1327, 2013.

9 Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

10 Rami Cohen, Liane Lewin-Eytan, Joseph Naor, and Danny Raz. Near optimal placement of virtual network functions. In *IEEE Infocom*, 2015.

11 A. J. Cole and D. Wishart. An improved algorithm for the Jardine-Sibson method of generating overlapping clusters. *The Computer Journal*, 13(2):156–163, 1970.

12 David L. Davies and Donald W. Bouldin. A cluster separation measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):224–227, 1979.

13 Ran Duan. A simpler scaling algorithm for weighted matching in general graphs. *CoRR*, abs/1411.1919, 2014. URL: `http://arxiv.org/abs/1411.1919`.

14 Joseph C. Dunn. Well-separated clusters and optimal fuzzy partitions. *Journal of cybernetics*, 4(1):95–104, 1974.

15 Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

16 Vladimir Estivill-Castro. Why so many clustering algorithms: a position paper. *ACM SIGKDD explorations newsletter*, 4(1):65–75, 2002.

17 Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM*, 45(4):634–652, 1998.

18 Andreas Frommer and Daniel B. Szyld. Weighted max norms, splittings, and overlapping additive Schwarz iterations. *Numerische Mathematik*, 83(2):259–278, 1999.

19 Harold N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *ACM-SIAM SODA*, 1990.

20 David S. Johnson. Approximation algorithms for combinatorial problems. In *ACM symposium on Theory of computing*, 1973.

21 Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations, IBM Thomas J. Watson Research Center*, 1972.

22 Marius Leordeanu and Cristian Sminchisescu. Efficient hypergraph clustering. In *AISTATS*, 2012.

23 László Lovász. On the ratio of optimal integral and fractional covers. *Discrete mathematics*, 13(4):383–390, 1975.

24 Nina Mishra, Robert Schreiber, Isabelle Stanton, and Robert E. Tarjan. Clustering social networks. In *Algorithms and Models for the Web-Graph*, pages 56–67. Springer, 2007.

25 Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 14:849–856, 2002.

26 Matthias Peiser et al. Allergic contact dermatitis: epidemiology, molecular mechanisms, in vitro methods and regulatory aspects. *Cellular and Molecular Life Sciences*, 69(5):763–781, 2012.

27 Ori Rottenstreich, Isaac Keslassy, Yoram Revah, and Aviran Kadosh. Minimizing delay in network function virtualization with shared pipelines. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):156–169, 2017.

28 Amnon Shashua, Ron Zass, and Tamir Hazan. Multi-way clustering using super-symmetric non-negative tensor factorization. In *ECCV*, 2006.

29 Daniel A. Spielmat and Shang-Hua Teng. Spectral partitioning works: Planar graphs and finite element meshes. In *IEEE Foundations of Computer Science*, 1996.

30 Luca Trevisan. Non-approximability results for optimization problems on bounded degree instances. In *ACM symposium on Theory of computing*, 2001.

31 Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

32 Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. In *NIPS*, 2006.

K-Dominance in Multidimensional Data: Theory and Applications

Thomas Schibler[1] and Subhash Suri[2]

1 **University of California, Santa Barbara, CA, USA**
tschibler@gmail.com
2 **University of California, Santa Barbara, CA, USA**
suri@cs.ucsb.edu

Abstract

We study the problem of k-dominance in a set of d-dimensional vectors, prove bounds on the number of maxima (skyline vectors), under both worst-case and average-case models, perform experimental evaluation using synthetic and real-world data, and explore an application of k-dominant skyline for extracting a small set of top-ranked vectors in high dimensions where the full skylines can be unmanageably large.

1998 ACM Subject Classification H. Information Systems, G. Mathematics of Computing

Keywords and phrases Dominance, skyline, database search, average case analysis, random vectors

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.65

1 Introduction

In multi-criteria optimization and decision-making applications, there is often no single best answer, and a popular approach is to use pareto optimality. The set of pareto optimal points, which are the coordinate-wise *undominated* solutions, is called the *skyline*. Unfortunately as the dimension of the data grows,[1] the size of the skyline tends to explode and most, if not all, of the input vectors can appear on the skyline [4, 7, 8]. A database query for a car or a smart phone, for instance, can easily produce an overwhelming number of incomparable choices. In the National Basketball Association's (NBA) database of 21,961 players in 17 dimensions (scoring attributes), more than 1400 players appear on the skyline (see Figure 2 for real-world datasets with large skylines). The problem is even more pronounced in crowdsourced data such as movies or consumer product ratings – each input vector is the rating profile of a product by the users – where virtually every product can be highly ranked by some user, potentially elevating it to the skyline. While the classical result of Bentley et al [4] shows that the expected size of the skyline of n random vectors, whose components are chosen independently, is $O((\log n)^{d-1})$ in d-dimensions, the exponential dependence on d renders the skyline useless even in theory except in very low dimensions.

The *k-dominant skyline* $\mathcal{KDS}$ was introduced recently as a way to tame this curse of dimensionality, where by relaxing d-dominance to k-dominance, for $k < d$, many more points can be eliminated from the skyline, resulting in a smaller, more manageable, set of maxima. Formally, given a finite set of points $\mathcal{V}$ in R^d, a point u is said to *k-dominate* another point

[1] Although in theory all input points can appear on the skyline even in two dimensions, this pathological behavior is rarely observed in low-dimensional real-world data.

v if $u_i \geq v_i$ holds for k of the dimensions, $i = 1, 2, \ldots, d$, with strict inequality in at least one dimension. We use the notation $u \succ_k v$ to indicate k-domination of v by u, and write $u \not\succ_k v$ when u does not k-dominate v. The *k-dominant skyline* of $\mathcal{V}$, denoted $\mathcal{KDS}(\mathcal{V}, k)$, is the set of points that are not k-dominated:

$$\mathcal{KDS}(\mathcal{V}, k) \;=\; \{v \in \mathcal{V} \mid u \;\not\succ_k\; v,\; \forall\, u \in \mathcal{V} \setminus \{v\}\}.$$

In this paper we make both theoretical and applied contributions to the study of k-dominant skylines.

1.1 Main Contributions

We derive the first non-trivial (poly-logarithmic) upper bound on the *average* size of $\mathcal{KDS}$ for random vectors, and in the process generalize the result of Bentley et al. [4]. Our experiments on synthetic and real-world data show that $\mathcal{KDS}$ size conforms to these bounds. We then use movie ratings and NBA basketball datasets to show that $\mathcal{KDS}$ is an effective tool for high-dimensional ranking queries. The full-dimensional skylines of these data sets are unmanageably large, but $\mathcal{KDS}$ consistently finds top vectors using purely pareto property, without the need for ad hoc preference (utility) functions. (We discuss these issues further in Section 6.) Specifically, our paper makes the following contributions:

1. *Average Case Bound:* Let $\mathcal{V}$ be a set of n vectors in d dimensions where the components of each vector are distributed independently of each other, and for each component the magnitudes form a random permutation of $\{1, 2, \ldots, n\}$. We show that the expected number of vectors appearing on the k-dominant skyline is

$$\begin{cases} O\left((\log n)^{2k-d-1}\right) & \text{for } k > \frac{d+1}{2} \\ O(1) & \text{otherwise} \end{cases}$$

   Our result smoothly interpolates the cardinality of $\mathcal{KDS}$ for all values of k, and subsumes the classical result of Bentley et al. [4] as a special case for $k = d$.
2. *Efficient Algorithm:* The k-dominance relation does not obey transitivity, which is needed for the efficient (sub-quadratic) computation of skylines. We show that $\mathcal{KDS}$ can be computed from the (traditional) skylines of all k-dimensional projections of the input vectors. Our algorithm runs in worst-case time $O\left(d^{d-k} n \log^{k-1} n\right)$, which is subquadratic even for non-constant dimensions as long as $d \leq c \log n / \log\log n$, for some constant c.
3. *Experiments:* Our average case analysis assumes attribute distinctness and statistical independence of input vectors. While there is no reason to believe that real-world datasets meet these conditions, our experiments on a variety of data show that the $\mathcal{KDS}$ size follows the exponential growth predicted by our analysis.
4. *Applications:* We show that $\mathcal{KDS}$ is a useful tool for *robust ranking* in high dimensions. For instance, which players should be called the "10 most dominant NBA players" when more than 1400 are pareto optimal (undominated)? Our experiments show that the vectors (NBA players or movies) that are the *first ones* to populate the k-dominant skyline, and therefore have the most longevity because k-dominance is a monotone property, are indeed the natural candidates for top ranking.

1.2 Related Work

The skyline or maxima problem has a long history in computational geometry, databases, and mathematical optimization [3, 4, 13, 6, 5, 9, 15, 19, 20, 21, 27, 28, 14, 22, 25], and the quest for efficient algorithms that scale to high dimensions and large input continues to this

day. It has also been observed that as the dimension grows, the skylines can quickly lose their data-reduction utility because most of the input vectors may appear on the skyline [7, 10].

A number of approaches have been considered for identifying a smaller size representation of the full skyline in high dimensions. One simple method aggregates all the dimensions into a single *utility function*, and uses the ranking defined by this function. This approach while computationally attractive is problematic because it simply transfers all the burden to the "designer" of the utility function. Indeed, one of the important benefits of skyline-based approaches is to let the users *explore* various tradeoffs across different components such as price, location, brand etc. [7, 8]. Another approach is to compute a small set of input vectors that approximates the skyline over *all* preference functions. Unfortunately, these approaches are computationally intractability even for linear preference functions, as in the case of k-regret set [10, 24, 26, 1], or *approximate*-skyline [18].

The focus of our paper is to both prove an upper bound on the size of the $\mathcal{KDS}$ and to explore its applications for ranking high dimensional data. There exists a substantial and rich literature on estimating the cardinality of (conventional) skylines [4, 6, 14, 15, 21, 28], under a variety of data models, including in-memory, distributed, and data streams. However, very little is known about the cardinality of k-dominant skylines. The k-dominant skyline was introduced in [7] but the primary goal of that paper is to design efficient heuristics for computing the $\mathcal{KDS}$ scalably in practice. (In a related work, Chan et al. [8] compute vectors that appear in many k-dominant skylines, but again the paper is concerned with the design and evaluation of an efficient heuristic.) In [17], Hwang et al. consider certain threshold phenomena in k-dominant skylines under a continuous probability model, and derive limit bounds as $n, d \to \infty$. By contrast, we establish parametric upper bounds on the cardinality of $\mathcal{KDS}$ under both random and worst-case inputs, similar to those for the conventional skylines obtained by Bentley et al. [4] or Buchta [6].

2 K-Dominance and the Worst-Case Bound

Let $\mathcal{V}$ be a set of n vectors in d-dimensional space. We will use the letters u and v to denote generic vectors of $\mathcal{V}$, and v_i as the ith coordinate of $v \in \mathcal{V}$. That is, $v = (v_1, v_2, \ldots, v_d)$ is the coordinate representation of vector v. We will use the terms vectors and points interchangeably since vectors are commonly viewed as points in d-dimensional space. Given two vectors $u, v \in \mathcal{V}$, we say that u *dominates* v, denoted $u \succ v$, if $u_i \geq v_i$ for $i = 1, 2, \ldots, d$, with strict inequality in at least one dimension. We say that a vector u *k-dominates* v if $u_i \geq v_i$ for at least k of the d possible dimensions, with strict inequality in at least one. We write this as $u \succ_k v$, generalizing the original definition of domination, which is equivalent to d-domination $\succ_d$. The *k-dominant skyline* of $\mathcal{V}$ is the set of vectors that are not k-dominated by any other vector. That is,

$$\mathcal{KDS}(\mathcal{V}, k) \;=\; \{v \in \mathcal{V} \mid u \not\succ_k v,\ \forall\, u \in \mathcal{V} \setminus \{v\}\}.$$

The k dimensions involved in k-dominance $u \succ_k v$ need not be the same as those involved in $u' \succ_k v'$, and so the k-dominant skyline does not equal the skyline of $\mathcal{V}$ after projection onto some fixed k-dimensional subspace. It also means that the k-dominance relation is *not transitive*: if $u \succ_k v$ and $v \succ_k w$, then we do not necessarily have $u \succ_k w$.

Given a set of input vectors $\mathcal{V}$ in d dimensions, we define a *directed graph* G_k, called the *k-dominant graph*, as follows: the vertices of G_k correspond to the vectors of $\mathcal{V}$, and its edges correspond to k-dominance relationships between pairs of vectors. That is, the graph G_k has a *directed* edge (u, v) whenever $u \succ_k v$, for $u, v \in \mathcal{V}$. The *in-degree* of a node v in G_k is the number of edges directed into v, namely, in-degree$(v) = |\{(u, v) \in G_k\}|$. The following two facts are easy to establish, whose proofs are omitted from this abstract due to lack of space.

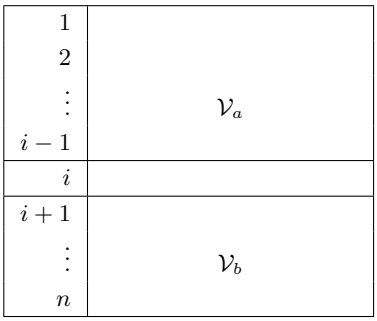


Figure 1 Average case analysis of $\mathcal{KDS}$. The vector v is the ith vector. The first $(i-1)$ vectors form the subset $\mathcal{V}_a$, and the last $(n-i)$ vectors form the subset $\mathcal{V}_b$.

▶ **Lemma 1.** *A vector v belongs to $\mathcal{KDS}(\mathcal{V},k)$ if and only if v has in-degree zero in G_k.*

▶ **Lemma 2.** *$\mathcal{KDS}(\mathcal{V},k') \subseteq \mathcal{KDS}(\mathcal{V},k)$, for $k' < k$.*

With these preliminaries, we can now prove the following bound for the size of the $\mathcal{KDS}$ in the worst-case. Due to lack of space, the proof is omitted from this abstract.

▶ **Theorem 3.** *The worst-case cardinality of $\mathcal{KDS}$ obeys the following bound:*

1. *Given any set of n vectors in d-space and any k with $k \leq (d+1)/2$, $|\mathcal{KDS}(\mathcal{V},k)| \leq 1$.*
2. *For any $n \geq 1$, and k,d such that $k > (d+1)/2$, there* exists *a set $\mathcal{V}$ of n vectors in d-space for which $|\mathcal{KDS}(\mathcal{V},k)| = n$.*

3 Average Case Analysis

We now analyze the size of the k-dominant skylines when the input vectors are drawn randomly from a distribution. Our analysis uses the standard "attribute distinctness and statistical independence" model [4, 6, 14], which only assumes that the vector components are distributed independently of each other, and for each component the magnitudes form a random permutation of $\{1,2,\ldots,n\}$, namely, a total rank ordering. While not all data sets necessarily satisfy these assumptions, the model is sufficiently general, elegant and mathematically tractable for deriving non-trivial bounds.

We interpret the input set of n vectors as an $n \times d$ array, whose rows are the vectors and whose columns are permutations of $\{1,2,\ldots,n\}$. The set of all possible permutations produces exactly $(n!)^d$ distinct vectors. We analyze the complexity of the k-dominant skyline for an input array $\mathcal{V}$ chosen uniformly at random from this set. What is the expected size of the k-dominant skyline for such an input $\mathcal{V}$?

We analyze the average size of $\mathcal{KDS}$ by setting up a recurrence. In order to aid that derivation, let $A(n,d,k)$ denote the average size of the k-dominant skyline for a set of n vectors in d dimensions, where the vectors are chosen under the random model described above. We assume, without loss of generality, that the first column of the input $n \times d$ array is sorted in the ascending order $(1,2,\ldots,n)$ – that is, the first coordinate of the ith vector is i, which if necessary can be realized by simply relabeling the vectors. See Figure 1 for illustration.

Let us focus on a single but arbitrary vector v, and derive the probability that it belongs to the k-dominant skyline. Suppose the vector v is represented as the ith row, which means its first coordinate is $v_1 = i$. We partition the remaining set of input vectors into two groups:

$$\mathcal{V}_a = \{u \in \mathcal{V} \mid u_1 < v_1\} \qquad \text{and} \qquad \mathcal{V}_b = \{u \in \mathcal{V} \mid u_1 > v_1\} \tag{1}$$

That is, $\mathcal{V}_a$ is the set of vectors that v dominates on the first coordinate, and $\mathcal{V}_b$ is the set of vectors that dominate v on the first coordinate. The key observation is that for v to be a $\mathcal{KDS}$ vector both of the following two events must occur:

1. None of the vectors in $\mathcal{V}_a$ k-dominates v among dimensions $\{2, 3, \ldots, d\}$, and
2. none of the vectors in $\mathcal{V}_b$ $(k-1)$-dominates v among dimensions $\{2, 3, \ldots, d\}$.

This follows because v can fail to be on the $\mathcal{KDS}$ only if either some vector in $\mathcal{V}_a$ or some vector in $\mathcal{V}_b$ k-dominates it. If some vector in $\mathcal{V}_a$ were to k-dominate v, it has to do so using all k dimensions from the set $\{2, 3, \ldots, d\}$ since v already dominates each vector of $\mathcal{V}_a$ on dimension 1. Condition (1) computes the probability that no $u \in \mathcal{V}_a$ k-dominates v. On the other hand, since each vector $u \in V_b$ already dominates v on the first coordinate, it needs to only find $k-1$ other dimensions among $\{2, 3, \ldots, d\}$ to achieve k-domination of v. Condition (2) computes the probability that no $u \in \mathcal{V}_b$ k-dominates v. The two events are independent because the remaining $d-1$ dimensions are independent of the first, and so the probability that v belongs to the $\mathcal{KDS}$ is the *product* of these two probabilities. The following two lemmas derive these probabilities.

▶ **Lemma 4.** *The probability of event (1) is* $\frac{A(i,d-1,k)}{i}$.

Proof. Consider the $i \times (d-1)$ array consisting of the first i rows and the last $(d-1)$ columns of $\mathcal{V}$. This is a random set of i vectors in $(d-1)$-dimensional space, which for convenience we call the *reduced space*. By induction, the expected $\mathcal{KDS}$ size for this set is $A(i, d-1, k)$. The probability that v is one of these skyline vectors is $A(i, d-1, k)/i$, by symmetry. Since v already dominates all the vectors of $\mathcal{V}_a$ in the first coordinate, it is on the $\mathcal{KDS}$ of $\mathcal{V}_a \cup \{v\}$ with the same probability. ◀

▶ **Lemma 5.** *The probability of event (2) is* $\frac{A(n-i+1,d-1,k-1)}{n-i+1}$.

Proof. The proof is similar to (1), and omitted from the abstract due to space. ◀

By combining the preceding two lemmas, we get the probability that v lies on the $\mathcal{KDS}$:

$$\Pr[v \in \mathcal{KDS}] = \frac{A(i, d-1, k)}{i} \times \frac{A(n-i+1, d-1, k-1)}{n-i+1}.$$

By summing over all n points, we get

$$A(n, d, k) = \sum_{i=1}^{n} \left(\frac{A(i, d-1, k)}{i} \times \frac{A(n-i+1, d-1, k-1)}{n-i+1} \right) \tag{2}$$

Since $\frac{A(i,d-1,k)}{i}$ is a probability in this recurrence, we can replace it with 1 and derive the following upper bound with a change of index:

$$A(n, d, k) \leq \sum_{i=1}^{n} \left(\frac{A(n-i+1, d-1, k-1)}{n-i+1} \right) \leq \sum_{i=1}^{n} \frac{A(i, d-1, k-1)}{i} \tag{3}$$

The function $A(n, d, k)$ is monotone non-decreasing with n because

$$A(n, d, k) = A(n-1, d, k) + \frac{A(n, d-1, k)}{n} \times \frac{A(1, d-1, k-1)}{1} \tag{4}$$

where the second term is non-negative. Therefore, we have the following upper bound:

$$A(n,d,k) \;\leq\; \sum_{i=1}^{n} \frac{A(i,d-1,k-1)}{i} \;\leq\; A(n,d-1,k-1) \times \sum_{i=1}^{n} \frac{1}{i} \tag{5}$$

Since the harmonic number $H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n$ [11], after j iterations, we get:

$$A(n,d,k) \;\leq\; A(n,d-j,k-j) \times (H_n)^j$$

The stopping condition for the recurrence is reached when $2(k-j)$ becomes less than or equal to $(d-j)+1$, at which point the skyline size drops to at most 1 by Theorem 3. We can show that the maximum number of iterations j is $j = 2k-(d+1)$, which leads to the following theorem.

▶ **Theorem 6.** *Let $\mathcal{V}$ be a set of n random points in d-dimensional space under the attribute distinctness and statistical independence model. Then, the expected cardinality of their k-dominant skyline is*

$$\begin{cases} O\left((\log n)^{2k-d-1}\right) & \text{for } k > \frac{d+1}{2}\,, \\ O(1) & \text{otherwise}\;. \end{cases}$$

Remarks

Theorem 6 smoothly interpolates between the two extreme cardinality bounds for $\mathcal{KDS}$ previously known, namely, $O(1)$ for $k \leq (d+1)/2$, and $O(\log^{d-1} n)$ for $k = d$, and the classical result of Bentley et al. [4] emerges as a special case of this theorem for $k = d$. In database systems, a constructive way to utilize this result could be this: by reducing the value of k by one, we can expect the $\mathcal{KDS}$ to shrink by a significant fraction, namely, a factor of $O(\log^2 n)$. A query engine can therefore tune the parameter k to predictably control the (expected) number of skyline vectors that appear on $\mathcal{KDS}$.

4 Computing KDS in Subquadratic Time

Unlike full-dimensional dominance, the k-dominance relation is *not transitive*: that is, $a \succ_k b$ and $b \succ_k c$ does not guarantee $a \succ_k c$. The failure of transitivity means that the algorithms for traditional skylines [3, 15] cannot be used for computing $\mathcal{KDS}$. In fact, to the best of our knowledge, no subquadratic time algorithm is known for k-dominant skylines even for a constant dimension.

Let $I_k \subset \{1,2,\ldots,d\}$ be an index set of size k, namely, $|I_k| = k$. There are $\binom{d}{k}$ size k distinct index sets, and each such set defines a subset of k dimensions. The projection of the input set of points $\mathcal{V}$ along any particular set I_k is called a *k-projection* of $\mathcal{V}$. A k-projection is a set of k-dimensional points, and we refer to its skyline as the skyline of the k-projection. Then, the following simple observation is the key to our algorithm.

▶ **Lemma 7.** *A point $v \in \mathcal{V}$ belongs to $\mathcal{KDS}(\mathcal{V},k)$ if and only if v belongs to the skylines of all distinct k-projections of $\mathcal{V}$.*

We can, therefore, compute $\mathcal{KDS}(\mathcal{V},k)$ by computing the skylines of all distinct k-projections of $\mathcal{V}$ and taking their common intersection.

▶ **Theorem 8.** *Let $\mathcal{V}$ be a set of n points in d dimensions, where $d = O(\log n/\log\log n)$. The k-dominant skyline $\mathcal{KDS}(\mathcal{V},k)$ is precisely the common intersection of the skylines of all possible k-projections of $\mathcal{V}$, and it can be computed in worst-case time $O(n\log^{d-1} n)$.*

Data	Size n	Dim d	Full Skyline
NBA Career	4051	17	80
NBA Season	21961	17	1466
Movie Lens 1	1682	943	1330
Movie Lens 2	3952	6040	3475
Wine Quality	4898	12	3094
Human Activity	7352	561	7352
Internet Ads	3279	1558	1976
Particle Signal	130065	50	129596

Figure 2 Data sets used in experiments and the size of their d-dimensional skyline.

Proof. The number of distinct k-projections of $\mathcal{V}$ is $\binom{d}{k} = \binom{d}{d-k}$. We can compute the skyline of each of these projections in time $O(n \log^{k-1} n)$ using the divide-and-conquer algorithm of [3]. The size of each of these skylines is at most n, and therefore we can compute their common intersection in $O(nd)$ time. The total running time of the algorithm is therefore $O\left(\binom{d}{d-k}\, n \log^{k-1} n\right)$. Since $\binom{d}{d-k} \leq (\frac{ed}{d-k})^{d-k} = O(d^{d-k})$, we can upper bound the running time as $O\left(d^{d-k} n \log^{k-1} n\right)$, which is sub-quadratic as long as $d \leq c \log n / \log\log n$, for an appropriate constant c. ◀

Finding a worst-case subquadratic algorithm in dimensions higher than $\Theta(\log n)$ remains a challenging open problem, even for conventional skylines.

5 Experimental Evaluation

Our theoretical bounds on the expected size of the $\mathcal{KDS}$ are predicated on certain properties of random data (attribute distinctness and statistical independence), which may or may not be satisfied by real-world datasets. In our experiments, we chose a number of diverse data sets to evaluate how their $\mathcal{KDS}$ size behaves in practice. We also generated synthetic data sets, following our theoretical distribution, to establish a baseline. (Since time complexity was not an important concern, we implemented a simple algorithm, which constructs an $O(dn)$ space data structure in $O(dn^2)$ time from which the $\mathcal{KDS}$ for any value of $k \leq d$ can be extracted in $O(n)$ time explicitly.)

Data Sets

The synthetic data sets are produced by generating random permutations with $n \approx 10^5$ and $d = 20$ or 30. For the real-world data, we use a number of popular high-dimensional data sets, as shown in Figure 2. The NBA basketball data is available at [2], the Movie Lens data at [23, 16], and all the remaining datasets are available from the UCI repository [12]. They vary in size from a few thousand points to more than hundred thousand points in dimensions ranging from 10 to several thousand. Altogether these data sets allow us to evaluate the behavior of $\mathcal{KDS}$ under highly diverse conditions. In addition, for many of these sets, nearly all the input vectors show up on the full-dimensional skyline, providing a useful test for the utility of the $\mathcal{KDS}$. In all the experimental plots, the x-axis is the value of k and y-axis the size of $\mathcal{KDS}$.

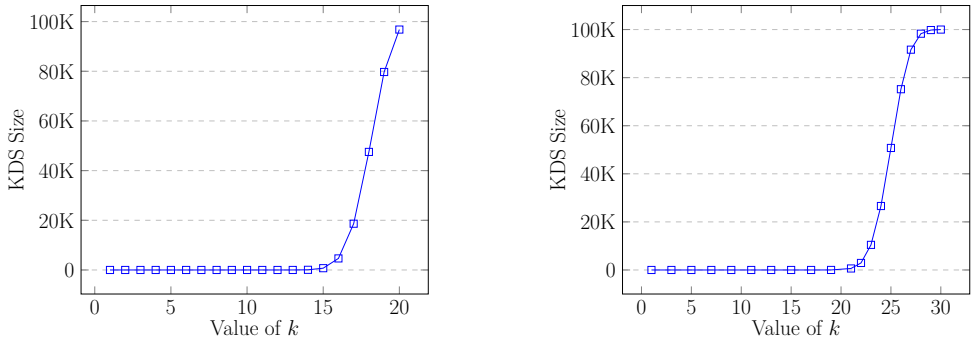


Figure 3 $\mathcal{KDS}$ size scaling for $n = 10^5$ random vectors in 20 (left) and 30 (right) dimensions.

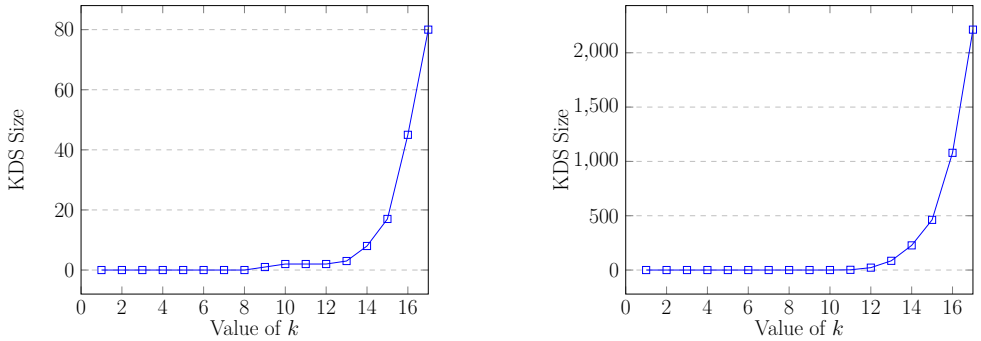


Figure 4 Results for the NBA player data sets, NBA-Career (left) and NBA-Season (right).

5.1 KDS Size for Synthetic Data

We generated a number of random data sets in moderate dimensions $d \leq 30$, which proved sufficient to show the exponential growth rate of the k-dominant skylines. Figure 3 shows the results for $n = 100K$, with $d = 20$ and $d = 30$; the plots for other values of n and d were found to be similar. (When the plots flatten out near the end, it means that the $\mathcal{KDS}$ size has reached the total number of input vectors.)

5.2 KDS Size for Real-World Data

The NBA data sets [2] include scores in 17 different categories for 4051 basketball players. A higher score indicates better performance in each skill (dimension). The NBA-Career set has data aggregated over each player's entire career, while the NBA-Season set has a separate vector for each season in which a player was active (a total of 21961 vectors). Figure 4 shows the $\mathcal{KDS}$ size as a function of increasing k, for both NBA data sets, confirming a clear exponential rate of growth.

Figure 5 shows the results for the Movie Lens data [23, 16], in which each entry is a movie rating. We used first $100,000$ and first $1,000,000$ ratings to generate two different data sets. The former (Movie Lens 1) has $n = 1682$ distinct movies with $d = 943$ distinct reviewers, and the latter (Movie Lens 2) has $n = 3952$ movies with $d = 6040$ reviewers. (Each movie is rated on the scale of 1 (worst) to 5 (best), and we use the default value of 3 (average) for all blank entries.)

Finally, the following figure shows the results for the remaining four data sets: wine quality, human activity, Internet ads, and particle signal.

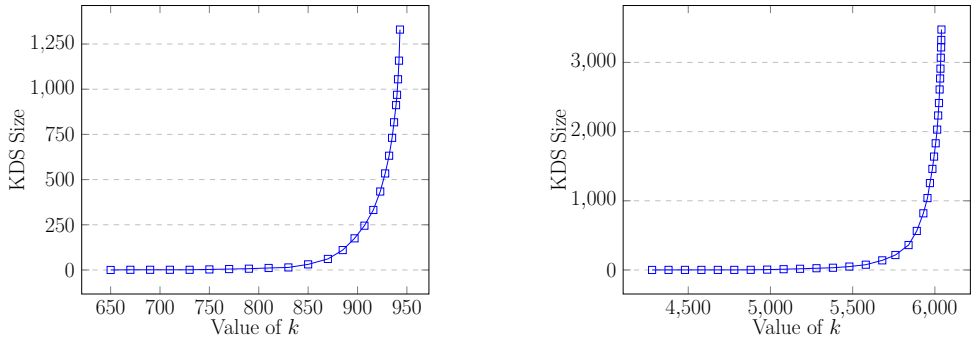


Figure 5 Results for Movie Lens 1 (left) and Movie Lens 2 (right) data sets.

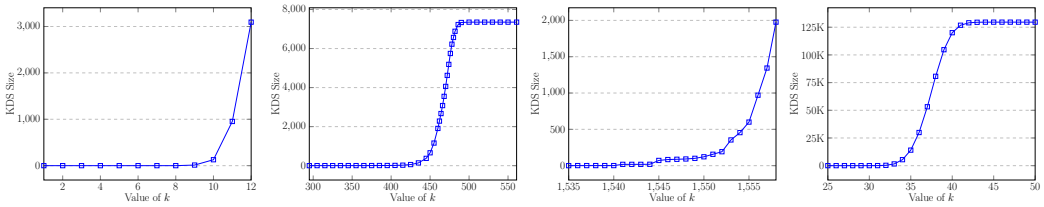


Figure 6 Results for wine quality, human activity, Internet Ads, and particle signal data sets.

In summary, across a diverse collection of data sets, the k-dominant skylines follow the exponential curve shown by our average case analysis, suggesting that the model of random data with attribute distinctness and statistical independence is potentially useful in practice.

6 Application of KDS in Ranking of High-Dimensional Vectors

In multi-criteria decision problems, there is typically no single best answer and often too many incomparable answers are possible. For instance, in the NBA basketball data, almost 1500 players are *undominated*, and thus arguably the best players. One approach, exemplified by the top-k operator in databases, is to define a *utility function* that aggregates the user preferences across multiple dimensions, for instance, as a linear combination. Formulating an appropriate utility function, however, is quite challenging because it requires users to accurately quantify their preferences, and also requires different dimensions to be similar in scale. (In Section 6.3, we also compare simple aggregation-type rankings with the $\mathcal{KDS}$-based ranking.) The alternative approach of *skylines* uses the easier-to-apply principle of pareto optimality – no rational user prefers a solution that is dominated by another on all dimensions – but is stymied by the explosion of skyline size in higher dimensions.

The $\mathcal{KDS}$ suggests a natural approach for *pruning* the skyline using the control knob of parameter k, based on the following insight: many vectors may be undominated in high dimensions, but some are undominated only because they score highly in one or few dimensions, while others are undominated because they score highly in most of the dimensions. Intuitively, the second kind are the more significant ones. The $\mathcal{KDS}$ based approach automatically selects the vectors that remain undominated on most dimensions, and thus appear on the *early* skyline when k is small. Using the asymptotic expected growth rate of $\mathcal{KDS}$ (cf. Theorem 6), a user can *control* the skyline to a desired size with parameter k.

In order to test this hypothesis, we carried out the following experiment. Suppose we consider a *random subset of dimensions* for the data, compute the $\mathcal{KDS}$ of the reduced set,

and repeat this trial multiple times. How similar are the $\mathcal{KDS}$ in these trials? Intuitively, if $\mathcal{KDS}$ vectors lacked any intrinsic significance, then these skylines should not have much in common. On the other hand, if $\mathcal{KDS}$ vectors were fundamentally dominant, then they would persist in many subsets of dimensions, and show a high Jaccard index of similarity. We chose the Movie Lens and NBA data sets for these experiments because their "top vectors" are familiar names, and easy to interpret.

6.1 Top Movies and NBA Players

For the movies, we use the Movie Lens 2 data set, and drop about 10% of the dimensions at random, obtaining a set with 5426 dimensions out of the original 6040. We chose $k = 4800$, for which we are assured to get a small $\mathcal{KDS}$ size, based on Figure 5. We repeated this experiment 5 times, producing 5 different k-dominant skyline sets. The resulting skylines had size between 27 and 29. We then counted how many times each $\mathcal{KDS}$ vector (movie) appeared among these 5 skylines. Figure 7 shows the result: remarkably, the top 27 movies are common to all five $\mathcal{KDS}$ sets. Arguably, all of these movies have claims to be considered "top fan favorites," demonstrating the consistency and utility of $\mathcal{KDS}$ as an automatic data exploration tool.

For the basketball data, we use the NBA-Season data where we randomly dropped 2 of the 17 dimensions, and chose $k = 12$. We performed 5 trials of this experiment, and counted how many times each $\mathcal{KDS}$ vector appears among these 5 skylines. Figure 8 shows the results. Once again, most of the names found by the algorithm are all-time greats.

6.2 Jaccard Similarity

Among the five random trials of the Movie Lens data, the smallest $\mathcal{KDS}$ had 27 vectors, and the largest one had 29 vectors. To quantify the pairwise similarity among these 5 sets, we use the Jaccard similarity measure, which is defined as $\frac{\|A \cap B\|}{\|A \cup B\|}$, for two sets A and B. The Jaccard index of 1 indicates a perfect match. As the following table shows, the similarity index is between 0.93 and 1 in all cases. Among the five random trials of the NBA-Season data, the smallest $\mathcal{KDS}$ for $k = 12$ had 49 vectors, and the largest one had 68 vectors. The table below shows the Jaccard index of similarity among these 5 sets.

6.3 KDS vs. Aggregation-based Ranking

As a point of comparison, we now discuss some pitfalls suffered by alternative ranking methods based on simple utility functions. We use the movie database because the results are easier to interpret. We recall that in this dataset each vector represents a movie, with each dimension being one user's ratings on a 1–5 scale. The movie data, however, is incomplete since not all users rate all movies, and so we use the following two (natural) scoring functions to compare different vectors.

1. the weighted average of each movie's ratings, and
2. the raw sum of each movie's ratings.

We observe that, as expected, the first method tends to highly rank those movies that have high scores but *very few ratings*. In fact, none of the movies ranked in the top 10 would be considered popular – each had an average score of 5 but rated by at most five users! Our second method corrects for this bias, and steers the ranking towards more popular movies by summing the scores of all the users for each movie. However, this method leads to movies that are widely known but not necessarily top rated candidates for many users. For instance,

Movie	Occur
Toy Story	5
The Usual Suspects	5
Braveheart	5
Star Wars: Ep. IV	5
Pulp Fiction	5
Shawshank Redemption	5
Forrest Gump	5
Schindler's List	5
Terminator 2	5
Silence of the Lambs	5
Fargo	5
The Godfather	5
Casablanca	5
One Flew Over Cuckoo's Nest	5
Star Wars: Episode V	5
The Princess Bride	5
Raiders of the Lost Ark	5
Groundhog Day	5
Back to the Future	5
L.A. Confidential	5
Saving Private Ryan	5
Shakespeare in Love	5
The Matrix	5
The Sixth Sense	5
American Beauty	5
Being John Malkovich	5
Gladiator	5

Figure 7 Occurrence frequency of top movies in 5 random trials of $\mathcal{KDS}$.

NBA Player	Occur
Wilt Chamberlain 1961	5
Artis Gilmore 1974	5
Bob Mcadoo 1974	5
George Mcginnis 1974	5
K. Abdul-jabbar 1975	5
Julius Erving 1975	5
Artis Gilmore 1975	5
Moses Malone 1978	5
Michael Jordan 1984	5
Michael Jordan 1986	5
Charles Barkley 1987	5
Michael Jordan 1987	5
Michael Jordan 1988	5
Karl Malone 1988	5
Hakeem Olajuwon 1988	5
Karl Malone 1989	5
David Robinson 1990	5
Hakeem Olajuwon 1992	5
Shaquille O'neal 1993	5
David Robinson 1993	5
Dwight Howard 2007	5
Allen Iverson 2007	5
LeBron James 2007	5
Al Jefferson 2007	5
Amare Stoudemire 2007	5
Dwight Howard 2008	5
Dwyane Wade 2008	5
Kevin Durant 2009	5
Dwight Howard 2009	5
David Lee 2009	5
Charles Barkley 1987	4
Karl Malone 1988	4

Figure 8 Occurrence frequency of top NBA players in 5 random trials.

	1	2	3	4	5
1	1	0.931	0.966	0.966	0.931
2	0.931	1	0.964	0.964	1
3	0.966	0.964	1	1	0.964
4	0.966	0.964	1	1	0.964
5	0.931	1	0.964	0.964	1

Figure 9 Jaccard Similarity for top movies.

	1	2	3	4	5
1	1	0.603	0.716	0.638	0.776
2	0.603	1	0.459	0.493	0.568
3	0.716	0.459	1	0.718	0.658
4	0.638	0.493	0.718	1	0.585
5	0.776	0.568	0.658	0.585	1

Figure 10 Jaccard Similarity for top NBA players.

the original 3 Star Wars movies appeared in the top 4, however the lowest had an average rating of roughly 4/5, which is significantly lower than that of many movies appearing much later in the list. The NBA data set reveals another problem with aggregation-based ranking: different dimensions have vastly different scales, making it difficult to combine them into a single meaningful utility function.

By comparison, as the preceding experiments have shown, the $\mathcal{KDS}$-based ranking gives sensible and robust results without the need for a domain-specific and difficult to formulate utility function. When the full skyline has far too many points, the tunable parameter k of the $\mathcal{KDS}$ automatically acts as a proxy for the importance of skyline vectors: the earlier a point appears on $\mathcal{KDS}$ the more significant it is.

7 Concluding Remarks

In this paper, we made both theoretical and applied contributions to the study of k-dominance in multidimensional data. On the theoretical front, we derived an upper bound on the average case complexity of $\mathcal{KDS}$, which generalizes a classical result of Bentley et al. [4]. We also showed that while k-dominance does not satisfy transitivity, one can still compute the $\mathcal{KDS}$ in roughly the same time as the conventional skyline (worst-case sub-quadratic) as long as the dimension is $d = O(\log n / \log\log n)$. Our experiments show that the size of $\mathcal{KDS}$ in many real-world multi-dimensional data sets follows the same exponential trend of our average case analysis, suggesting that our theoretical bounds can be helpful predictors in practice. Finally, we validate the usefulness of k-dominant skylines as a tool for selecting a small set of top-ranked vectors when the full skyline contains far too many.

References

1. P. Agarwal, N. Kumar, S. Sintos, and S. Suri. Efficient algorithms for k-regret minimizing sets. In *Proc. of 16th Int. Symp. on Experimental Algorithms*, 2017, to appear.
2. Basketball Data. URL: `http://databasebasketball.com/`.
3. J. L. Bentley. Multidimensional Divide and Conquer. *Communications of the ACM*, 23(4):214–229, 1980.
4. J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the Average Number of Maxima in a Set of Vectors and Applications. *Journal of the ACM*, 25(4):536–543, 1978.
5. S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, 2001.
6. C. Buchta. On the average number of maxima in a set of vectors. *Information Processing Letters*, 33(2):63–65, 1989.
7. C. Y. Chan, H. V. Jagadish, K. L. Tan, A. K. H. Tung, and Z. Zhang. Finding K-dominant Skylines in High Dimensional Space. In *Proceedings of the ACM SIGMOD International Conference on Management*, pages 503–514, 2006.
8. C. Y. Chan, H. V. Jagadish, K. L. Tan, A. K. H. Tung, and Z. Zhang. On High Dimensional Skylines. In *Proc. 10th International Conference on Extending Database Technology (EDBT)*, pages 478–495, 2006.
9. S. Chester and I. Assent. Explanations for Skyline Query Results. In *Proc. International Conference on Extending Database Technology (EDBT)*, pages 349–360, 2015.
10. S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides. Computing k-regret Minimizing Sets. *PVLDB*, 7(5):389–400, 2014.
11. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd edition*. MIT Press, McGraw-Hill Book Company, 2000.
12. UCI Data Repository. http://archive.ics.uci.edu/ml/.

13 M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

14 P. Godfrey. Skyline cardinality for relational processing. In *Proc. Foundations of Information and Knowledge Systems (FoIKS)*, pages 78–97, 2004.

15 P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB J.*, 16(1):5–28, 2007.

16 F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems*, 5(4):1–19, 2015.

17 H. K. Hwang, T. H. Tsai, and W. M. Chen. Threshold phenomena in k-dominant skylines of random samples. *SIAM Journal on Computing*, 42(2):405–441, 2013.

18 V. Koltun and C. H. Papadimitriou. Approximately Dominating Representatives. *Theoretical Computer Science*, 371(3):148–154, 2007.

19 D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 275–286, 2002.

20 H. T. Kung, Fabrizio Luccio, and F. P. Preparata. On Finding the Maxima of a Set of Vectors. *Journal of the ACM*, 22(4):469–476, 1975.

21 Yang Lu, Jiakui Zhao, Lijun Chen, Bin Cui, and Dongqing Yang. Effective skyline cardinality estimation on data streams. In *19th International Conference on Database and Expert Systems Applications*, pages 241–254, 2008.

22 J. Matousek. Computing dominances in E^n. *Information Processing Letters*, 38(5):277–278, 1991.

23 Movie Lens Data. URL: `http://grouplens.org/datasets/movielens/`.

24 D. Nanongkai, A. D. Sarma, A. Lall, R. J. Lipton, and J. Xu. Regret-Minimizing Representative Databases. *PVLDB*, 3(1):1114–1124, 2010.

25 D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proc. ACM SIGMOD*, pages 467–478, 2003.

26 P. Peng and R. C-W. Wong. Geometry approach for k-regret query. In *Proc. International Conference on Data Engineering (ICDE)*, pages 772–783, 2014.

27 F. P. Preparata and M. I. Shamos. *Computational Geometry – An Introduction*. Springer, 1985.

28 E. Tiakas, A. N. Papadopoulos, and Y. Manolopoulos. On estimating the maximum domination value and the skyline cardinality of multi-dimensional data sets. *Int. J. Knowledge-Based Organ.*, 3(4):61–83, 2013.

New Abilities and Limitations of Spectral Graph Bisection*†

Martin R. Schuster[1] and Maciej Liśkiewicz[2]

1 **Institute of Theoretical Computer Science, University of Lübeck, Lübeck, Germany**
2 **Institute of Theoretical Computer Science, University of Lübeck, Lübeck, Germany**

Abstract

Spectral based heuristics belong to well-known commonly used methods which determines provably minimal graph bisection or outputs "fail" when the optimality cannot be certified. In this paper we focus on Boppana's algorithm which belongs to one of the most prominent methods of this type. It is well known that the algorithm works well in the random *planted bisection model* – the standard class of graphs for analysis minimum bisection and relevant problems. In 2001 Feige and Kilian posed the question if Boppana's algorithm works well in the semirandom model by Blum and Spencer. In our paper we answer this question affirmatively. We show also that the algorithm achieves similar performance on graph classes which extend the semirandom model.

Since the behavior of Boppana's algorithm on the semirandom graphs remained unknown, Feige and Kilian proposed a new semidefinite programming (SDP) based approach and proved that it works on this model. The relationship between the performance of the SDP based algorithm and Boppana's approach was left as an open problem. In this paper we solve the problem in a complete way by proving that the bisection algorithm of Feige and Kilian provides exactly the same results as Boppana's algorithm. As a consequence we get that Boppana's algorithm achieves the optimal threshold for exact cluster recovery in the *stochastic block model*. On the other hand we prove some limitations of Boppana's approach: we show that if the density difference on the parameters of the planted bisection model is too small then the algorithm fails with high probability in the model.

1998 ACM Subject Classification I.1.2 Algorithms

Keywords and phrases Minimum Graph Bisection, Spectral Methods, Convex Programming

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.66

1 Introduction

The minimum graph bisection problem is one of the classical NP-hard problems [22]: for an undirected graph G the aim is to partition the set of vertices $V = \{1, \ldots, n\}$ (n even) into two equal sized sets, such that the number of cut edges, i.e. edges with endpoints in different bisection sides, is minimized. The bisection width of a graph G, denoted by $\mathrm{bw}(G)$, is then the minimum number of cut edges in a bisection of G. Due to practical significance in VLSI design, image processing, computer vision and many other applications (see [30, 5, 46, 29, 31, 38]) and its theoretical importance, the problem has been the subject of a considerable amount of research from different perspectives: approximability [37, 4, 20, 19, 28], average-case

* A full version of the paper is available at `https://arxiv.org/abs/1701.01337`.
† This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG) grant LI 634/4-1.

complexity [10], and parameterized algorithms [33, 44] including the seminal paper in this field by Cygan et al. [15] showing that the minimum bisection is fixed parameter tractable.

In this paper we consider polynomial-time algorithms that for an input graph either output the *provable* minimum-size bisection or "fail" when the optimality cannot be certified. The methods should work well for all (or almost all, depending on the model) graphs of particular classes, i.e. provide for them a certified optimum bisection, while for irregular, worst case instances the output can be "fail", what is justifiable. We investigate two well-studied graph models: the *planted bisection model* and its extension the *semirandom model* which are widely used to analyze and benchmark graph partitioning algorithms. We refer to [10, 16, 9, 6, 14, 18, 11, 34, 8, 12, 32] to cite some of the relevant works. Moreover, we consider the *regular graph model* introduced of Bui et al. [10] and a new extension of the semirandom model. For a (semi)random model we say that some property is satisfied with high probability (w.h.p.) if the probability that the property holds tends to 1 as the number of vertices $n \to \infty$.

In the planted bisection model, denoted as $\mathcal{G}_n(p,q)$ with parameters $1 > p = p(n) \geq q(n) = q > 0$, the vertex set $V = \{1, \ldots, n\}$ is partitioned randomly into two equal sized sets V_1 and V_2, called the *planted bisection*. Then for every pair of vertices do independently: if both vertices belong to the same part of the bisection (either both belong to V_1 or both belong to V_2) then include an edge between them with probability p; If the two vertices belong to different parts, then connect the vertices by an edge with probability q. In the semirandom model for graph bisection [18], initially a graph G is chosen at random according to model $\mathcal{G}_n(p,q)$. Then a monotone adversary is allowed to modify G by applying an arbitrary sequence of the following monotone transformations: (1) The adversary may remove from the graph any edge crossing a minimum bisection; (2) The adversary may add to the graph any edge not crossing the bisection. Finally, in the regular random model, denoted as $\mathcal{R}_n(r,b)$, with $r = r(n) < n$ and $b = b(n) \leq (n/2)^2$, the probability distribution is uniform on the set of all graphs on V that are r-regular and have bisection width b.

The planted bisection model was first proposed in the sociology literature [27] under the name *stochastic block model* to study community detection problems in random graphs. In this setting, the planted bisection V_1, V_2 (as described above) models latent communities in a network and the goal here is to recover the communities from the observed graph. In the general case, the model allows some errors by recovering, multiple communities, and also that $p(n) < q(n)$. The community detection problem on the stochastic block model has been subject of a considerable amount of research in physics, statistics and computer science (see e.g. [1, 35] for current surveys). In particular, an intensive study has been carried out on providing lower bounds on $|p - q|$ to ensure recoverability of the planted bisection.

The main focus of our work is the bisection algorithm proposed by Boppana [9]. Though introduced almost three decades ago, the algorithm belongs still to one of the most important heuristics in this area. However, several basic questions concerning the algorithm's performance remain open. Using a spectral based approach, Boppana constructs an implementable algorithm which, assuming the density difference

$$p - q \geq c\sqrt{p \ln n}/\sqrt{n} \quad \text{for a certain constant } c > 0 \tag{1}$$

bisects $\mathcal{G}_n(p,q)$ optimally w.h.p. (certifying the optimality of the solutions). Remarkably, for a long time this was the largest subclass of graphs $\mathcal{G}_n(p,q)$ for which a minimum bisection could be found. Since under the assumption (1) the planted bisection is minimum w.h.p., Boppana's algorithm solves the recovery problem for the stochastic block model with two communities. Boppana's algorithm works well also on the regular graph model $\mathcal{R}_n(r,b)$,

assuming that

$$r \geq 6 \quad \text{and} \quad b \leq o(n^{1-1/\lfloor (r/2+1)/2 \rfloor}). \tag{2}$$

In this paper we investigate the problem if, under assumption (1), Boppana's algorithm works well for the semirandom model. This question was posed by Feige and Kilian in [18] and remained open so far. In our work we answer the question affirmatively. We show also that Boppana's algorithm provides the same results as the algorithm proposed currently by Hajek, Wu, and Xu [25]. As a consequence we get that Boppana's algorithm achieves the optimal threshold for exact recovery in the stochastic block model with parameters $p = \alpha \log(n)/n$ and $q = \beta \log(n)/n$. On the other hand we show some limitations of the algorithm. One of the main results in this direction is that the density difference (1) is tight: we prove that if $p - q \leq o(\sqrt{p \cdot \ln n}/\sqrt{n})$ then the algorithm fails on $\mathcal{G}_n(p, q)$ w.h.p.

Our Results. The motivation of our research was to systematically explore graph properties which guarantee that Boppana's algorithm outputs a certified optimum bisection. Due to [9] we know that random graphs from $\mathcal{G}_n(p, q)$ and $\mathcal{R}_n(r, b)$ satisfy such properties w.h.p. under assumptions (1) and (2) on p, q, r, and b as discussed above. But, as we will see later, the algorithm works well also for instances which deviate significantly from such random graphs.

Our first technical contribution is a modification of the algorithm to cope with graphs of more than one optimum bisection, like e.g. hypercubes. The algorithm proposed originally by Boppana does not manage to handle such cases. Our modification is useful to work on wider classes of graphs.

In this paper we introduce a natural generalization of the semirandom model of Feige and Kilian [18]. Instead of $\mathcal{G}_n(p, q)$, we start with an arbitrary initial graph model $\mathcal{G}_n$, and then apply a sequence of the transformations by a monotone adversary as in [18]. We denote such a model by $\mathcal{A}(\mathcal{G}_n)$. One of our main positive results is that if Boppana's algorithm outputs the minimum-size bisection for graphs in $\mathcal{G}_n$ w.h.p., then the algorithm finds a minimum bisection w.h.p. for the adversarial graph model $\mathcal{A}(\mathcal{G}_n)$, too. As a corollary, we get that under assumption (1), Boppana's algorithm works well in the semirandom model, denoted here as $\mathcal{A}(\mathcal{G}_n(p, q))$, and, assuming (2), in $\mathcal{A}(\mathcal{R}_n(r, b))$ – the semirandom regular model. This solves the open problem posed by Feige and Kilian in [18]. To the best of our knowledge, Boppana's algorithm is the only method known so far, that finds (w.h.p.) provably optimum bisections on all of the above random graph classes.

Since the behavior of the algorithm on the (common) semirandom model $\mathcal{A}(\mathcal{G}_n(p, q))$ remained unknown so far, Feige and Kilian proposed in [18] a new semidefinite programming (SDP) based approach which works for semirandom graphs, assuming (1). The relationship between the performance of the SDP based algorithm and Boppana's approach was left in [18] as an open problem. Feige and Kilian conjecture that for every graph G, their objective function $h_p(G)$ to certify the bisection optimality and the lower bound computed in Boppana's algorithm give the same value. In our paper we answer this question affirmatively. To compare the algorithms, we provide a primal SDP formulation for Boppana's approach and prove that it is equivalent to the dual SDP of Feige and Kilian. Next we give a dual program to the primal formulation of Boppana's algorithm and prove that the optima of the primal and dual programs are equal to each other. Note that unlike linear programming, for semidefinite programs there may be a duality gap. Thus, we show that the bisection algorithm of Feige and Kilian provides exactly the same results as Boppana's algorithm. However, an important advantage of the spectral method by Boppana over the SDP based approach by Feige and Kilian is that the spectral method is practically implementable reducing the

bisection problem for graphs with n vertices to computing minima of a convex function of n variables while the algorithm in [18] needs to solve a semidefinite program over n^2 variables.

From the result that the method by Feige and Kilian is equivalent to Boppana's we get, as a consequence, that Boppana's algorithm achieves the sharp threshold for exact cluster recovery in the stochastic block model which has been obtained recently by Abbe et al. [2] and independently by Mossel et al. [36]. In [2, 36] it is proved that in the (binary) stochastic block model, with $p = \alpha \log(n)/n$ and $q = \beta \log(n)/n$ for fixed constants $\alpha \neq \beta$, if $(\sqrt{\alpha} - \sqrt{\beta})^2 > 2$, the planted clusters can be exactly recovered (up to a permutation of cluster indices) with probability converging to one; if $(\sqrt{\alpha} - \sqrt{\beta})^2 < 2$, no algorithm can exactly recover the clusters with probability converging to one. Note, that the choice of p and q is well justified: Mossel et al. show that if $q < p = \log(n)/n$ then the exact recovery is impossible for these parameters. In [25] Hajek et al. proved that the SDP of Feige and Kilian achieves the optimal threshold, i.e. if $(\sqrt{\alpha} - \sqrt{\beta})^2 > 2$ the SDP reconstructs communities w.h.p. From our result we get, that Boppana's algorithm achieves the threshold, too.

To analyze limitations of the spectral approach we provide structural properties of the space of feasible solutions searched by the algorithm. This allows us to prove that if an optimal bisection contains some forbidden subgraphs, then Boppana's algorithm fails. Using these tools, we were able to show that if the density difference $p - q$ is asymptotically smaller than $\sqrt{p \cdot \ln n}/\sqrt{n}$ then Boppana's algorithm fails to determine a certified optimum bisection on $\mathcal{G}_n(p, q)$ w.h.p. Note that our impossibility result is not a direct consequence of the lower bound for the exact cluster recovery discussed above. For example, for $q = \mathcal{O}(1)/n$ and $p = \sqrt{\log n}/n$ from Mossel et al. [36] we know that for these parameters the exact recovery is impossible but obviously this does not imply that determining of a certified optimum bisection is impossible either.

Related Works. Spectral partitioning goes back to Fiedler [21], who first proposed to use eigenvectors to derive partitions. Spielman and Teng e.g. showed, that spectral partitioning works well on planar graphs [40, 41], although there are also graphs on which purely spectral algorithms perform poorly, as shown by Guattery and Miller [24].

Also other algorithms have been proven to work on the planted bisection model. Condon and Karp [14] developed a linear time algorithm for the more general l-partitioning problem. Their algorithm finds the optimal partition with probability $1 - \exp(-n^{\Theta(\varepsilon)})$ in the planted bisection model with parameters satisfying $p - q = \Omega(1/n^{1/2-\varepsilon})$. Carson and Impagliazzo [11] show that a hill-climbing algorithm is able to find the planted bisection w.h.p. for parameters $p - q = \Omega((\ln^3 n)/n^{1/4})$. Dyer and Frieze [16] provide a min-cut via degrees heuristic that, assuming $n(p - q) = \Omega(n)$ finds and certifies the minimum bisection w.h.p. Note, that the density difference (1) assumed by Boppana still outperforms the above ones. Moreover a disadvantage of the methods against Boppana's algorithm, except for the last one, is that they do not certify the optimality of the solutions. In [34] McSherry describes a spectral based heuristic that applied to $\mathcal{G}(p, q)$ finds a minimum bisection w.h.p if p and q satisfy assumption (1) but it does not certify the optimality. Importantly, the algorithms above, similarly as Boppana's method, solve the recovery problem for the stochastic block model with two communities.

In [12] Coja-Oghlan developed a new spectral-based algorithm which, on the planted partition model $\mathcal{G}_n(p, q)$, enables for a wider range of parameters than (1), certifying the optimality of its solutions. The algorithm [12] assumes that $p - q \geq \Omega(\sqrt{p \ln(np)}/\sqrt{n})$. If the parameters p and q describe non-sparse graphs, this condition is essentially the same as Boppana's assumption. For sparse graphs, however, Coja-Oghlan's constraint allows a larger

subclass. For example, the algorithm works in $\mathcal{G}_n(p,q)$ for $q = \mathcal{O}(1)/n$ and $p = \sqrt{\log n}/n$. Due to results presented in our paper we know that Boppana's algorithm fails w.h.p. for such graphs. Interestingly, the condition on the density difference by Coja-Oghlan allows graphs for which the minimum bisection width is strictly smaller than the width of the planted bisection w.h.p. However, a drawback of Coja-Oghlan's algorithm is that to work well in the planted bisection model with *unknown* parameters p and q, the algorithm has to learn the parameters since it is based on the knowledge of values p and q. Also the performance of the algorithm on other families, like e.g. semirandom graphs and the regular random graphs $\mathcal{R}_n(r,b)$, is unknown. Recent research by Coja-Oghlan et al. [13] contributes to a better understanding of the planted bisection model and average case behavior of a minimum bisection.

The paper is organized as follows. The next section contains an overview over Boppana's algorithm. In Section 3 we define the adversarial graph model and show, that Boppana's algorithm works well on this class. In Section 4 we compare the algorithm to the SDP approach of Feige and Kilian. Next, in Section 5 we propose a modification of the algorithm to deal with non-unique optimum bisections. Finally, we develop a new analysis of the algorithm and use it to show some limitations of the method. We conclude the paper with a discussion. The proofs of most of the propositions presented in Sections 2 through 6 can be found in the full version [39].

2 Boppana's Graph Bisection Algorithm

In this section we fix definitions and notations used in our paper and we recall Boppana's algorithm and known facts on its performance. We need the details of the algorithm to describe its extension in the next section. For a given graph $G = (V, E)$, with $V = \{1, \ldots, n\}$, Boppana defines a function f for all real vectors $x, d \in \mathbb{R}^n$ as

$$f(G, d, x) = \sum_{\{i,j\} \in E} \frac{1 - x_i x_j}{2} + \sum_{i \in V} d_i (x_i^2 - 1). \tag{3}$$

Call by $S \subset \mathbb{R}^n$ the subspace of all vectors $x \in \mathbb{R}^n$, with $\sum_i x_i = 0$. Based on f, the function g' is defined as follows

$$g'(G, d) = \min_{\|x\|^2 = n, x \in S} f(G, d, x), \tag{4}$$

where $\|x\|$ denotes the L_2 norm of x. Note that g' is invariant under shifting d, i. e. $g'(G, d + \beta(1, \ldots, 1)^T) = g'(G, d)$ for every $\beta \in \mathbb{R}$. Vector x is named a *bisection vector* if $x \in \{+1, -1\}^n$ and $\sum_i x_i = 0$. Such x determines a bisection of G of the cut width denoted as $\mathrm{cw}(x) = \sum_{\{i,j\} \in E} \frac{1 - x_i x_j}{2}$. For a bisection vector x the function f takes the value (3) regardless of d. Minimization over all such x would give the minimum bisection width. Since g' uses a relaxated constraint we get $g'(G, d) \leq \mathrm{bw}(G)$ where, recall, $\mathrm{bw}(G)$ denotes the bisection width of G. To improve the bound, Boppana tries to find some d which leads to a minimal decrease of the function value of g' compared to the bisection width:

$$h(G) = \max_{d \in \mathbb{R}^n} g'(G, d). \tag{5}$$

It is easy to see that for every graph G we have $h(G) \leq \mathrm{bw}(G)$.

In order to compute g' efficiently, Boppana expresses the function in spectral terms. To describe this we need some definitions. Let I denote the n-dimensional identity matrix and let $P = I - \frac{1}{n} J$ be the projection matrix which projects a vector $x \in \mathbb{R}^n$ to the projection Px of vector x into the subspace S. Here, J denotes an $n \times n$ matrix of ones. For a matrix

$B \in \mathbb{R}^{n\times n}$, the matrix $B_S = PBP$ projects a vector $x \in \mathbb{R}^n$ to S, then applies B and projects the result again into S. Further, for $B \in \mathbb{R}^{n\times n}$ and $d \in \mathbb{R}^n$ we denote the sum of B's elements as $\mathrm{sum}(B) = \sum_{ij} B_{ij}$ and by $\mathrm{diag}(d)$ we denote the $n \times n$ diagonal matrix D with the entries of the vector d on the main diagonal, i. e. $D_{ii} = d_i$.

Now assume $B \in \mathbb{R}^{n\times n}$ is symmetric and let $B_S = PBP$. Denote by $\mathbb{R}^n_{\neq c\mathbf{1}}$ the real space $\mathbb{R}^n$ without the subspace spanned by the identity vector $\mathbf{1}$, i. e. $\mathbb{R}^n_{\neq c\mathbf{1}} = \mathbb{R}^n \setminus \{c\mathbf{1} : c \in \mathbb{R}\}$. We define $\lambda(B_S) = \max_{x\in\mathbb{R}^n_{\neq c\mathbf{1}}} \frac{x^T B_S x}{\|x\|^2}$. It is easy to see that if $\lambda(B_S) \geq 0$ then

$$\lambda(B_S) = \max_{x\in\mathbb{R}^n} \frac{x^T B_S x}{\|x\|^2} \tag{6}$$

i. e. $\lambda(B_S)$ is the largest eigenvalue of the matrix B_S. Vectors x that attain the maximum are exactly the eigenvectors corresponding to the largest eigenvalue $\lambda(B_S)$ of B_S.

Let G be an undirected graph with n vertices and adjacency matrix A. Let further $d \in \mathbb{R}^n$ be some vector and let $B = A + \mathrm{diag}(d)$, then we define

$$g(G, d) = \frac{\mathrm{sum}(B) - n\lambda(B_S)}{4}.$$

In [9] it is shown that function g' can be expressed as $g'(G, d) = g(G, -4d)$. Since in the definition of h in (5) we maximize over all d, we can conclude that

$$h(G) \;=\; \max_{d\in\mathbb{R}^n} g(G, d) \;=\; \max_{d\in\mathbb{R}^n} \frac{\mathrm{sum}(A + \mathrm{diag}(d)) - n\lambda((A + \mathrm{diag}(d))_S)}{4}. \tag{7}$$

Boppana's algorithm that finds and certifies an optimal bisection, works as follows:

Algorithm 1: Boppana's Algorithm

1 Compute $h(G)$: Numerically find a vector d^{opt} which maximizes $g(G, d)$. Let $D = \mathrm{diag}(d^{\mathrm{opt}})$. Use constraint $\sum_i d_i^{\mathrm{opt}} = 2|E|$ to ensure $\lambda((A + D)_S) > 0$;
2 Construct a bisection: Let x be an eigenvector corresponding to the eigenvalue $\lambda((A + D)_S)$. Construct a bisection vector $\hat{x}$ by splitting at the median $\bar{x}$ of x, i.e. let $\hat{x}_i = +1$ if $x_i \geq \bar{x}$ and $\hat{x}_i = -1$ if $x_i < \bar{x}$. If $\sum_i \hat{x}_i > 0$, move (arbitrarily) $\frac{1}{2}\sum_i \hat{x}_i$ vertices i with $x_i = \bar{x}$ to part -1 letting $\hat{x}_i = -1$;
3 Output $\hat{x}$; If $\mathrm{cw}(\hat{x}) = h(G)$ output "optimum bisection" else output "fail".

One can prove that g is concave and hence, the maximum in Step 1 can be found in polynomial time with arbitrary precision [23]. To analyze the algorithm's performance, Boppana proves the following, for a sufficiently large constant $c > 0$:

▶ **Theorem 1** (Boppana [9]). *Let G be a random graph from $\mathcal{G}_n(p, q)$, and let $p - q \geq c(\sqrt{p \ln n}/\sqrt{n})$. Then with probability $1 - \mathcal{O}(1/n)$, the bisection width of G equals $h(G)$.*

From this result one can conclude that the value $h(G)$ computed by the algorithm is, w.h.p., equal to the optimal bisection width of G. However, to guarantee that the algorithm works well one needs additionally to show that it also finds an optimal bisection:

▶ **Theorem 2.** *For random graphs G from $\mathcal{G}_n(p, q)$, with $p - q \geq c(\sqrt{p \ln n}/\sqrt{n})$, Boppana's algorithm certifies the optimality of $h(G)$ revealing w.h.p. bisection vector $\hat{x}$ of $\mathrm{cw}(\hat{x}) = h(G)$.*

To prove this theorem one first has to revise carefully the proof of Theorem 1 in [9] and show that w.h.p. the multiplicity of the largest eigenvalue of the matrix $(A + D)_S$ in Step 1 is 1. This was observed already in [7]. Next we need the following property:

▶ **Lemma 3.** *Let G be a graph with $h(G) = \mathrm{bw}(G)$ and let $d^{\mathrm{opt}} \in \mathbb{R}^n$ s. t. $g(G, d^{\mathrm{opt}}) = \mathrm{bw}(G)$ and $\sum_i d_i^{\mathrm{opt}} \geq 4\,\mathrm{bw}(G) - 2|E|$. Denote further by $B^{\mathrm{opt}} = A + \mathrm{diag}(d^{\mathrm{opt}})$. Then every optimum bisection vector y is an eigenvector of B_S^{opt} corresponding to the largest eigenvalue $\lambda(B_S^{\mathrm{opt}})$.*

(The proof of Lemma 3, as the proofs of most of the remaining propositions presented in this paper, can be found in the full version [39].) This completes the proof that the algorithm works well on random graphs from $\mathcal{G}_n(p, q)$.

3 Bisections in Adversarial Models

We introduce the *adversarial model*, denoted by $\mathcal{A}(\mathcal{G}_n)$, as a generalization of the semirandom model in the following way. Let $\mathcal{G}_n$ be a graph model, i.e. a class of graphs with distributions over graphs of n nodes (n even). In the model $\mathcal{A}(\mathcal{G}_n)$, initially a graph G is chosen at random according to $\mathcal{G}_n$. Let (Y_1, Y_2) be a fixed, but arbitrary optimal bisection of G. Then, similarly as in [18], a monotone adversary is allowed to modify G by applying an arbitrary sequence of the following monotone transformations: The adversary may

1. remove from the graph any edge $\{u, v\}$ crossing the bisection ($u \in Y_1$ and $v \in Y_2$);
2. add to the graph any edge $\{u, v\}$ not crossing the bisection ($u, v \in Y_1$ or $u, v \in Y_2$).

For example, $\mathcal{A}(\mathcal{G}_n(p, q))$ is the semirandom model as defined in [18].

We will prove that Boppana's algorithm works well for graphs from adversarial model $\mathcal{A}(\mathcal{G}_n)$ if the algorithm works well for $\mathcal{G}_n$. First we show that, if the algorithm is able to find an optimal bisection size of a graph, we can add edges within the same part of an optimum bisection and that we can remove cut edges, and the algorithm will still work. This solves the open question of Feige and Kilian [18].

Note that the result follows alternatively from Corollary 11 (presented in Section 4) that the SDPs of [18] are equivalent to Boppana's optimization function and form the property proved in [18] that the objective function of the dual SDP of Feige and Kilian preserves minimal bisection regardless of monotone transformations. The aim of this section is to give a direct proof of this property for Boppana's algorithm.

▶ **Theorem 4.** *Let $G = (V, E)$ be a graph with $h(G) = \mathrm{bw}(G)$. Consider some optimum bisection Y_1, Y_2 of G.*

1. *Let u and v be two vertices within the same part, i.e. $u, v \in Y_1$ or $u, v \in Y_2$, and let $G' = (V, E \cup \{\{u, v\}\})$. Then $h(G') = \mathrm{bw}(G')$.*
2. *Let u and v be two vertices in different parts, i.e. $u \in Y_1$ and $v \in Y_2$, with $\{\{u, v\}\} \in E$ and let $G' = (V, E \setminus \{\{u, v\}\})$. Then $h(G') = \mathrm{bw}(G) - 1 = \mathrm{bw}(G')$.*

Sketch of Proof. In order to prove the first part of the theorem, i.e. when we add an edge $\{u, v\}$, let A and A' denote the adjacency matrices of G and G', respectively. It holds $A' = A + A^{\Delta}$ with $A^{\Delta}_{uv} = A^{\Delta}_{vu} = 1$ and zero everywhere else. The main idea is now, that we can derive a new optimal correction vector d' for G' based on the optimal correction vector d^{opt} for G. We set $d' = d^{\mathrm{opt}} + d^{\Delta}$ with $d^{\Delta}_i = \begin{cases} -1 & \text{if } i = u \text{ or } i = v, \\ 0 & \text{else.} \end{cases}$

The known changes in the adjacency matrix as well as the derived correction vector allow us to compute $g(G', d')$ and to show that $g(G', d') = \mathrm{bw}(G')$. The proof of the second part of the theorem works analogously. The complete proof can be found in the full version [39]. ◀

▶ **Theorem 5.** *If Boppana's algorithm finds a minimum bisection for a graph model $\mathcal{G}_n$ w.h.p., then it finds a minimum bisection w.h.p. for the adversarial model $\mathcal{A}(\mathcal{G}_n)$, too.*

As a direct consequence, we obtain the following corollary regarding the semirandom graph model considered by Feige and Kilian:

▶ **Corollary 6.** *Under assumption* (1) *on* p *and* q*, Boppana's algorithm computes the minimum bisection in* $\mathcal{A}(\mathcal{G}_n(p,q))$*, i.e. in the semirandom model, w.h.p.*

In [9], Boppana also considers random regular graphs $\mathcal{R}_n(r,b)$, where a graph is chosen uniformly over the set of all r-regular graphs with bisection width b. He shows that his algorithm works w.h.p. on this graph under the assumption that $b = o(n^{1-1/\lfloor (r+1)/2 \rfloor})$. We can now define the semirandom regular graph model as adversarial model $\mathcal{A}(\mathcal{R}_n(r,b))$. Applying Theorem 5, we obtain

▶ **Corollary 7.** *Under assumption* (1) *on* p *and* q*, Boppana's algorithm computes the minimum bisection in the semirandom regular model w.h.p.*

4 SDP Characterizations of the Graph Bisection Problem

Feige and Kilian express the minimum-size bisection problem for an instance graph G as a semidefinite programming problem (SDP) with solution $h_p(G)$ and prove that the function $h_d(G)$, which is the solution to the dual SDP, reaches $\mathrm{bw}(G)$ w.h.p. Since $\mathrm{bw}(G) \geq h_p(G) \geq h_d(G)$, they conclude that $h_p(G)$ as well reaches $\mathrm{bw}(G)$ w.h.p. The proposed algorithm computes $h_p(G)$ and reconstructs the minimum bisection of G from the optimum solution of the primal SDP. The authors conjecture in [18, Sec. 4.1.] the following: "Possibly, for every graph G, the function $h_p(G)$ and the lower bound $h(G)$ computed in Boppana's algorithm give the same value, making the lemma that $h_p(G) = \mathrm{bw}(G)$ w.h.p. a restatement of the main theorem of [9]. In this section we answer this question affirmatively.

The semidefinite programming approach for optimization problems was studied by Alizadeh [3], who as first provided an equivalent SDP formulation of Boppana's algorithm. Before we give an SDP introduced by Feige an Kilian, we recall briefly some basic definitions and provide an SDP formulation for Boppana's approach. On the space $\mathbb{R}^{n\times m}$ of $n\times m$ matrices, we denote by $A\bullet B$ an inner product of A and B defined as $A\bullet B = \mathrm{tr}(AB) = \sum_{i=1}^{n}\sum_{j=1}^{m} A_{ij}B_{ij}$, where $\mathrm{tr}(C)$ is the trace of the (square) matrix C. Let A be an $n \times n$ symmetric real matrix, then A is called symmetric positive semidefinite (SPSD) if A is symmetric, i.e. $A^T = A$, and for all real vectors $v \in \mathbb{R}^n$ we have $v^T Av \geq 0$. This property is denoted by $A \succeq 0$. Note that the eigenvalues of a symmetric matrix are real.

For given real vector $c \in \mathbb{R}^n$ and $m+1$ symmetric matrices $F_0, \ldots, F_m \in \mathbb{R}^{n\times n}$ an SDP over variables $x \in \mathbb{R}^n$ is defined as

$$\min_x c^T x \quad \text{subject to} \quad F_0 + \sum_{i=1}^{m} x_i F_i \ \succeq \ 0. \tag{8}$$

The dual program associated with the SDP (for details see e.g. [45]) is the program over the variable matrix $Y = Y^T \in \mathbb{R}^{n\times n}$:

$$\max_Y \ -F_0 \bullet Y \quad \text{subject to} \quad \forall i: \ F_i \bullet Y = c_i \quad \text{and} \quad Y \ \succeq \ 0. \tag{9}$$

It is known that the optimal value of the maximization dual SDP is never larger than the optimal value of the minimization primal counterpart. However, unlike linear programming, for semidefinite programs there may be a duality gap, i.e. the primal and/or dual might not attain their respective optima.

To prove that for any graph G Boppana's function $h(G)$ gives the same value as $h_p(G)$ we formulate the function h as a (primal) SDP. We provide also its dual program and prove that the optimum solutions of primal and dual are equal in this case. Then we show that the dual formulation of the Boppana's optimization is equivalent to the primal SDP defined by Feige and Kilian [18].

Below, $G = (V, E)$ denotes a graph, A the adjacency matrix of G and for a given vector d, as usually, let $D = \mathrm{diag}(d)$, for short. We provide the SDP for the function h (Eq. (7)) that differ slightly from that one given in [3].

▶ **Proposition 8.** *For any graph $G = (V, E)$, the objective function*

$$h(G) \;=\; \max_{d\in\mathbb{R}^n} \frac{\mathrm{sum}(A+D) - n\lambda((A+D)_S)}{4}$$

maximized by Boppana's algorithm can be characterized as an SDP as follows:

$$\begin{cases} p(G) = \min\limits_{z\in\mathbb{R},d\in\mathbb{R}^n} (nz - \mathbf{1}^T d) \quad \textit{subject to} \\ zI - A + \frac{JA+AJ}{n} - \frac{\mathrm{sum}(A)J}{n^2} - D + \frac{\mathbf{1}d^T + d\mathbf{1}^T}{n} - \frac{\mathrm{sum}(D)J}{n^2} \;\succeq\; 0, \end{cases} \tag{10}$$

with the relationship $h(G) = \frac{|E|}{2} - \frac{1}{4}p(G)$. The dual program to the program (10) can be expressed as follows:

$$\begin{cases} d(G) = \max\limits_{Y\in\mathbb{R}^{n\times n}} \left(A \bullet Y - \frac{1}{n}\sum_j \deg(j)\sum_i y_{ij} - \frac{1}{n}\sum_i \deg(i)\sum_j y_{ij} + \frac{1}{n^2}\sum_{i,j} y_{ij}\right) \\ \textit{subject to} \\ \qquad\qquad \sum_i y_{ii} \;=\; n, \\ \forall i \quad y_{ii} - \frac{1}{n}\sum_j y_{ji} - \frac{1}{n}\sum_j y_{ij} + \frac{1}{n^2}\sum_{k,j} y_{kj} \;=\; 1, \\ \qquad\qquad Y \;\succeq\; 0. \end{cases} \tag{11}$$

Using these formulations we prove that the primal and dual SDPs attain the same optima.

▶ **Theorem 9.** *For the semidefinite programs of Proposition 8 the optimal value p^* of the primal SDP (10) is equal to the optimal value d^* of the dual SDP (11). Moreover, there exists a feasible solution (z, d) achieving the optimal value p^*.*

Proof. Consider the primal SDP (10) of Boppana in the form

$$\min_{z\in\mathbb{R},d\in\mathbb{R}^n} z \quad \text{s.t.} \quad zI - M(d) \;\succeq\; 0,$$

with $M(d) = P(A + \mathrm{diag}(d))P - \frac{\mathbf{1}^T d}{n} I$ and, recall, $P = I - \frac{J}{n}$. Note that this formulation is equivalent to (10), as we have shown in the proof of Proposition 8. We show that this primal SDP problem is strictly feasible, i.e. that there exists an z' and an d' with $z'I - M(d') \succ 0$. To this aim we choose an arbitrary d' and then some $z' > \lambda(M(d'))$. From [45, Thm. 3.1], it follows that the optima of primal and dual obtain the same value.

To prove the second part of the theorem, i.e. there exists a feasible solution achieving the optimal value p^*, consider the following. The function $h(G)$ maximizes $g(G, d)$ over vectors $d \in \mathbb{R}^n$, while d can be restricted to vectors of mean zero. The function g is convex and goes to $-\infty$ for vectors d with some component going to ∞. Thus, g reaches its maximum at some finite d^{opt}. Now we choose $d = d^{\mathrm{opt}}$ and $z = \lambda(M(d^{\mathrm{opt}}))$. Clearly, this solution is feasible and obtains the optimal value p^*. ◀

For a graph $G = (V, E)$, Feige and Kilian express the minimum bisection problem as an SDP over an $n \times n$ matrix Y as follows:

$$h_p(G) = \min_{Y \in \mathbb{R}^{n \times n}} h_Y(G) \quad \text{s.t.} \quad \forall i \; y_{ii} = 1, \; \sum_{i,j} y_{ij} = 0, \text{ and } Y \succeq 0, \tag{12}$$

where $h_Y(G) = \sum_{\substack{\{i,j\} \in E \\ i<j}} \frac{1-y_{ij}}{2}$. For proving that the SDP takes as optimum the bisection width w.h.p. on $\mathcal{G}_n(p, q)$, the authors consider the dual of their SDP:

$$h_d(G) = \max_{x \in \mathbb{R}^n} \left(\frac{|E|}{2} + \frac{1}{4} \sum_i x_i \right) \quad \text{s.t.} \quad M = -A - x_0 J - \operatorname{diag}(x) \succeq 0, \tag{13}$$

where A is the adjacency matrix of G. They show that the dual takes the value of the bisection width w.h.p. and bounds the optimum of the primal SDP. Although we know that their SDP and Boppana's algorithm both work well on $\mathcal{G}_n(p, q)$, it was open so far how they are related to each other. Below we answer this question showing that the formulations are equivalent. We start with the following:

▶ **Theorem 10.** *The primal SDP* (12) *is equivalent to the dual SDP* (11)*, with the relationship* $h_p(G) = \frac{|E|}{2} - \frac{1}{4} d(G)$.

From Theorems 9 and 10 we get

▶ **Corollary 11.** *Let G be an arbitrary graph. Then for the lower bound $h(G)$ of Boppana's algorithm and for the objective functions $h_p(G)$ of the primal SDP* (12)*, resp. $h_d(G)$ of the dual SDP* (12) *of Feige and Kilian [18] it is true*

$$h(G) = h_p(G) = h_d(G).$$

Thus, the both algorithms provide for any graph G the same objective value. We want to point out another important fact: the bisection algorithm proposed in [18] use an SDP formulation, where the variables are a matrix with dimension $n \times n$. Thus, there are n^2 variables for a graph with n vertices. In contrast, Boppana's algorithm uses n variables in the convex optimization problem. If we consider the dual SDP, we again have only $n + 1$ variables. However, due to Corollary 11, we can't be better than Boppana's algorithm.

Abbe et al. [2] and independently Mossel et al. [36] have shown, that there is a sharp threshold phenomenon when considering the $\mathcal{G}_n(p, q)$ model with $p = \alpha \log(n)/n$ and $q = \beta \log(n)/n$ for fixed constants α, β, $\alpha > \beta$. Exact recovery of the planted bisection is possible if and only if $(\sqrt{\alpha} - \sqrt{\beta})^2 > 2$ (see e.g. [36] for a formal definition of exact cluster recovery problem). Hajek et al. [25] show, than an SDP equivalent to the one of Feige and Kilian achieves this bound. Since, due to Corollary 11, we know that the SDP is equivalent to Boppana's algorithm, we conclude that also Boppana's algorithm achieves the optimal threshold for finding and certifying the optimal bisection in the considered model. We get:

▶ **Theorem 12.** *Let α and β, $\alpha > \beta$, be constants. Consider the graph model $\mathcal{G}_n(p, q)$ with $p = \alpha \log(n)/n$ and $q = \beta \log(n)/n$. Then, as $n \to \infty$, if $(\sqrt{\alpha} - \sqrt{\beta})^2 > 2$, Boppana's algorithm recovers the planted bisection w.h.p. If $(\sqrt{\alpha} - \sqrt{\beta})^2 < 2$, no algorithm is able to recover the planted bisection w.h.p.*

Proof. The second part of the theorem is exactly the statement from [2]. The first part, i.e. that Boppana's algorithm is able to recover the bisection, follows from [25, Thm. 2]. Hajek et al. show, that for $(\sqrt{\alpha} - \sqrt{\beta})^2 > 2$ the SDP of Feige and Kilian obtain the optimal solution. Due to Theorem 10, the same holds for Boppana's algorithm. ◀

5 Certifying Non-Unique Optimum Bisections

From Section 2 we know that if the bound $h(G)$ is tight and the bisection of minimum size is unique, or more precisely the multiplicity of the largest eigenvector of B_S is 1, Boppana's algorithm is able to certify the optimality of the resulting bisection. We say that a graph G has a unique optimum bisection if there exists a unique, up to the sign, bisection vector x such that $\mathrm{cw}(x) = \mathrm{cw}(-x) = \mathrm{bw}(G)$. In this paper we also investigate families of graphs, different than random graphs $\mathcal{G}_n(p,q)$, for which Boppana's approach works well. To this aim we show a modification which handles cases such that $h(G) = \mathrm{bw}(G)$ but for which no unique bisection of minimum size exists. As we will see later hypercubes satisfy these two conditions. We present our algorithm below. Note that if the multiplicity of the largest eigenvalue of B_S^{opt} is 1, then the algorithm outputs the same result as in the original algorithm by Boppana.

1. Perform Step 1 of Algorithm 1; Let x be an eigenvector corresponding to the eigenvalue $\lambda((A+D)_S)$ and let k be the multiplicity of the largest eigenvalue of $(A+D)_S$
2. If $k = 1$ then construct a bisection vector $\hat{x}$ by splitting at the median $\bar{x}$ as in Step 2 of Algorithm 1; Next output $\hat{x}$ and if $\mathrm{cw}(\hat{x}) = h(G)$ output "optimum bisection" else output "fail"; If $k > 1$ then perform the steps below
3. Let $M \in \mathbb{R}^{n\times k}$ be the matrix with k linear independent eigenvectors corresponding to this largest eigenvalue; Transform the matrix to the reduced column echelon form, i. e. there are k rows which form an identity matrix, s.t. M still spans the same subspace
4. Brute force: for every combination of k coefficients from $\{+1,-1\}$ take the linear combination of the k vectors of M with the coefficients and verify if the resulting vector x is a bisection vector, i.e. $x \in \{+1,-1\}^n$ with $\sum_i x_i = 0$. If yes and if $\mathrm{cw}(x) = h(G)$ then output x and continue. This needs 2^k iterations
5. If in Step 4 no bisection vector x is given then output "fail".

▶ **Theorem 13.** *If $h(G) = \mathrm{bw}(G)$ then the algorithm above reconstructs all optimal bisections. Every achieved bisection vector corresponds to an optimal bisection.*

The eigenvalues for the family of hypercubes are explicitly known [26]. Hence, we can verify that the bound $h(G)$ is tight and Boppana's algorithm with the modification above works, i.e. finds an optimal bisection. For a hypercube H_n with n vertices we have $h(H_n) = g(H_n, (2-\log n)\mathbf{1}) = n/2 = \mathrm{bw}(H_n)$. Since the hypercube with n vertices has $\log n$ optimal bisections and the largest eigenspace of B_S has multiplicity $\log n$, the brute force part in our modification of Boppana's algorithm results in a linear factor of n for the overall runtime. Thus, the algorithm runs in polynomial time. With the results from Section 3 we can extend this result and obtain, that Boppana's algorithm with our modification works on adversarially modifieded hypercubes as well.

6 The Limitations of the Algorithm

Boppana shows, that his algorithm works well on some classes of random graphs. However, we do not know which graph properties force the algorithm to fail. For example, for the considered planted bisection model, we require a small bisection width. On the other hand, as we have seen in Section 5 Boppana's algorithm works for the hypercubes and their semirandom modifications – graphs that have large minimum bisection sizes.

In the following, we present newly discovered structural properties from inside the algorithm, which provide a framework for a better analysis of the algorithm itself. Let y be a bisection vector of G. We define

$$d^{(y)} = -\operatorname{diag}(y)Ay. \tag{14}$$

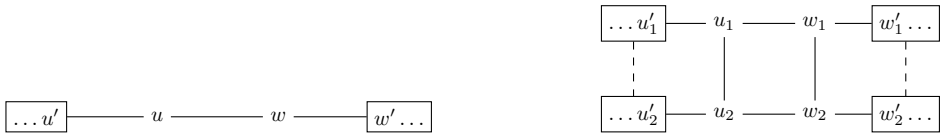


Figure 1 Forbidden graph structures as in Corollary 17 (left) and in Corollary 18 (right).

An equivalent but more intuitive characterization of $d^{(y)}$ is the following: $d_i^{(y)}$ is the difference between the number of adjacent vertices in other partition as vertex i and the number of adjacent vertices in same partition as i.

▶ **Lemma 14.** *Let G be a graph with $h(G) = \mathrm{bw}(G)$ and assume there is more than one optimum bisection in G. Then (up to constant translation vectors $c\mathbf{1}$) there exists a unique vector d^{opt} with $g(G, d^{\mathrm{opt}}) = \mathrm{bw}(G)$. Additionally, for every bisection vector y of an arbitrary optimum bisection in G there exists a unique $\alpha^{(y)}$ and the corresponding $d^{(y)}$, with $g(G, d^{(y)} + \alpha^{(y)} y) = \mathrm{bw}(G)$.*

Thus, if there are two optimum bisections represented by y and y' with $d^{(y)} \neq d^{(y')}$, then the difference of the d-vectors in component i is only dependent on y_i and y'_i, since we have $d^{(y)} - d^{(y')} = \beta' y' - \beta y$ for some constants β and β'. This structural property allows us to show the following limitation for the sparse planted partition model $\mathcal{G}_n(p, q)$.

▶ **Theorem 15.** *The algorithm of Boppana fails w.h.p. in the subcritical phase from [12], defined as $n(p-q) = \sqrt{np \cdot \gamma \ln n}$, for real $\gamma > 0$.*

In the planted partition model $\mathcal{G}_n(p, q)$, if the graphs are dense, e.g. $p = 1/n^c$ for a constant c with $0 < c < 1$, the constraints for the density difference $p - q$ assumed in Boppana's [9] and Coja-Oghlan's [12] algorithms are essentially the same. However for sparse graphs, e.g. such that $q = \mathcal{O}(1)/n$, the situation changes drastically. Now, e.g. $p = \sqrt{\log n}/n$ satisfy Coja-Oghlan's constraint $p - q \geq \Omega(\sqrt{p \ln(pn)}/\sqrt{n})$ but the condition on the difference $p - q$ assumed by Boppana is not true any more. Theorem 15 shows that Boppana's algorithm indeed fails under this setting. The proof of this theorem relies on the following observation, which can be derived from our newly discovered structural properties from above.

▶ **Lemma 16.** *Let G be a graph with $h(G) = \mathrm{bw}(G)$ and let (Y_1, Y_{-1}) be an arbitrary optimal bisection. Then, for each pair of vertices $v_i \in Y_i$, $i \in \{1, -1\}$, not connected by an edge ($\{v_i, v_{-i}\} \notin E$), we have: If $e(v_i, Y_i) = e(v_i, Y_{-i})$ for $i \in \{1, -1\}$ (the vertices have balanced degree), then $N(v_i) = N(v_{-i})$, i.e. both vertices have the same neighbors.*

I.e. if we have two balanced vertices in different parts of an optimal bisection, not connected by an edge, then the two vertices must have the same neighborhood as a necessary criterion for Boppana's algorithm to work. In the subcritical phase in Theorem 15, there exist most likely many of such pairs of vertices, but they are unlikely to have all even the same degree.

We can also provide forbidden substructures, which make Boppana's algorithm fail. This is e.g. the case, when the graph contains a path segment located on an optimal bisection:

▶ **Corollary 17.** *Let G be a graph, as illustrated in Fig. 1 (left), with $n \geq 10$ vertices containing a path segment $\{u', u\}, \{u, w\}, \{w, w'\}$, where u and w have no further edges. If there is an optimal bisection y, s. t. $y_u = y_{u'} = +1$ and $y_w = y_{w'} = -1$ (i. e. $\{u, w\}$ is a cut edge), then $h(G) < \mathrm{bw}(G)$.*

To prove this corollary, we use the more general but more technical Lemma 22 (Appendix of the full version [39]) with parameters $\tilde{C}_{+1} = \{u\}$ and $\tilde{C}_{-1} = \{w\}$. The result can also be applied for $2 \times c$ lattices:

▶ **Corollary 18.** *Let G be a graph with $n \geq 10c$ vertices containing a $2 \times c$ lattice with vertices u_i and w_i, as illustrated in Fig. 1 (right). (The construction is similar to the corollary above, but now we have a lattice instead of a single cut edge.) If there is an optimal bisection y, s. t. $y_{u_i} = y_{u'_i} = +1$ and $y_{w_i} = y_{w'_i} = -1$, then $h(G) < \mathrm{bw}(G)$.*

The algorithm fails if there are isolated vertices in both parts of an optimal bisection:

▶ **Theorem 19.** *Let G be a graph with $h(G) = \mathrm{bw}(G)$. Let G' be the graph G with two additional isolated vertices, then $h(G') \leq h(G) - \frac{4\,\mathrm{bw}(G)}{n^2}$.*

7 Discussion and Open Problems

Boppana's spectral method is a practically implementable heuristic. Computing eigenvalues and eigenvectors is well-studied and can be done very efficiently. Falkner, Rendl and Wolkowicz [17] show in a numerical study that using spectral techniques for graph partitioning is very robust and upper and lower bounds for the bisection width can be obtained such that the relative gap is often just a few percentage points apart. In [43] and [42], Tu, Shieh and Cheng present numerical experiments including results for Boppana's algorithm. They verify that the algorithm indeed has good average case behavior over certain probability distributions on graphs. We conducted further experiments on the graph model $\mathcal{R}_n(r, b)$ which indicated, that Boppana's algorithm also works for $r = 5$, but not for $r = 3$ and $r = 4$. An interesting question arising is, which properties of 3- and 4-regular graphs from the planted bisection model let the algorithm fail.

References

1. Emmanuel Abbe. Community detection and stochastic block models: recent developments. *arXiv preprint arXiv:1703.10146*, 2017.
2. Emmanuel Abbe, Afonso S. Bandeira, and Georgina Hall. Exact recovery in the stochastic block model. *IEEE Transactions on Information Theory*, 62(1):471–487, 2016.
3. Farid Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal on Optimization*, 5(1):13–51, 1995.
4. Sanjeev Arora, David Karger, and Marek Karpinski. Polynomial time approximation schemes for dense instances of NP-hard problems. In *Proc. of the 27th Annual ACM Symposium on Theory of Computing (STOC)*, pages 284–293. ACM, 1995.
5. Sandeep N Bhatt and Frank Thomson Leighton. A framework for solving VLSI graph layout problems. *Journal of Computer and System Sciences*, 28(2):300–343, 1984.
6. Avrim Blum and Joel Spencer. Coloring random and semi-random k-colorable graphs. *Journal of Algorithms*, 19(2):204–234, 1995.
7. Robert D. Blumofe. Spectral methods for bisecting graphs. Unpublished Manuscript, 1993.
8. Béla Bollobás and Alex D. Scott. Max cut for random graphs with a planted partition. *Combinatorics, Probability and Computing*, 13(4-5):451–474, 2004.
9. Ravi B. Boppana. Eigenvalues and graph bisection: An average-case analysis. In *Proc. of the 28th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 280–285. IEEE Computer Society, 1987. doi:10.1109/SFCS.1987.22.
10. T. N. Bui, S. Chaudhuri, F. T. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987. doi:10.1007/BF02579448.
11. Ted Carson and Russell Impagliazzo. Hill-climbing finds random planted bisections. In *Proc. of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 903–909. SIAM, 2001.

12 Amin Coja-Oghlan. A spectral heuristic for bisecting random graphs. In *Proc. of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 850–859. SIAM, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070552.

13 Amin Coja-Oghlan, Charilaos Efthymiou, and Nor Jaafari. Local convergence of random graph colorings. In *Proc. Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, (APPROX/RANDOM)*, volume 40 of *LIPIcs*, pages 726–737, 2015. URL: http://arxiv.org/abs/1505.02985.

14 Anne Condon and Richard M. Karp. Algorithms for graph partitioning on the planted partition model. *Random Structures and Algorithms*, 18(2):116–140, 2001.

15 Marek Cygan, Daniel Lokshtanov, Marcin Pilipczuk, MichałPilipczuk, and Saket Saurabh. Minimum bisection is fixed parameter tractable. In *Proc. of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 323–332. ACM, 2014. doi:10.1145/2591796.2591852.

16 Martin E. Dyer and Alan M. Frieze. The solution of some random NP-hard problems in polynomial expected time. *Journal of Algorithms*, 10(4):451–489, 1989.

17 Julie Falkner, Franz Rendl, and Henry Wolkowicz. A computational study of graph partitioning. *Mathematical Programming*, 66(1-3):211–239, 1994. doi:10.1007/BF01581147.

18 Uriel Feige and Joe Kilian. Heuristics for semirandom graph problems. *Journal of Computer and System Sciences*, 63(4):639–671, 2001. doi:10.1006/jcss.2001.1773.

19 Uriel Feige and Robert Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM Journal on Computing*, 31(4):1090–1118, April 2002. doi:10.1137/S0097539701387660.

20 Uriel Feige, Robert Krauthgamer, and Kobbi Nissim. Approximating the minimum bisection size. In *Proc. of the 32nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 530–536. ACM, 2000.

21 Miroslav Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(4):619–633, 1975.

22 Michael R. Garey, David S. Johnson, and Larry Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

23 M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981. doi:10.1007/BF02579273.

24 Stephen Guattery and Gary L. Miller. On the quality of spectral separators. *SIAM Journal on Matrix Analysis and Applications*, 19(3):701–719, July 1998. doi:10.1137/S0895479896312262.

25 Bruce Hajek, Yihong Wu, and Jiaming Xu. Achieving exact cluster recovery threshold via semidefinite programming. *IEEE Transactions on Information Theory*, 62(5):2788–2797, 2016.

26 Frank Harary, John P. Hayes, and Horng-Jyh Wu. A survey of the theory of hypercube graphs. *Computers and Mathematics with Applications*, 15(4):277–289, 1988.

27 Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social networks*, 5(2):109–137, 1983.

28 Subhash Khot. Ruling out PTAS for graph min-bisection, dense k-subgraph, and bipartite clique. *SIAM Journal on Computing*, 36(4):1025–1071, 2006.

29 Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: image and video synthesis using graph cuts. *ACM Transactions on Graphics (ToG)*, 22(3):277–286, 2003.

30 Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media, 2012.

31 Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, 1980.

32 Konstantin Makarychev, Yury Makarychev, and Aravindan Vijayaraghavan. Approximation algorithms for semi-random partitioning problems. In *Proc. of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pages 367–384. ACM, 2012.

33 Dániel Marx. Parameterized graph separation problems. *Theoretical Computer Science*, 351(3):394–406, 2006.

34 Frank McSherry. Spectral partitioning of random graphs. In *Proc. of the 42nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 529–537. IEEE, 2001.

35 Cristopher Moore. The computer science and physics of community detection: Landscapes, phase transitions, and hardness. *arXiv preprint arXiv:1702.00467*, 2017.

36 Elchanan Mossel, Joe Neeman, and Allan Sly. Consistency thresholds for the planted bisection model. In *Proc. of the 47th ACM Symposium on Theory of Computing (STOC)*, pages 69–75. ACM, 2015.

37 Huzur Saran and Vijay V. Vazirani. Finding k cuts within twice the optimal. *SIAM Journal on Computing*, 24(1):101–108, 1995.

38 Kirk Schloegel, George Karypis, and Vipin Kumar. *Graph partitioning for high performance scientific simulations*. Army High Performance Computing Research Center, 2000.

39 Martin R. Schuster and Maciej Liśkiewicz. New abilities and limitations of spectral graph bisection. *CoRR*, 2017. URL: `https://arxiv.org/abs/1701.01337`.

40 Daniel A. Spielman and Shang-Hua Teng. Spectral partitioning works: Planar graphs and finite element meshes. In *Proc. of the 37th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 96–105. IEEE Computer Society, 1996. URL: `http://dl.acm.org/citation.cfm?id=874062.875505`.

41 Daniel A. Spielman and Shang-Hua Teng. Spectral partitioning works: Planar graphs and finite element meshes. *Linear Algebra and its Applications*, 421(2):284–305, 2007. `doi:10.1016/j.laa.2006.07.020`.

42 Chih-Chien Tu and Hsuanjen Cheng. Spectral methods for graph bisection problems. *Computers & operations research*, 25(7):519–530, 1998. `doi:10.1016/S0305-0548(98)00021-5`.

43 Chih-Chien Tu, Ce-Kuen Shieh, and Hsuanjen Cheng. Algorithms for graph partitioning problems by means of eigenspace relaxations. *European Journal of Operational Research*, 123(1):86–104, 2000. `doi:10.1016/S0377-2217(99)00060-0`.

44 René van Bevern, Andreas Emil Feldmann, Manuel Sorge, and Ondřej Suchỳ. On the parameterized complexity of computing graph bisections. In *Proc. International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 76–87. Springer, 2013.

45 Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, March 1996. `doi:10.1137/1038003`.

46 Zhenyu Wu and Richard Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 15(11):1101–1113, 1993.

A Space-Optimal Grammar Compression*

Yoshimasa Takabatake[1], Tomohiro I[2], and Hiroshi Sakamoto[3]

1 **Kyushu Institute of Technology, Fukuoka, Japan**
`takabatake@ai.kyutech.ac.jp`
2 **Kyushu Institute of Technology, Fukuoka, Japan**
`tomohiro@ai.kyutech.ac.jp`
3 **Kyushu Institute of Technology, Fukuoka, Japan**
`hiroshi@ai.kyutech.ac.jp`

Abstract

A grammar compression is a context-free grammar (CFG) deriving a single string deterministically. For an input string of length N over an alphabet of size σ, the smallest CFG is $O(\lg N)$-approximable in the offline setting and $O(\lg N \lg^* N)$-approximable in the online setting. In addition, an information-theoretic lower bound for representing a CFG in Chomsky normal form of n variables is $\lg(n!/n^\sigma) + n + o(n)$ bits. Although there is an online grammar compression algorithm that directly computes the succinct encoding of its output CFG with $O(\lg N \lg^* N)$ approximation guarantee, the problem of optimizing its working space has remained open. We propose a fully-online algorithm that requires the fewest bits of working space asymptotically equal to the lower bound in $O(N \lg \lg n)$ compression time. In addition we propose several techniques to boost grammar compression and show their efficiency by computational experiments.

1998 ACM Subject Classification E.4 Coding and Information Theory

Keywords and phrases Grammar compression, fully-online algorithm, succinct data structure

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.67

1 Introduction

1.1 Motivation

Data never ceases to grow. Especially, we have witnessed so-called *highly-repetitive* text collections are rapidly increasing. Typical examples are genome sequences collected from similar species, version controlled documents and source codes in repositories. As such datasets are highly-compressible in nature, employing the power of data compression is the right way to process and analyze them. In order to catch up the speed of data increase, there is a strong demand for *fully online* and *really scalable* compression methods.

In this paper, we focus on the framework of grammar compression, in which a string is compressed into a context-free grammar (CFG) that derives the string deterministically [23]. In the last decade, grammar compression has been extensively studied from both theoretical and practical points of view: While it is mathematically clean, it can model many practical compressors such as LZ78 [48], LZW [47], LZD [13], repair [22], sequitor [33], and so on. Furthermore, there are wide varieties of algorithms working on grammar compressed strings, e.g., self-indexes [3, 9, 21, 25, 34, 38, 45, 46], pattern matching [10, 17], pattern mining [12, 8], machine learning [41], edit-distance computation [14, 43], and regularities detection [29, 15].

* This work was supported by JST CREST (Grant Number JPMJCR1402), and KAKENHI (Grant Numbers 17H01791 and 16K16009).

Table 1 Improvement of FOLCA: the fully-online grammar compression. Here N is the length of the input string received so far, σ and n are the numbers of alphabet symbols and generated variables, respectively, and $\frac{1}{\alpha} \geq 1$ is the load factor of the hash table.[2] For any input string, these algorithms construct the same SLP, which has $O(\lg N \lg^* N)$ approximation guarantee.

algorithm	compression time	working space (bits)
FOLCA ([28])	$O(\frac{N \lg n}{\alpha \lg \lg n})$ expected	$(1+\alpha)n \lg(n+\sigma) + n(3 + \lg(\alpha n)) + o(n)$
SOLCA (ours)	$O(N \lg \lg n)$ expected	$n \lg(n+\sigma) + o(n \lg(n+\sigma))$

Note that in order to take full advantage of these applications, a text should be compressed globally, that is, typical workarounds to memory limitation such as setting window-size or reusing variables (by forgetting previous ones) are prohibitive. This further motivates us to design really scalable grammar compression methods that can compress huge texts.

The primary goal of grammar compression is to build a small CFG that derives an input string only. The problem to build the smallest grammar is known to be NP-hard, but approximable within a reasonable ratio, e.g., $O(\lg N)$-approximable in the offline setting [23] and $O(\lg N \lg^* N)$-approximable[1] in the online setting [28], where N is input size and $\lg^*$ is the iterative logarithms.

On the other hand, to get a scalable grammar compression we have to seriously consider reducing the working space to fit into RAM. First of all, the algorithm should work in space comparable to the output CFG size. This has a great impact especially when we deal with highly-repetitive texts because output CFG size grows much slower than input size. We are aware of several work (including other compression scheme than grammar compression) addressing this [28, 13, 7, 20, 36, 35], but very few care about a constant factor hidden in big-O notation. More extremely and ideally, the output CFG should be encoded succinctly in an online fashion, and the algorithm should work in "succinct space", i.e., the encoded size plus lower order terms. To the best of our knowledge, fully-online LCA (FOLCA) [28] (and its variants) is the only existing algorithm addressing this problem. Whereas FOLCA achieved a significant improvement in memory consumption, there is still a gap between the memory consumption and its theoretical lower bound because FOLCA requires extra space for a hash table other than the succinct encoding of the CFG. Therefore the problem of optimizing the working space of FOLCA has been a challenging open problem.

In this paper, we tackle the above mentioned problem, resulting in the first space-optimal fully-online grammar compression. In doing so, we propose a novel succinct encoding that allows us to simulate the hash table of FOLCA in a dynamic environment. We further introduce two techniques to speed up compression. We call this improved algorithm *Space-Optimal FOLCA (SOLCA)*. See Table 1 for the improved time and space complexities. Experimental results show that both working space and running time are significantly improved from original FOLCA. We also compare our algorithm with other state-of-the-arts, and see that ours outperforms others in memory consumption, while the compression time is four to seven times slower than the fastest opponent.

[1] The authors in [28] only claimed $O(\lg^2 N)$ approximation, but it can be improved to $O(\lg N \lg^* N)$ adopting edit sensitive parsing (ESP) technique [4], which was pointed out in [40]. Naively the use of ESP adds $\lg^* N$ factor to computation time, but it can be eliminated by a neat trick of table lookup (e.g., see Theorem 6 of [8]). In practice, we have observed that the use of ESP does not improve the compression ratio much (or often even worsens), so our implementation still uses the algorithm with $O(\lg^2 N)$ approximation guarantee.

[2] In the previous papers, the inverse of the load factor is mistakenly referred to as the load factor. Here we fix the misuse.

1.2 Our Contribution in More Details

In the framework of grammar compression, an algorithm must refer to two data structures, the *dictionary* D (a set of production rules) and the *reverse dictionary* D^{-1}. Considering any symbol Z_i to be identical to integer i, D is regarded as an array such that $D[i]$ stores the phrase β if the production rule $Z_i \to \beta$ exists. Without loss of generality, we can assume that G is a *straight-line program (SLP)* [19] such that any β is a *bigram*, i.e., a pair of symbols (each symbol is a variable or an alphabet symbol). It follows that a naive representation of D occupies $2n \lg(n+\sigma)$ bits for n variables and σ alphabet symbols. Because an information-theoretic lower bound of SLP is $\lg((n+\sigma)!/n^{\sigma}) + 2(n+\sigma) + o(n)$ bits [42], the naive representation is highly redundant. Fully-online LCA (FOLCA) [28] is the first fully-online algorithm that directly outputs an encoded $e(D)$ whose size is asymptotically equal to the size of the optimal one.

On the other hand, given a phrase β, D^{-1} is required to return Z if $Z \to \beta$ exists. Using D^{-1}, a grammar compression algorithm can remember the existing name Z associated with β, i.e., we can avoid generating a useless $Z' \to \beta$ for the same β. In previous compression algorithms [26, 44, 42, 28], the reverse dictionary was simulated by a hash table whose size is comparable to the size of $e(D)$. This is the reason that the space optimization problem has remained open.

To solve this problem, we introduce a novel mechanism that allows FOLCA to directly compute D^{-1} by $e(D)$ with an auxiliary data structure in a dynamic environment. We develop a very simple data structure satisfying those requirements, and then we improve the working space of FOLCA. Note that the new data structure itself is independent from FOLCA/SOLCA, and applicable to any SLP for which fast access to both D and D^{-1} is required. Thus, it can be a new standard of succinct SLP encoding for such purposes.

FOLCA and SOLCA share the same idea to encode the topology of the derivation tree of the SLP by a succinct indexable dictionary, and heavily use it for simulating several navigational operations on the tree. As its operation time is the theoretical bottleneck of FOLCA, appearing as $O(\lg n / \lg \lg n)$ factor, we show that we can improve it to constant time. We then propose a practical implementation. Experimental results show that the improved version runs about 30% faster than original FOLCA.

Finally, we introduce a customized cache structure to grammar compression. The idea is inspired by the work [27] that proposed a variant of FOLCA working in constant space, in which only a constant number of frequently used variables are maintained to build SLP. Although the algorithm of [27] cannot make use of infrequent variables, it runs very fast as it is quite cache friendly. On the basis of this idea, we introduce a hash table (of size fitting into L3 cache) to lookup reverse dictionary for self-maintained frequent variables. Unlike [27], infrequent variables are looked up by the SOLCA's reverse dictionary. Experimental results show that this simple cache structure significantly improves the running time of plain SOLCA with a small overhead in space.

1.3 Related Work

There are compression algorithms with smaller space. For example, Maruyama and Tabei [27] proposed a variant of FOLCA working in constant space where the reverse dictionary with a fixed size is reset when the vacancy for a new entry runs out. We can find similar algorithms in constant space, e.g., repair, gzip, bzip, and etc. On the other hand, restricting the memory size not only saturates the compression ratio but also interferes with an important application like self-indexes [3, 9, 21, 25, 34, 38, 45, 46] because the memory is reset according

to the increase of input for the constant memory model. In fact, the SLP produced by FOLCA/SOLCA can be used for self-indexes [46], and for this application it is important that the whole text is globally compressed.

2 Framework of Grammar Compression

2.1 Notation

We assume finite sets Σ and V of symbols where $\Sigma \cap V = \emptyset$. Symbols in Σ and V are called *alphabet symbols* and *variables*, respectively. Σ^* is the set of all strings over Σ, and Σ^q the set of strings of length just q over Σ. The length of a string S is denoted by $|S|$. The i-th character of a string S is denoted by $S[i]$ for $i \in [1, |S|]$. For a string S and interval $[i, j]$ $(1 \leq i \leq j \leq |S|)$, let $S[i, j]$ denote the substring of S that begins at position i and ends at position j. Throughout this paper, we set $\sigma = |\Sigma|$, $n = |V|$ and $N = |S|$.

2.2 SLPs

We consider a special type of CFG $G = (\Sigma, V, D, X_s)$ where V is a finite subset of $\mathcal{X}$, D is a finite subset of $V \times (V \cup \Sigma)^*$, and $X_s \in V$ is the start symbol. A grammar compression of a string S is a CFG that derives only S deterministically, i.e., for any $X \in V$ there exists exactly one production rule in D and there is no loop.

We assume that G is an SLP [19]: any production rule is of the form $X_k \to X_i X_j$, where $X_i, X_j \in \Sigma \cup V$, and $1 \leq i, j < k \leq n + \sigma$. The size of an SLP is the number of variables, i.e., $|V|$, and we let $n = |V|$. For variable $X_i \in V$, $val(X_i)$ denotes the string derived from X_i. Also for $c \in \Sigma$, let $val(c) = c$. For $w \in (V \cup \Sigma)^*$, let $val(w) = val(w[1]) \cdots val(w[|w|])$.

The parse tree of G is a rooted ordered binary tree such that (i) the internal nodes are labeled by variables and (ii) the leaves are labeled by alphabet symbols. In a parse tree, any internal node Z corresponds to a production rule $Z \to XY$, where X (resp. Y) is the label of the left (resp. right) child of Z.

The set D of production rules is regarded as the data structure, called the dictionary, for accessing the phrase $X_i X_j$ for any X_k, if $X_k \to X_i X_j$ exists. On the other hand, the reverse dictionary D^{-1} is the data structure for accessing X_k for $X_i X_j$, if $X_k \to X_i X_j$ exists.

2.3 Succinct Data Structures

Here we introduce some succinct data structures, which we will use for encoding an SLP.

A rank/select dictionary for a bit string B [16] is a data structure supporting the following queries: $rank_c(B, i)$ returns the number of occurrences of $c \in \{0, 1\}$ in $B[1, i]$; $select_c(B, i)$ returns the position of the i-th occurrence of $c \in \{0, 1\}$ in B; $access(B, i)$ returns the i-th bit in B. There is a rank/select dictionary for B that uses $|B| + o(|B|)$ bits of space and supports the queries in $O(1)$ time [37]. In addition, the rank/select dictionary can be constructed from B in $O(|B|)$ time and $|B| + o(|B|) + O(1)$ bits of space.

It is natural to generalize the queries for a string T over an alphabet of size > 2. In particular, we consider the case where the alphabet size is $\Theta(|T|)$. Using a data structure called GMR [11], we obtain rank/select dictionary that occupies $|T| \lg |T| + o(|T| \lg |T|)$ bits of space and supports both rank and access queries in $O(\lg \lg (|T|))$ time and select queries in $O(1)$ time. Here we introduce the ingredients of the GMR for T (we remark that we use a simplified GMR as we consider only $\Theta(|T|)$-size alphabets), each of which we refer to as GMRDS1–4. Note that each query uses a distinct subset of them: $select_c(T, i)$ uses GMRDS1–2; $rank_c(T, i)$ uses GMRDS1–3; and $access(T, i)$ uses GMRDS1–2 and GMRDS4.

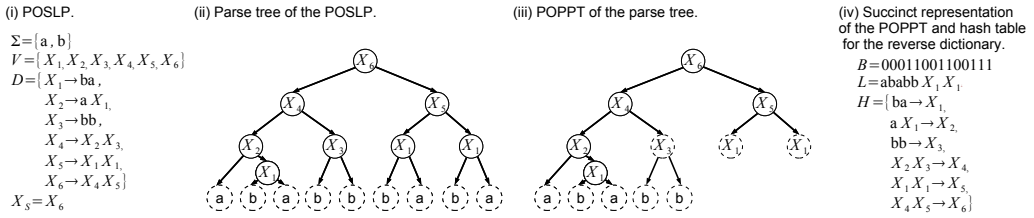


Figure 1 Example of post-order SLP (POSLP), parse tree, post-order partial parse tree (POPPT), and succinct representation of POPPT.

GMRDS1: A permutation π_T of $[1, |T|]$ obtained by stably sorting $[1, |T|]$ according to the values of $T[1, |T|]$. It is stored naively, and thus, occupies $|T| \lg |T|$ bits of space.

GMRDS2: A unary encoding of $T[\pi_T[1]]T[\pi_T[2]] \cdots T[\pi_T[|T|]]$ to support rank/select operations on $GB_T = 0^{T[\pi_T[1]]}10^{T[\pi_T[2]]-T[\pi_T[1]]}1 \ldots 0^{T[\pi_T[|T|]]-T[\pi_T[|T|-1]]}1$. The space usage is $O(|T|)$ bits.

GMRDS3: A data structure to support predecessor queries on sub-ranges of $\pi_T[1, |T|]$. Note that for any character c appearing in T there is a unique range $[i_c, j_c]$ s.t. $T[\pi_T[k]] = c$ iff $k \in [i_c, j_c]$. Also, the sequence $\pi_T[i_c], \pi_T[i_c + 1], \ldots, \pi_T[j_c]$ is non-decreasing. The task is, given such a range and an integer x, to compute the largest position $k \in [i_c, j_c]$ with $\pi_T[k] < x$ if such exists. We can employ y-fast trie to support the queries in $O(\lg \lg |T|)$ time by adding extra $O(|T|)$ bits on top of π_T (note that the search on bottom trees of y-fast trie can be implemented by simple binary search on a sub-range of π_T as we only consider static GMR).

GMRDS4: A data structure to support fast access to $\pi_T^{-1}[i]$ for any $1 \leq i \leq |T|$. We can use the data structure of [31] to compute $\pi_T^{-1}[i]$ in $O(\lg \lg |T|)$ time. It adds extra $O(|T| + \lg |T| / \lg \lg |T|)$ bits on top of π_T.

2.4 Online Construction of Succinct SLP

▶ **Definition 1** (POSLP and post-order partial parse tree (POPPT) [39, 26])**.** A partial parse tree is a binary tree built by traversing a parse tree in a depth-first manner and pruning all of the descendants under every node of a previously appearing nonterminal symbol. A POPPT is a partial parse tree whose internal nodes have post-order variables. A POSLP is an SLP whose partial parse tree is a POPPT.

Figures 1(i) and (iii) show an example of a POSLP and POPPT, respectively. The resulting POPPT (iii) has internal nodes consisting of post-order variables. FOLCA [28] is a fully-online grammar compression for directly computing the succinct POSLP (B, L) of a given string, where B is the bit string obtained by traversing POPPT in post-order, and putting '0', if a node is a leaf, and '1', otherwise, and L is the sequence of leaves of the POPPT. B encodes the topology of POPPT in $2n$ bits by taking advantage of the fact that POPPT is a full binary tree (note that for general trees we need $4n$ bits instead). By enhancing B with a data structure supporting some primitive operations considered in [32] (*fwdsearch* and *bwdsearch* on the so-called *excess array* of B), we can support some basic navigational operations (like move to parent/child) on the tree as well as rank/select queries on B. Using the dynamic data structure proposed in [32], we can support these operations as well as dynamic updates on B in $O(\lg n / \lg \lg n)$ time. In theory, FOLCA uses this result to get Theorem 2 (though its actual implementation uses a simplified version, which only has $O(\lg n)$-time guarantee).

▶ **Theorem 2** ([28]). *Given a string of length N over an alphabet of size σ, FOLCA computes a succinct POSLP of the string in $O(\frac{N \lg n}{\alpha \lg \lg n})$ expected time using $(1+\alpha)n \lg(n+\sigma)+n(3+\lg(\alpha n))$ bits of working space, where $\frac{1}{\alpha} \geq 1$ is the load factor of the hash table.*

In Section 3 we improve FOLCA in two ways: First, we improve the running time for operations on B from both theoretical and practical points of view in Subsection 3.1. Second, we slash $O(\alpha n \lg(n+\sigma))$ bits of working space of FOLCA needed for implementing D^{-1} by hash table. In Subsection 3.2, we propose a novel dynamic succinct POSLP to remove the redundant working space.

3 Improved Algorithm

3.1 Improving and Engineering Operations on B

Recall that FOLCA uses the dynamic tree data structure of [32], for which improving $O(\lg n / \lg \lg n)$ operation time is unlikely due to known lower bound. However, in our problem fully dynamic update operations are not needed as new tree topologies (bits) are always "appended". Therefore, in theory it is not difficult to get constant time operations: While appending bits, we mainly manage to update *range min-max trees* (*RmM-trees* in short) and a weighted level-ancestor data structure. For the former, it is fairly easy to fill up the min/max values for nodes of RmM-trees incrementally in worst case constant time per addition. For the latter, we can use the data structure of [1] supporting weighted level-ancestor queries and updates under adding leaf/root in worst case constant time. As a result, the running time of FOLCA can be improved to $O(N/\alpha)$ expected time.

Next we present a more practical implementation utilizing the fact that our B is well-balanced: Because FOLCA produces a well-balanced grammar, the resulting POPPT has height of at most $2 \lg N$. In our actual implementation, we allow the following overhead in space: We use some precomputed tables that occupy 2^8 bytes each so that some operations (like rank/select) on a single byte can be performed by a table lookup in constant time. Such tables are commonly used in modern implementations of succinct data structures (e.g., sdsl-lite `https://github.com/simongog/sdsl-lite`).

Now we briefly review the static data structure of [32]. Let E denote the excess array of B, i.e., for any $1 \leq i \leq n$, $E[i]$ is the difference of $\mathit{rank}_0(B,i)$ and $\mathit{rank}_1(B,i)$. Note that E is conceptual and we do not have a direct access to E. We consider a primitive query $\mathit{fwdsearch}(E,i,d)$ that returns the minimum $j > i$ such that $E[j] = E[i] + d$, where we assume $d \leq 0$ (it is simplified from the original *fwdsearch*, but enough for our problem). The data structure consists of three layers. The lowest layer partitions B into equal length mini-blocks of $\beta = \Theta(\lg N)$ bits. If query can be answered in a mini-block, it is processed by $O(\beta/8)$ table lookups, otherwise the query is passed to the middle layer. The middle layer partitions B into equal length block of $\beta' = \Theta(\lg^3 N)$ bits. Each block contains $O(\lg^2 N)$ mini-blocks and is managed by an RmM-tree. If the answer exists in a block, the RmM-tree identifies the right mini-block where the answer exists, otherwise the query is passed to the top layer. The task of the top layer is, given a block and target excess value e $(= E[i] + d)$, to find the nearest block (to the right for *fwdsearch*) whose minimum excess value is no greater than e, which is exactly the block where the answer exists.

Our ideas for a practical implementation are listed below:

- Since all excess values are in $[0, 2 \lg N]$, each node of RmM-trees can hold absolute excess value using $1 + \lg \lg N$ bits. (Note that in general case we only afford to store relative values, and thus, we have to retrieve absolute values by traversing from the root of the

tree when needed.) In particular, we can directly access absolute excess values at every ending position of mini-block by storing them in an array $E'[1, \lceil n/\beta \rceil]$, which only uses $O(n \lg\lg N/\beta) = O(n \lg\lg N/\lg N)$ bits.

- Since $rank_0(B, i) = (i - E[i])/2$ and $rank_1(B, i) = (i + E[i])/2$, rank queries are answered by computing $E[i]$, which can be now computed by accessing $E'[\lceil i/\beta \rceil]$ and $O(\lg N/8)$ table lookups.
- For select query $select_0(B, j)$ whose answer is i, we remark that $rank_0(B, i) = j = (i - E[i])/2$ holds. Since $i = 2j + E[i]$ and $E[i] \in [0, 2 \lg N]$, the answer i exists in $[2j, 2j + 2 \lg N]$. Thus, $select_0(B, j)$ can be computed by accessing $E'[\lceil 2j/\beta \rceil]$ and $O(\lg N/8)$ table lookups. Similarly, $select_1(B, j)$ can be answered by screening the range $[2j - 2 \lg N, 2j]$.
- For the top layer, we can simply remember, for every combination of block and target excess value, the answer for *fwdsearch* query. Since the number of possible combinations is $O(n \lg N/\beta')$, it takes $O(n \lg^2 N/\beta') = O(n/\lg N)$ bits.

3.2 Improved Dynamic Succinct POSLP

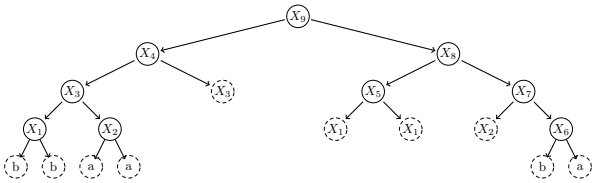
We propose a novel space-efficient representation of POSLP that occupies $n \lg(n + \sigma) + o(n \lg(n + \sigma))$ bits of space *including* the reverse dictionary. The concept of a succinct representation of POSLP is unchanged, but now we consider integrating the reverse dictionary into it.

We start with categorizing every production rule into two groups. A production rule $Z \to XY \in (V \cup \Sigma)^2$ (or variable Z) is said to be *outer*, if both children of the node corresponding to Z in the POPPT are leaves, and *inner*, otherwise. The reverse dictionaries for inner and outer variables are implemented differently. Particularly, the reverse dictionary for inner variables can be implemented *without* having any other data structures than (B, L) (see Section 3.2.1). Although we do not know which dictionary is to be used when looking up a phrase, it is sufficient to try them both.

The proposed dynamic succinct POSLP consists of the same (B, L) as the previous POSLP. The difference is the encoding of L: We partition L into L_1, L_2, and L_3 such that L_2 (resp. L_3) consists of every element of L that is a left (resp. right) child of an outer variable (preserving their original order), and L_1 consists of the remaining elements. In addition, we add functions $rank_{001}(B, i)$ and $select_{001}(B, i)$ to B, which return the number of occurrences of 001 in $B[1, i + 2]$ and the position of the i-th occurrence of 001 in B, respectively. Note that each occurrence of 001 corresponds to an occurrence of outer variable, and $rank_{001}/select_{001}$ enables us to map any leaf to the corresponding entry distributed to one of L_1, L_2 and L_3. More precisely, given any position i in B representing a leaf (i.e., $B[i] = 0$), the corresponding label is retrieved as follows: return $L_2[rank_{001}(B, i)]$, if $B[i, i + 2] = 001$; return $L_3[rank_{001}(B, i)]$, if $B[i - 1, i + 1] = 001$; and return $L_1[rank_0(B, i) - 2rank_{001}(B, i)]$, otherwise. While storing L_1 in a standard variable length array that supports pushback of elements, we store L_2 and L_3 implicitly in a data structure that provides the functionality of the reverse dictionary for outer variables.

Let n_{in} and n_{out} be the numbers of inner and outer variables, respectively, i.e., $n_{\mathsf{in}} = |L_1|$ and $n_{\mathsf{out}} = |L_2| = |L_3|$. Each of L_2 and L_3 is further partitioned into the prefix of length n'_{out} and the suffix of length $n_{\mathsf{out}} - n'_{\mathsf{out}}$ for some n'_{out} satisfying $n_{\mathsf{out}} - n'_{\mathsf{out}} < \frac{n_{\mathsf{out}}}{\lg\lg n_{\mathsf{out}}}$, that is, the suffixes are relatively short. Let π_2 be the permutation of $[1, n'_{\mathsf{out}}]$ obtained by sorting $[1, n'_{\mathsf{out}}]$ stably according to the values of $L_2[1, n'_{\mathsf{out}}]$, and let $\hat{L}_2 = L_2[\pi_2[1]]L_2[\pi_2[2]] \cdots L_2[\pi_2[n'_{\mathsf{out}}]]$ and $\hat{L}_3 = L_3[\pi_2[1]]L_3[\pi_2[2]] \cdots L_3[\pi_2[n'_{\mathsf{out}}]]$. Roughly we consider a two-stage GMR, the first for $L_2[1, n'_{\mathsf{out}}]$ and the second for $\hat{L}_3$ (although we only use select/access queries for

(i) Example of the POPPT.



(ii) Succinct representation of the POPPT.

$$B = 00100110100100011111$$
$$L = b, b, a, a, X_3, X_1, X_1, X_2, b, a$$

(iii) Decomposition of L: if the parent of $L[i]$ is inner, $L[i] \in L_1$, else if $L[i]$ is the left child, $L[i] \in L_2$, and otherwise, $L[i] \in L_3$.

$$L_1 = X_3, X_2$$
$$L_2 = b, a, X_1, b$$
$$L_3 = b, a, X_1, a$$

(iv) Encode of L: L_1 is represented by the integer array. The prefix $L_2[1, n'_{\mathsf{out}}]$ is represented by the bit array GB_2 and the permutation π_2 in GMR. The remaining short suffix of L_2 is represented by the integer array GA_2 (iv-1). In the GMR encoding of $L_2[1, n'_{\mathsf{out}}]$, $L_2[1, n'_{\mathsf{out}}]$ is sorted in lexicographical order and each $L_3[i]$ is sorted by the rank of $L_2[i]$ (iv-2). Then, L_3 is similarly encoded with n'_{out} dividing them into the suffix and prefix (iv-3). Additionally, the hash table h returns i ($i > n'_{\mathsf{out}}$) if $L_2[i] = X_j$ and $L_3[i] = X_k$ for the query X_jX_k (iv-4).

(iv-1) Data structure for L_2 ($n'_{\mathsf{out}} = 2$).
$GB_2 = 101$, $\pi_2 = 2, 1$
$GA_2 = X_1, b$

(iv-3) Data structure for L_3.
$GB_3 = 101$, $\pi_3 = 1, 2$
$GA_3 = X_1, a$

(iv-2) Sort $L_2[1, n'_{\mathsf{out}}]$ and $L_3[1, n'_{\mathsf{out}}]$ to $\hat{L}_2[1, n'_{\mathsf{out}}]$ and $\hat{L}_3[1, n'_{\mathsf{out}}]$, respectively.
$\hat{L}_2[1, n'_{\mathsf{out}}] = a, b$
$\hat{L}_3[1, n'_{\mathsf{out}}] = a, b$

(iv-4) Hash table for $L_2[i]$ and $L_3[i]$ ($i > n'_{\mathsf{out}}$).
$h = \{X_1X_1 \to 3, ba \to 4\}$

(v) The proposed dynamic succinct POPPT is formed by L_1 of (iii), (iv-1), (iv-3), and (iv-4).

Figure 2 Example of the proposed data structure for dynamic succinct POSLP.

$L_2[1, n'_{\mathsf{out}}]$). By the data structures, fitting in $2n'_{\mathsf{out}} \lg(n + \sigma) + o(n'_{\mathsf{out}} \lg(n + \sigma))$ bits of space in total, we can lookup a phrase of outer variables in $[1, n'_{\mathsf{out}}]$ in $O(\lg \lg n)$ time (see Section 3.2.2).

The reverse dictionary for the remaining outer variables (that are in short suffix) is implemented by dynamic perfect hashing [5] that occupies $O(\frac{n_{\mathsf{out}} \lg n_{\mathsf{out}}}{\lg \lg n_{\mathsf{out}}}) = o(n_{\mathsf{out}} \lg n_{\mathsf{out}})$ bits of space and supports lookup and addition in $O(1)$ expected time.

Note that we use "static" GMRs for $L_2[1, n'_{\mathsf{out}}]$ and $\hat{L}_3$. Since most dynamic updates of POSLP are supported by the hash (adding variables in the short suffix one by one), we do nothing to GMRs. When the short suffix becomes too long, i.e., $n_{\mathsf{out}} - n'_{\mathsf{out}}$ reach $\frac{n_{\mathsf{out}}}{\lg \lg n_{\mathsf{out}}}$, we increase n'_{out} (i.e., the number of variables managed by GMRs) by $\frac{n_{\mathsf{out}}}{\lg \lg n_{\mathsf{out}}}$ and just "reconstruct" the static GMRs from scratch (and clear all variables in the hash). Since the GMR for a string can be constructed in linear time to the length of the string, the total cost of reconstruction is $O(\frac{n}{\lg \lg n} \sum_{i=1}^{\lg \lg n} i) = O(n \lg \lg n)$.

Figure 2 shows an example of our POSLP.

In what follows we show how to implement the reverse dictionaries as well as access to the production rules of outer variables.

3.2.1 Reverse dictionary for inner variables

If there is an inner variable deriving XY, at least one of the following conditions holds, where v_X (resp. v_Y) is the corresponding node of X (resp. Y) in the POPPT:

- **(i)** v_X is a left child of its parent, and the parent has a right child (regardless of whether an internal node or leaf) representing Y, and
- **(ii)** v_Y is a right child of its parent, and the parent has a left child (regardless of whether an internal node or leaf) representing X.

Therefore, $D^{-1}(XY)$ can be looked up by a constant number of parent/child queries on B and access to L_1. Moreover, the next lemma suggests that we do not need to check both conditions (i) and (ii); check (ii), if $X < Y$, and check (i), otherwise.

▶ **Lemma 3.** *Let Z be an inner variable deriving $XY \in (V \cup \Sigma)^2$, and v_Z be the corresponding node of Z in the POPPT. If $X < Y$, the right child of v_Z is an internal node. Otherwise the left child of v_Z is an internal node.*

Proof. $X < Y$: Assume for the sake of contradiction that the right child of v_Z is a leaf (which represents Y). As Z is inner, the left child of v_Z must be the internal node corresponding to X. Since Y is larger than X and smaller than Z, the internal node corresponding to Y must be in the subtree rooted at the right child of v_Z, which contradicts the assumption.

$X \geq Y$: Assume for the sake of contradiction that the left child of v_Z is a leaf (which represents X). As Z is inner, the right child of v_Z must be the internal node corresponding to Y. Since the internal node corresponding to X appears before the left child of v_Z, $X < Y$ holds, a contradiction. ◀

Due to Lemma 3 and the above discussions, we get the following lemma.

▶ **Lemma 4.** *We can implement the reverse dictionary for inner variables that supports lookup in $O(1)$ time.*

3.2.2 Reverse dictionary for outer variables

▶ **Lemma 5.** *We can implement the reverse dictionary for outer variables to support lookup in $O(\lg \lg n)$ expected time.*

Proof. Recall that for any $1 \leq i \leq n'_{\mathsf{out}}$ the pair $L_2[i]L_3[i]$ is the right-hand side of the i-th outer production rule (in post-order). Given i, we can compute the post-order number of the variable deriving $L_2[i]L_3[i]$ by $rank_1(B, select_{001}(B, i)) + 1$. Hence, the task of our reverse dictionary is, given $XY \in (V \cup \Sigma)^2$, to return integer i such that $L_2[i] = X$ and $L_3[i] = Y$, if such exists. If a phrase is found in the short suffix, the query is answered in $O(1)$ expected time by using hash table. Thus, in what follows, we focus on the case where the answer is not found in the short suffix.

By the GMRDS2 GB_2 for $L_2[1, m']$, we can compute in constant time, given an integer X, the range $[i_X, j_X]$ in π_2 such that the occurrences of X in L_2 is represented by $\pi_2[i_X, j_X]$ in increasing order, namely, $i_X = rank_1(GB_2, select_0(GB_2, X)) + 1$ and $j_X = rank_1(GB_2, select_0(GB_2, X + 1))$. Note that Y occurs in $\hat{L}_3[i_X, j_X]$ (the occurrence is unique) iff there is an outer variable deriving XY. In addition, if $k \in [i_X, j_X]$ is the occurrence of Y, then $\pi_2[k]$ is the post-order number of the variable we seek. Hence, the

Table 2 Detail of memory consumption (MB).

	Wikipedia				genome			
method	B	L	H	CRD	B	L	H	CRD
FOLCA	17.63	180.06	1342.43	–	141.00	1247.67	9442.64	–
FOLCA+	17.26	180.06	1342.43	–	138.09	1247.67	9442.64	–
SOLCA	17.26	523.85	–	–	138.09	3856.84	–	–
SOLCA+CRD	17.26	523.85	–	22.00	138.09	3856.84	–	22.00

problem reduces to computing $select_Y(\hat{L}_3, rank_Y(\hat{L}_3, i_X - 1) + 1)$, which can be performed in $O(\lg \lg n)$ time by using the GMR for $\hat{L}_3$. ◀

3.2.3 Access to the production rules of outer variables

Since L_2 and L_3 are stored implicitly, here we show how to access the production rules of outer variables.

▶ **Lemma 6.** *Given $1 \leq i \leq n_{\mathsf{out}}$, we can access $L_2[i]L_3[i]$ in $O(\lg \lg n)$ time.*

Proof. If $i > n'_{\mathsf{out}}$, $L_2[i]L_3[i]$ is in the short suffixes. As we can afford to store $L_2[i]L_3[i]$ in a plain array of $O(\frac{n_{\mathsf{out}} \lg n_{\mathsf{out}}}{\lg \lg n_{\mathsf{out}}}) = o(n_{\mathsf{out}} \lg n_{\mathsf{out}})$ bits of space, we can access it in $O(1)$ time.

If $i \leq n'_{\mathsf{out}}$, $L_2[i]L_3[i]$ is represented by GMRs for $L_2[1, n_{\mathsf{out}}]$ and $\hat{L}_3$. Using GMRDS4 for $L_2[1, n_{\mathsf{out}}]$, we can compute $j = \pi_2^{-1}[i]$ in $O(\lg \lg n)$ time. Then, we can obtain $L_2[i]$ by $rank_0(GB_2, select_1(GB_2, j))$ in $O(1)$ time. In addition, $L_3[i]$ can be retrieved by accessing $\hat{L}_3[j]$, which is supported in $O(\lg \lg n)$ time by GMR for $\hat{L}_3$. ◀

To tell the truth, SOLCA does not access the production rules of outer variables during compression, and hence, the implementation of SOLCA is further simplified by deleting GMRDS4 for both $L_2[1, n_{\mathsf{out}}]$ and $\hat{L}_3$, needed to support access queries on the GMRs.

3.3 SOLCA

Plugging our new succinct representation of POSLP into FOLCA, we get a space-optimal grammar compression algorithm, SOLCA.

▶ **Theorem 7.** *Given a string of length N over an alphabet of size σ, SOLCA computes a succinct POSLP of the string in $O(N \lg \lg n)$ expected time using $n \lg(n + \sigma) + o(n \lg(n + \sigma))$ bits of working space.*

Proof. SOLCA processes the input string online exactly the same as FOLCA does. During compression, it is required to lookup a phrase by the reverse dictionary and append new variables to POSLP if the phrase does not exist so far. By Lemmas 4 and 5, this is done in $O(\lg \lg n)$ expected time. Our dynamic succinct POSLP including the reverse dictionary takes only $n \lg(n + \sigma) + o(n \lg(n + \sigma))$ bits of space as described in Section 3.2. ◀

4 Experiments

We implement FOLCA applying the dynamic succinct tree representation introduced in Section. 3.1 called FOLCA+ and the SOLCA proposed in Section 3.3.[3] Furthermore, as

[3] Currently we do not implement the last idea of Section 3.1 for *fwdsearch* queries. Instead we answer queries by traversing up a tree (so called 2D-Min-Heap [6]) built on the minimum excess values of

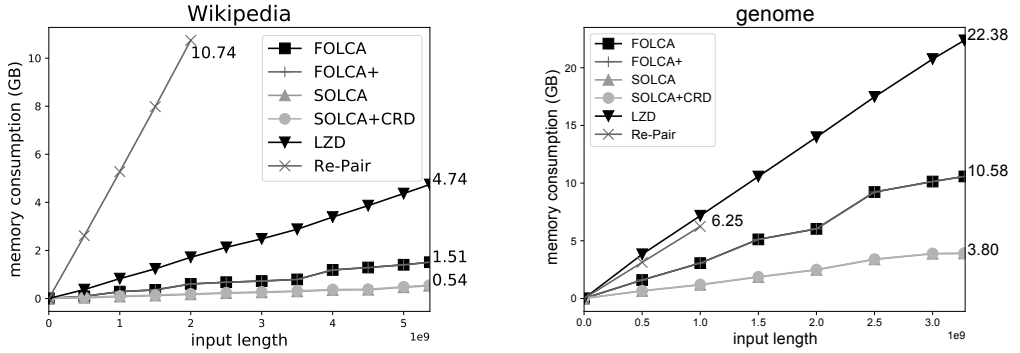


Figure 3 Working space for Wikipedia (left) and genome (right).

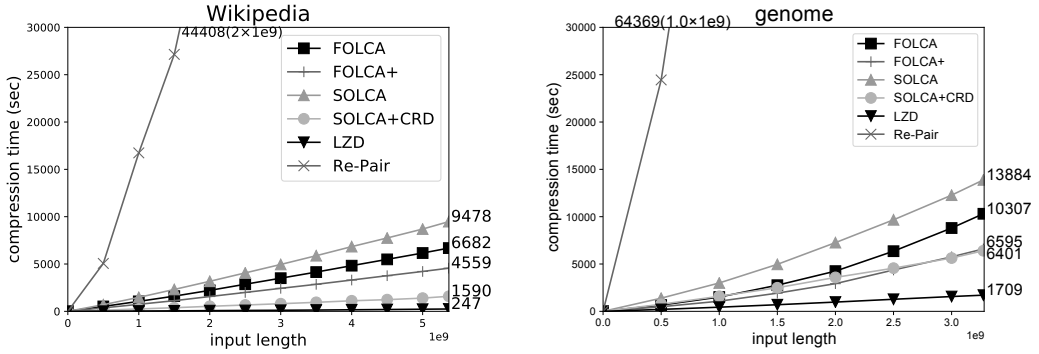


Figure 4 Compression time for Wikipedia (left) and genome (right).

a practical method for the fast computation of SOLCA, we implement the SOLCA with the constant space reverse dictionary (CRD) storing frequent production rules. We call it SOLCA+CRD[4]. The CRD is proposed in [27] and it supports the reverse dictionary query in constant expected time while keeping a constant space by constant space algorithms for finding frequent items [18, 24, 30]. The reverse dictionary query of SOLCA+CRD is performed by two phases: (1) we check if a given X_iX_j exists in the CRD and (2) if the X_iX_j is not found in phase (1), we check the reverse dictionary of SOLCA. Although the worst case time of the reverse dictionary query of SOLCA+CRD is the same as SOLCA's $O(\lg\lg(n+\sigma))$ time, if the query rule exists in the CRD, we can support the query in constant expected time. Our implementation of CRD is based on [18] and restricts the space to 22MB that is almost the same cache size of experimental machine. We compare the time/space consumption of these variants of FOLCA with that of existing three grammar compression algorithms: FOLCA, LZD [5] [13] and Re-Pair [6] [2]. The Re-Pair is a space-efficient version of the original algorithm [22]. The experiments perform on Intel Xeon Processor E7-8837

blocks. In the worst case it requires an $O(\lg N)$-long traversal, but it works well enough in practice as performing such a long traversal is rare.

[4] This implementation is downloadable from `https://github.com/tkbtkysms/solca`. We will show additional experiments in this web site.

[5] The patricia trie space computation (the compress function of the class STree::Tree) in `https://github.com/kg86/lzd`

[6] `https://github.com/nicolaprezza/Re-Pair`

Table 3 Statistical information of input strings.

dataset	length of string (N)	alphabets (σ)	compression ratio (%)		
			SOLCA	LZD	Re-Pair
Wikipedia	5,368,709,120	210	3.65	3.46	0.62[9]
genome	3,273,481,150	20	41.38	36.34	9.05[10]

(2.67GHz, 24MB cache, 8 cores) and 1TB RAM. Here, the load factor of the hash table used in FOLCA is fixed to $\frac{1}{\alpha} = 1$.

We use two large-scale datasets: Wikipedia[7] (5GB) and genome[8] (3GB). The detail is shown in Table 3 where we note that POSLP by SOLCA is exactly the same as FOLCA's. The difference is only their succinct representations.

Figure 3 shows a comparison of the memory consumption of each method for Wikipedia and genome. The points are displayed for every length of 5×10^8. FOLCA and FOLCA+ maintain data structure (B, L, H); B is the skeleton of POSLP T, L is the sequence of the leaves of T, and H is the reverse dictionary. When $\alpha = 1$, H occupies almost $2n \lg(n + \sigma)$ bits. Since the size of B and L is $n \lg(n + \sigma)$ bits and $2n$ bits, respectively, the total space of FOLCA's variants is about $3n \lg(n + \sigma)$ bits. On the other hand, SOLCA and SOLCA+CRD maintains (B, L') supporting the reverse dictionary; L' is the representation of L in Section 3.2. The size is almost the same as L. Thus, it is expected that the memory consumption of SOLCA and SOLCA+CRD is about $\frac{1}{3}$ of FOLCA's. The experimental result confirms this prediction on both datasets. Furthermore, the memory consumption of each data structure is shown in Table 2. Comparing with other methods, the space of SOLCA and SOLCA+CRD is significantly small for each string.

Figure 4 shows a comparison of the construction time for the input. Our succinct tree representation used in FOLCA+ improves the time consumption of FOLCA. The difference of SOLCA from FOLCA+ comes from the use of L' (queries to L' and reconstruction of L'). SOLCA+CRD is fastest in FOLCA's and SOLCA's variants for Wikipedia and competitive with FOLCA+ for genome. By this result, we can confirm the efficiency of the fast computation of CRD. SOLCA's and FOLCA's variants are faster than Re-pair and slower than LZD.

5 Conclusion

We have presented SOLCA: a space-optimal version of fully-online LCA (FOLCA) [28]. Since FOLCA is extended to its self-index in [46], our future work is developing a self-index based on our SOLCA while preserving the optimal working space.

References

1 Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *27th International Colloquium on Automata, Languages and Programming*, pages 73–84, 2000. doi:10.1007/3-540-45022-X_8.

[7] `https://dumps.wikimedia.org/enwikinews/20170101/enwikinews-20170101-pages-meta-history.xml` (the first 5GB)
[8] `http://hgdownload.cse.ucsc.edu/goldenPath/hg38/chromosomes/chr*`
[9] Up to the length of 2.0×10^9.
[10] Up to the length of 1.0×10^9.

2 Philip Bille, Inge Li Gørtz, and Nicola Prezza. Space-efficient re-pair compression. In *Data Compression Conference*, pages 171–180, 2017. doi:10.1109/DCC.2017.24.

3 Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundam. Inform.*, 111(3):313–337, 2011. doi:10.3233/FI-2011-565.

4 Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007. doi:10.1145/1219944.1219947.

5 Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994. doi:10.1137/S0097539791194094.

6 Johannes Fischer. Optimal succinctness for range minimum queries. In *Theoretical Informatics, 9th Latin American Symposium, LATIN 2010, Oaxaca, Mexico, April 19-23, 2010. Proceedings*, pages 158–169, 2010. doi:10.1007/978-3-642-12200-2_16.

7 Johannes Fischer, Travis Gagie, Pawel Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *23rd Annual European Symposium on Algorithms*, pages 533–544, 2015. doi:10.1007/978-3-662-48350-3_45.

8 Shouhei Fukunaga, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. Online grammar compression for frequent pattern discovery. In *13th International Conference on Grammatical Inference*, pages 93–104, 2016.

9 Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A faster grammar-based self-index. In *6th International Conference Language and Automata Theory and Applications*, pages 240–251, 2012. doi:10.1007/978-3-642-28332-1_21.

10 Pawel Gawrychowski. Optimal pattern matching in LZW compressed strings. *ACM Trans. Algorithms*, 9(3):25:1–25:17, 2013. doi:10.1145/2483699.2483705.

11 Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 368–373, 2006.

12 Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Fast q-gram mining on SLP compressed strings. *J. Discrete Algorithms*, 18:89–99, 2013. doi:10.1016/j.jda.2012.07.006.

13 Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. LZD factorization: Simple and practical online grammar compression with variable-to-fixed encoding. In *26th Annual Symposium on Combinatorial Pattern Matching*, pages 219–230, 2015. doi:10.1007/978-3-319-19929-0_19.

14 Danny Hermelin, Gad M. Landau, Shir Landau, and Oren Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. In *26th International Symposium on Theoretical Aspects of Computer Science*, pages 529–540, 2009. doi:10.4230/LIPIcs.STACS.2009.1804.

15 Tomohiro I, Wataru Matsubara, Kouji Shimohira, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, Kazuyuki Narisawa, and Ayumi Shinohara. Detecting regularities on grammar-compressed strings. *Inf. Comput.*, 240:74–89, 2015. doi:10.1016/j.ic.2014.09.009.

16 Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.

17 Artur Jez. Faster fully compressed pattern matching by recompression. *ACM Trans. Algorithms*, 11(3):20:1–20:43, 2015. doi:10.1145/2631920.

18 Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003. doi:10.1145/762471.762473.

19 Marek Karpinski, Wojciech Rytter, and Ayumi Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nord. J. Comput.*, 4(2):172–186, 1997.

20 Dominik Kempa and Dmitry Kosolobov. Lz-end parsing in compressed space. In *Data Compression Conference*, pages 350–359, 2017. doi:10.1109/DCC.2017.73.

21 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013. doi:10.1016/j.tcs.2012.02.006.

22 N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000. doi:10.1109/5.892708.

23 Eric Lehman. *Approximation algorithms for grammar-based data compression*. PhD thesis, MIT, Cambridge, MA, USA, 2002.

24 Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *28th International Conference on Very Large Data Bases*, pages 346–357, 2002.

25 Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. Esp-index: A compressed index based on edit-sensitive parsing. *J. Discrete Algorithms*, 18:100–112, 2013. doi:10.1016/j.jda.2012.07.009.

26 Shirou Maruyama, Hiroshi Sakamoto, and Masayuki Takeda. An online algorithm for lightweight grammar-based compression. *Algorithms*, 5(2):214–235, 2012. doi:10.3390/a5020214.

27 Shirou Maruyama and Yasuo Tabei. Fully online grammar compression in constant space. In *Data Compression Conference*, pages 173–182, 2014. doi:10.1109/DCC.2014.69.

28 Shirou Maruyama, Yasuo Tabei, Hiroshi Sakamoto, and Kunihiko Sadakane. Fully-online grammar compression. In *20th International Symposium on String Processing and Information Retrieval*, pages 218–229, 2013. doi:10.1007/978-3-319-02432-5_25.

29 Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8-10):900–913, 2009. doi:10.1016/j.tcs.2008.12.016.

30 Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *10th International Conference on Database Theory*, pages 398–412, 2005. doi:10.1007/978-3-540-30570-5_27.

31 J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theor. Comput. Sci.*, 438:74–88, 2012. doi:10.1016/j.tcs.2012.03.005.

32 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014. doi:10.1145/2601073.

33 Craig G. Nevill-Manning and Ian H. Witten. Compression and explanation using hierarchical grammars. *Comput. J.*, 40(2/3):103–116, 1997.

34 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. In *Prague Stringology Conference*, pages 158–170, 2016.

35 Tatsuya Ohno, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online run-length Burrows-Wheeler Transform. In *28th International Workshop on Combinatorial Algorithms (to appear)*, 2017.

36 Alberto Policriti and Nicola Prezza. Computing LZ77 in run-compressed space. In *2016 Data Compression Conference*, pages 23–32, 2016. doi:10.1109/DCC.2016.30.

37 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

38 Luís M. S. Russo and Arlindo L. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Inf. Retr.*, 11(4):359–388, 2008. doi:10.1007/s10791-008-9050-3.

39 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

40 Hiroshi Sakamoto, Shirou Maruyama, Takuya Kida, and Shinichi Shimozono. A space-saving approximation algorithm for grammar-based compression. *IEICE Transactions*, 92-D(2):158–165, 2009. doi:10.1587/transinf.E92.D.158.

41 Yasuo Tabei, Hiroto Saigo, Yoshihiro Yamanishi, and Simon J. Puglisi. Scalable partial least squares regression on grammar-compressed data matrices. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1875–1884, 2016. doi:10.1145/2939672.2939864.

42 Yasuo Tabei, Yoshimasa Takabatake, and Hiroshi Sakamoto. A succinct grammar compression. In *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*, pages 235–246, 2013. doi:10.1007/978-3-642-38905-4_23.

43 Yoshimasa Takabatake, Kenta Nakashima, Tetsuji Kuboyama, Yasuo Tabei, and Hiroshi Sakamoto. siedm: An efficient string index and search algorithm for edit distance with moves. *Algorithms*, 9(2):26, 2016. doi:10.3390/a9020026.

44 Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Variable-length codes for space-efficient grammar-based compression. In *19th International Symposium on String Processing and Information Retrieval*, pages 398–410, 2012. doi:10.1007/978-3-642-34109-0_42.

45 Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Improved esp-index: A practical self-index for highly repetitive texts. In *13th International Symposium on Experimental Algorithms*, pages 338–350, 2014. doi:10.1007/978-3-319-07959-2_29.

46 Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Online self-indexed grammar compression. In *22nd International Symposium on String Processing and Information Retrieval*, pages 258–269, 2015. doi:10.1007/978-3-319-23826-5_25.

47 Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984. doi:10.1109/MC.1984.1659158.

48 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978. doi:10.1109/TIT.1978.1055934.

Positive-Instance Driven Dynamic Programming for Treewidth*

Hisao Tamaki

Department of Computer Science, Meiji University, Kawasaki, Japan
tamaki@cs.meiji.ac.jp

Abstract

Consider a dynamic programming scheme for a decision problem in which all subproblems involved are also decision problems. An implementation of such a scheme is *positive-instance driven* (PID), if it generates positive subproblem instances, but not negative ones, building each on smaller positive instances.

We take the dynamic programming scheme due to Bouchitté and Todinca for treewidth computation, which is based on minimal separators and potential maximal cliques, and design a variant (for the decision version of the problem) with a natural PID implementation. The resulting algorithm performs extremely well: it solves a number of standard benchmark instances for which the optimal solutions have not previously been known. Incorporating a new heuristic algorithm for detecting safe separators, it also solves all of the 100 public instances posed by the exact treewidth track in PACE 2017, a competition on algorithm implementation.

We describe the algorithm and prove its correctness. We also perform an experimental analysis counting combinatorial structures involved, which gives insights into the advantage of our approach over more conventional approaches and points to the future direction of theoretical and engineering research on treewidth computation.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases treewidth, dynamic programming, minimal separators, potential maximal cliques, positive instances

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.68

1 Introduction

Suppose we design a dynamic programming algorithm for some decision problem, formulating subproblems, which are decision problems as well, and recurrences among those subproblems. A standard approach is to list all subproblem instances from “small" ones to “large" and scan the list, deciding the answer, positive or negative, to each instance by means of these recurrences. When the number of positive subproblem instances are expected to be much smaller than the total number of subproblem instances, a natural alternative is to generate positive instances only, using recurrences to combine positive instance to generate a “larger" positive instance. We call such a mode of dynamic programming execution *positive-instance driven* or *PID* for short. One goal of this paper is to demonstrate that PID is not simply a low-level implementation strategy but can be a paradigm of algorithm design for some problems.

The decision problem we consider is that of deciding, given graph G and positive integer k, if the treewidth of G is at most k. This graph parameter was introduced by Robertson and

* Full paper available as an arXiv preprint, https://arxiv.org/abs/1704.05286.

25th Annual European Symposium on Algorithms (ESA 2017).
Editors: Kirk Pruhs and Christian Sohler; Article No. 68; pp. 68:1–68:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Seymour [17] and has had a tremendous impact on graph theory and on the design of graph algorithms (see, for example, a survey [7].) The treewidth problem is NP-complete [1] but fixed-parameter tractable: it has an $f(k)n^{O(1)}$ time algorithm for some fixed function $f(k)$ as implied by the graph minor theorem of Robertson and Seymour [18], and explicit $O(f(k)n)$ time algorithm is given by Bodlaender [3]. A classical dynamic programming algorithm due to Arnborg, Corneil, and Proskurowsky (ACP algorithm) [1] runs in $n^{k+O(1)}$ time. Bouchitté and Todinca [9] developed a more refined dynamic programming algorithm (BT algorithm) based on the notions of minimal separators and potential maximal cliques, which lead to algorithms running in $O(1.7549^n)$ time or in $O(n^5\binom{\lceil (2n+k+8)/3 \rceil}{k+2})$ time [11, 12].

Another important approach to treewidth computation is based on the perfect elimination order (PEO) of a minimal chordal completion of the given graph. PEO-based dynamic programming algorithms run in $O^*(2^n)$ time with exponential space and in $O^*(4^n)$ time with polynomial space [5], where $O^*(f(n))$ means $O(n^c f(n))$ for some constant c.

There has been a considerable amount of effort on implementing treewidth algorithms to be used in practice and, prior to this work, the most successful implementations for exact treewidth computation are all based on PEO. The authors of [5] implemented the $O^*(2^n)$ time dynamic programming algorithm and experimented on its performance, showing that it works well for small instances. For larger instances, PEO-based branch-and-bound algorithms are known to work well in practice [14]. Recent proposals for reducing treewidth computation to SAT solving are also based on PEO [19, 2].

From the PID perspective, this situation is somewhat surprising, for the following reasons. Let us first review the PEO approach. See [5], for example, for details. Let G be the input graph. Recall that a PEO of G is a total order $v_1, \ldots, v_n$ on $V(G)$ such that, for $1 \leq i \leq n$, v_i is simplicial in $G[V_i]$, where $V_i = \{v_i, \ldots, v_n\}$ and a vertex is *simplicial* in a graph if its neighbors form a clique. A graph is *chordal* if it has no induced cycle of length four or greater. A *chordal completion* of G is a chordal supergraph of G with vertex set $V(G)$. The above PEO-based algorithms utilizes two facts: that every chordal graph has a PEO and that, for chordal graphs, the optimal tree-decomposition consists of all maximal cliques as bags. Thus, these algorithms look for a total order of $V(G)$ that is a PEO of a chordal completion of G whose optimal tree-decomposition is an optimal tree-decomposition of G. The dynamic programming algorithm reduces the search space size from the naive $O(n!)$ to $O(2^n)$ applying the Held-Karp paradigm for sequencing problems [4]. In the decision problem version, it consists in defining the "feasibility" of each subset of $V(G)$, to be inductively decided by dynamic programming. Informally, $S \subseteq V(G)$ is *feasible* if it has a total ordering that qualifies as a prefix of a total ordering of $V(G)$ that gives a chordal completion with the clique number k or smaller. This feasibility notion, however, has a more direct interpretation in terms of tree-decompositions: S is feasible if each connected component of $G[S]$ is feasible and each connected vertex set C is feasible if $G[C \cup N(C)]$, where $N(C)$ is the open neighborhood of S, has a tree-decomposition of width k or smaller that has a bag containing $N(C)$. This feasibility of connected sets is nothing but the feasibility considered in the classical ACP algorithm. Thus, each positive subproblem instance in the PEO-based dynamic programming scheme corresponds to a combination of an indefinite number of positive subproblem instances in the ACP algorithm, and hence the number of positive subproblem instances can be exponentially larger than that in the ACP algorithm. Indeed, a PID variant of the ACP algorithm was implemented by the present author and has won the first place in the exact treewidth track of PACE 2016 [10], a competition on algorithm implementations, outperforming other submissions based on PEO. Given this success, a natural next step is to design a PID variant of the BT algorithm, which is tackled in this paper.

The resulting algorithm performs extremely well, as reported in Section 7. It is tested on DIMACS graph-coloring instances [15], which have been used in the literature as standard benchmark instances [14, 8, 16, 19, 5, 2]. Our implementation of the algorithm solves all the instances that have been previously solved (that is, with matching upper and lower bounds known) within 10 seconds per instance on a typical desktop computer and solves 13 out of the 42 previously unsolved instances. For nearly half of the instances which it leaves unsolved, it significantly reduces the gap between the lower and upper bounds. It is interesting to note that this is done by improving the lower bound. Since the number of positive subproblem instances are much smaller when k is below the treewidth than when k equals the treewidth, the PID approach is particularly good at establishing strong lower bounds.

We also adopt the notion of safe separators due to Bodlaender and Koster [6] in our preprocessing and design a new heuristic algorithm for detecting safe separators. With this preprocessing, our implementation also solves all of the 100 public instances posed by PACE 2017 [21], the successor of PACE 2016. It should be noted that these test instances of PACE 2017 are much harder than those of PACE 2016: the winning implementation of PACE 2016 mentioned above, which solved 199 of the 200 instances therein, solves only 62 of these 100 instances of PACE 2017 in the given time of 30 minutes per instance.

Adapting the BT algorithm to work in PID mode has turned out non-trivial. It requires concepts and observations not present in [9]. We describe these concepts and observations, formulate our variant in full details, and prove its correctness.

We also perform an experimental analysis in which we count combinatorial structures involved in both PID and non-PID approaches, namely minimal separators, potential maximal cliques, and related objects. The analysis reveals that the practical bottleneck of the original BT algorithm lies in listing potential maximal cliques. Let $\mathcal{P}_k(G)$ denote the set of all potential maximal cliques of cardinality of $k+1$ or smaller of graph G. Although there are theoretical upper bounds of $O(1.7549^n)$ and $n^{O(1)}\binom{\lceil(2n+k+8)/3\rceil}{k+2}$ on the time to compute $\mathcal{P}_k(G)$ [12], where n is the number of vertices, huge gaps between these bounds and $|\mathcal{P}_k(G)|$ are observed in the experiments. This motivates the need of output sensitive algorithms that run fast when $|\mathcal{P}_k(G)|$ is small. Our PID algorithm is a first step in this direction. Although it does not compute $|\mathcal{P}_k(G)|$ in an output sensitive manner, it does compute the set of positive subproblem instances, whose size is empirically comparable to $|\mathcal{P}_k(G)|$, in an output sensitive manner.

Due to the space limitation, we omit proofs of lemmas and theorems, all of which can be found in the full paper. Our implementation in source code is available at our GitHub repository [13].

2 Preliminaries

In this paper, all graphs are simple, that is, without self loops or parallel edges. Let G be a graph. We denote by $V(G)$ the vertex set of G and by $E(G)$ the edge set of G. For each $v \in V(G)$, $N_G(v)$ denote the set of neighbors of v in G: $N_G(v) = \{u \in V(G) \mid \{u, v\} \in E(G)$. For $U \subseteq V(G)$, the *open neighbor set of* U *in* G, denoted by $N_G(U)$, is the set of vertices adjacent to some vertex in U but not belonging to U itself: $N_G(U) = (\bigcup_{v\in U} N_G(v)) \setminus U$. The *closed neighbor set of* U *in* G, denoted by $N_G[U]$, is defined by $N_G[U] = U \cup N_G(U)$. We also write $N_G[v]$ for $N_G[\{v\}] = N_G(v) \cup \{v\}$. We denote by $G[U]$ the subgraph of G induced by U: $V(G[U]) = U$ and $E(G[U]) = \{\{u, v\} \in E(G) \mid u, v \in U\}$. In the above notation, as well as in the notation further introduced below, we will often drop the subscript G when the graph is clear from the context.

We say that vertex set $C \subseteq V(G)$ is *connected in* G if, for every $u, v \in C$, there is a path in $G[C]$ between u and v. It is a *connected component* of G if it is connected and is inclusion-wise maximal subject to this condition. A vertex set C in G is a *component associated with* $S \subseteq G$, if C is a connected component of $G[V(G) \setminus S]$. For each $S \subseteq V(G)$, we denote by $\mathcal{C}_G(S)$ (or $\mathcal{C}(S)$ when G is clear from the context) the set of all components associated with S. A vertex set $S \subseteq V(G)$ is a *separator* of G if $|\mathcal{C}_G(S)| \geq 2$. A component C is a *full component* associated with separator S if $N(C) = S$. A separator S is a *minimal separator* if there are at least two full components associated with S. This term is justified by this fact: if S is a minimal separator and a, b vertices belonging to two distinct full components associated with S, then for every proper subset S' of S, a and b belong to the same component associated with S'; S is a minimal set of vertices that separates a from b.

Graph H is *chordal* if every induced cycle of H has length exactly three. H is a *minimal chordal completion of* G if it is chordal, $V(H) = V(G)$, $E(G) \subseteq E(H)$, and $E(H)$ is minimal subject to these conditions. A vertex set $\Omega \subseteq V(G)$ is a potential maximal clique of G, if Ω is a clique in some minimal chordal completion of G.

A *tree-decomposition* of G is a pair $(T, \mathcal{X})$ where T is a tree and $\mathcal{X}$ is a family $\{X_i\}_{i \in V(T)}$ of vertex sets of G such that the following three conditions are satisfied. We call members of $V(T)$ *nodes* of T and each X_i the *bag* at node i.

1. $\bigcup_{i \in V(T)} X_i = V(G)$.
2. For each edge $\{u, v\} \in E(G)$, there is some $i \in V(T)$ such that $u, v \in X_i$.
3. For each $v \in V(G)$, the set of nodes $I_v = \{i \in V(T) \mid v \in X_i\}$ of $V(T)$ induces a connected subtree of T.

The *width* of this tree-decomposition is $\max_{i \in V(T)} |X_i| - 1$. The *treewidth* of G, denoted by $\mathrm{tw}(G)$ is the minimum width of all tree-decompositions of G. We may assume that the bags X_i and X_j are distinct from each other for $i \neq j$ and, under this assumption, we will often regard a tree-decomposition as a tree T in which each node is a bag.

We call a tree-decomposition T of G *canonical* if each bag of T is a potential maximal clique of G and, for every pair X, Y of adjacent bags in T, $X \cap Y$ is a minimal separator of G. The following fact is well-known. It easily follows, for example, from Proposition 2.4 in [9].

▶ **Lemma 1.** *Let G be an arbitrary graph. There is a tree-decomposition T of G of width $\mathrm{tw}(G)$ that is canonical.*

The following local characterization of a potential maximal clique is crucial. We say that a vertex set $S \subseteq V(G)$ is *cliquish* in G if, for every pair of distinct vertices u and v in S, either u and v are adjacent to each other or there is some $C \in \mathcal{C}(S)$ such that $u, v \in N(C)$. In other words, S is cliquish if completing $N(C)$ for every $C \in \mathcal{C}(S)$ into a clique makes S a clique.

▶ **Lemma 2** (Theorem 3.15 in [9]). *A separator S of G is a potential maximal clique of G if and only if* (1) *S has no full-component associated with it and* (2) *S is cliquish.*

It is also shown in [9] that if Ω is a potential maximal clique of G and S is a minimal separator contained in Ω, then there is a unique component C_S associated with S that contains $\Omega \setminus S$. We need an explicit way of forming C_S from Ω and S.

Let $K \subseteq V(G)$ be an arbitrary vertex set and S an arbitrary proper subset of K. We say that a component $C \in \mathcal{C}(K)$ is *confined to* S if $N(C) \subseteq S$; otherwise it is *unconfined to* S. Let $\mathrm{unconf}(S, K)$ denote the set of components associated with K that are unconfined to S. Define $\mathrm{crib}(S, K) = (K \setminus S) \cup \bigcup_{C \in \mathrm{unconf}(S,K)} C$. The following lemma relies only on

the second property of potential maximal cliques, namely that they are cliquish, and will be applied not only to potential maximal cliques but also to separators with full components, which are trivially cliquish.

▶ **Lemma 3.** *Let $K \subseteq V(G)$ be a cliquish vertex set. Let S be an arbitrary proper subset of K. Then,* $\mathrm{crib}(S, K)$ *is a full component associated with S.*

▶ Remark. As $\mathrm{crib}(S, K)$ contains $K \setminus S$, it is clearly the only component associated with S that intersects K. Therefore, the above mentioned assertion on potential maximal cliques is a corollary of this Lemma.

3 Recurrences on oriented minimal separators

In this section, we fix graph G and positive integer k that are given in the problem instance: we are to decide if the treewidth of G is at most k.

For connected set $C \subseteq V(G)$, we denote by $G\langle C\rangle$ the graph obtained from $G[N[C]]$ by completing $N(C)$ into a clique: $V(G\langle C\rangle) = N[C]$ and $E(G\langle C\rangle) = E(G[N[C]]) \cup \{\{u, v\} \mid u, v \in N(C), u \neq v\}$. We say C is *feasible* if $\mathrm{tw}(G\langle C\rangle) \leq k$. Equivalently, C is feasible if $G[N[C]]$ has a tree-decomposition of width k or smaller that has a bag containing $N(C)$.

Let us first review the BT algorithm [9] adapting it to our decision problem. We first list all minimum separators of cardinality k or smaller and all potential maximal cliques of cardinality $k + 1$ or smaller. Then, for each pair of a potential maximal clique Ω and a minimal separator S such that $S \subset \Omega$, place a link from S to Ω. To understand the difficulty of formulating a PID variant of the algorithm, it is important to note that the pair (Ω, S) to be linked is easy to find from the side of Ω, but not the other way round. Then, we scan the full blocks $(N(C), C)$ of minimal separators in the increasing order of $|C|$ to decide if C is feasible, using the following recurrence: C is feasible if and only if there is some potential maximal clique Ω such that $N(C) \subset \Omega$, $C = \mathrm{crib}(N(C), \Omega)$, and every component $D \in \mathrm{unconf}(N(C), \Omega)$ is feasible. Finally, we have $\mathrm{tw}(G) \leq k$ if and only if there is a potential maximal clique Ω with $|\Omega| \leq k + 1$ such that every component associated with Ω is feasible.

To facilitate the PID construction, we orient minimal separators as follows. We assume a total order $<$ on $V(G)$. For each vertex set $U \subseteq V(G)$, the *minimum element* of U, denoted by $\min(U)$, is the smallest element of U under $<$. For vertex sets U and W, we say U *precedes* W and write $U \prec W$ if $\min(U) < \min(W)$.

We say that a connected set C is *inbound* if there is some full block associated with $N(C)$ that precedes C; otherwise, it is *outbound*. Observe that if C is inbound then $N(C)$ is a minimal separator, since $N(C)$ has another full component associated with it. Contrapositively, if $N(C)$ is not a minimal separator then C is necessarily outbound. We say a full block $(N(C), C)$ is *inbound* (*outbound*) if C is inbound (outbound, respectively).

▶ **Lemma 4.** *Let K be a cliquish vertex set and let A_1, A_2 be two components associated with K. Suppose that A_1 and A_2 are outbound. Then, either $N(A_1) \subseteq N(A_2)$ or $N(A_2) \subseteq N(A_1)$.*

Let K be a cliquish vertex set. Based on the above lemma, we define the *outlet* of K, denoted by $\mathrm{outlet}(K)$, as follows. If no non-full component associated with K is outbound, then we let $\mathrm{outlet}(K) = \emptyset$. Otherwise, $\mathrm{outlet}(K) = N(A)$, where A is a non-full component associated with K that is outbound, chosen so that $N(A)$ is maximal. We define $\mathrm{support}(K) = \mathrm{unconf}(\mathrm{outlet}(K), K)$, the set of components associated with K that are not confined to $\mathrm{outlet}(K)$. By Lemma 4, every member of $\mathrm{support}(K)$ is inbound.

We call a full block $(N(C), C)$ an *I-block* if C is inbound and $|N(C)| \leq k$. We call it an *O-block* if C is outbound and $|N(C)| \leq k$.

We say that an I-block $(N(C), C)$ is *feasible* if C is feasible. We say that an O-block $(N(A), A)$ is feasible if $N(A) = \bigcup_{C \in \mathcal{C}} N(C)$ for some set $\mathcal{C}$ of feasible inbound components. Note that this definition of feasibility of an O-block is somewhat weak in the sense that we do not require every inbound component associated with $N(A)$ to be feasible.

Let Ω be a potential maximal clique with $|\Omega| \leq k+1$. For each $C \in \text{support}(\Omega)$, block $(N(C), C)$ is an I-block, since C is inbound as observed above and we have $|N(C)| \leq k$ by our assumption that $|\Omega| \leq k+1$. We say that Ω is *feasible* if $|\Omega| \leq k+1$ and either

1. $\Omega = N[v]$ for some $v \in V(G)$,
2. there is some subset $\mathcal{C}$ of $\text{support}(\Omega)$ such that $\Omega = \bigcup_{D \in \mathcal{C}} N(D)$ and every member of $\mathcal{C}$ is feasible, or
3. $\Omega = N(A) \cup (N(v) \cap A)$ for some feasible O-block $(N(A), A)$ and a vertex $v \in N(A)$.

We say that Ω is *strongly feasible* if $|\Omega| \leq k+1$ and every $C \in \text{support}(\Omega)$ is feasible. It will turn out that every strongly feasible potential maximal clique is feasible (Lemma 9). This implication, however, is not immediate from the definitions.

▶ **Lemma 5.** *We have* $\text{tw}(G) \leq k$ *if and only if* G *has a strongly feasible potential maximal clique* Ω *with* $\text{outlet}(\Omega) = \emptyset$.

▶ **Lemma 6.** *Let* C *be a connected set of* G *such that* $N(C)$ *is a minimal separator. Let* Ω *be a potential maximal clique of* $G\langle C \rangle$*. Then,* Ω *is a potential maximal clique of* G.

The following is our oriented version of the recurrence in the BT algorithm described in the beginning of this section.

▶ **Lemma 7.** *An I-block* $(N(C), C)$ *is feasible if and only if there is some strongly feasible potential maximal clique* Ω *with* $\text{outlet}(\Omega) = N(C)$ *and* $\bigcup_{D \in \text{support}(\Omega)} D = C$.

▶ **Lemma 8.** *Let* K *be a cliquish vertex set,* $\mathcal{C}$ *a non-empty subset of* $\text{support}(K)$*, and* $S = \bigcup_{C \in \mathcal{C}} N(C)$*. If* S *is a proper subset of* K *then* $\text{crib}(S, K)$ *is outbound.*

The following lemma is crucial for our PID result: the algorithm described in the next section generates all feasible potential maximal cliques and we need to guarantee all strongly feasible maximal cliques to be among them.

▶ **Lemma 9.** *Let* Ω *be a strongly feasible potential maximal clique. Then,* Ω *is feasible.*

4 Algorithm

Given graph G and positive integer k, our algorithm generates all I-blocks, O-blocks, and potential maximal cliques that are feasible. In the following algorithm, variable $\mathcal{I}$ is used for listing feasible I-blocks, $\mathcal{O}$ for feasible O-blocks, $\mathcal{P}$ for feasible potential maximal cliques, and $\mathcal{S}$ for strongly feasible potential maximal cliques.

Algorithm PID-BT
Input Graph G and positive integer k
Output "YES" if $\text{tw}(G) \leq k$; "NO" otherwise
Procedure

1. Let $\mathcal{I}_0 = \emptyset$ and $\mathcal{O}_0 = \emptyset$.
2. Initialize $\mathcal{P}_0$ and $\mathcal{S}_0$ to $\emptyset$.
3. Set $j = 0$.

4. For each $v \in V(G)$, if $N[v]$ is a potential maximal clique with $|N[v]| \leq k+1$ then add $N[v]$ to $\mathcal{P}_0$ and if, moreover, $\mathrm{support}(N[v]) = \emptyset$ then do the following.
   a. Add $N[v]$ to $\mathcal{S}_0$.
   b. If $\mathrm{outlet}(N[v]) \neq \emptyset$ then let $C = \mathrm{crib}(\mathrm{outlet}(N[v]), N[v])$ and, provided that $C \neq C_h$ for $1 \leq h \leq j$, increment j and let $C_j = C$.
5. Set $i = 0$.
6. Repeat the following and stop repetition when j is not incremented during the iteration step.
   a. While $i < j$, do the following.
      i. Increment i and let $\mathcal{I}_i$ be $\mathcal{I}_{i-1} \cup \{C_i\}$.
      ii. Initialize $\mathcal{O}_i$ to $\mathcal{O}_{i-1}$, $\mathcal{P}_i$ to $\mathcal{P}_{i-1}$, and $\mathcal{S}_i$ to $\mathcal{S}_{i-1}$.
      iii. For each $B \in \mathcal{O}_{i-1}$ such that $C_i \subseteq B$ and $|N(C_i) \cup N(B)| \leq k+1$, let $K = N(C_i) \cup N(B)$ and do the following.
         A. If K is a potential maximal clique, then add K to $\mathcal{P}_i$.
         B. If $|K| \leq k$ and there is a full component A associated with K (which is unique), then add A to $\mathcal{O}_i$.
      iv. Let A be the full component associated with $N(C_i)$ and add A to $\mathcal{O}_i$.
      v. For each $A \in \mathcal{O}_i \setminus \mathcal{O}_{i-1}$ and $v \in N(A)$, let $K = N(A) \cup (n(v) \cap A)$ and if $|K| \leq k+1$ and K is a potential maximal clique then add K to $\mathcal{P}_i$.
      vi. For each $K \in \mathcal{P}_i \setminus \mathcal{S}_{i-1}$, if $\mathrm{support}(K) \subseteq \mathcal{I}_i$ then add K to $\mathcal{S}_i$ and do the following: if $\mathrm{outlet}(K) \neq \emptyset$ then let $C = \mathrm{crib}(\mathrm{outlet}(K), K)$ and, provided that $C \neq C_h$ for $1 \leq h \leq j$, increment j and let $C_j = C$.
7. If there is some $K \in \mathcal{S}_j$ such that $\mathrm{outlet}(K) = \emptyset$, then answer "YES"; otherwise, answer "NO".

▶ **Theorem 10.** *Algorithm PID-BT, given G and k, answers "YES" if and only if* $\mathrm{tw}(G) \leq k$.

5 Experimental analysis

To identify the practical bottleneck in the BT algorithm, we have performed some experiments. We are interested in the number of combinatorial objects involved in the treewidth computation: minimal separators, potential maximal cliques, and feasible objects used in our PID algorithm. In the case of minimal separators and potential maximal cliques, we count the total numbers of those as well as of those *relevant* in our decision problem: minimal separators with cardinality k or smaller and potential maximal cliques with cardinality $k+1$ or smaller.

Table 1 shows the results on some random instances, with k set to the treewidth of the graph: we are not interested in larger k and, for smaller k, the numbers in the columns dependent on k are smaller. The full paper contains results for more graphs with varying number of edges. The total number of minimal separators and that of potential maximal cliques grow much faster than the number of feasible objects in our algorithm, as the size of the graph grows. However, the growth in the numbers of relevant minimal separators and relevant potential maximal cliques is similar to the growth in the number of feasible objects. For example, the number of relevant potential maximal cliques grows only slightly faster than the number of feasible potential maximal cliques and is within 1.2 times the latter for the graph with 40 vertices.

Thus, scanning all relevant minimal separators and all relevant potential maximal cliques as in the original BT algorithm may not be an immediate disadvantage. The bottleneck lies rather in the time to list all relevant potential maximal cliques. Table 2 shows the number of

Table 1 The numbers of principal objects in treewidth computation.

			minimal separators		potential maximal cliques		feasible objects		
$\|V\|$	$\|E\|$	tw	all	$\leq$ tw	all	$\leq$ tw + 1	I-blocks	O-blocks	PMCs
20	60	8	191	48	796	96	46	108	93
30	90	11	2983	247	20154	682	228	708	618
40	120	14	164773	2356	1740644	10372	2080	8637	8577

Table 2 The number of objects involved in generating principal objects.

			$\leq$ tw + 1	vertex	feasible objects			pairs to be
$\|V\|$	$\|E\|$	tw	PMCs	representations	I-blocks	O-blocks	PMCs	examined
20	60	8	96	25263	46	108	93	206
30	90	11	682	3480559	228	708	618	1351
40	120	14	10372	167700496	2080	8637	8577	17906

additional combinatorial objects, called vertex representations, which needs to be generated in the algorithm in [12] in order to list all relevant potential maximal cliques.

The figures in the table suggests that a more output sensitive algorithm for listing relevant potential maximal cliques is desirable and that some method not relying on vertex representation is needed to achieve this goal.

In our PID approach, each feasible potential maximal clique, except in the base case, is generated from a combination of a feasible O-block and a feasible I-block. Each feasible O-block in turn is generated also from a combination of a feasible O-block and a feasible I-block. Let $\mathcal{I}$ be the set of feasible I-blocks and $\mathcal{O}$ the set of feasible O-blocks of the given graph. The crucial fact to our advantage is that most of the pairs in $\mathcal{I} \times \mathcal{O}$ are easily seen not to generate a new O-block or a potential maximal clique. The last column in Table 2 shows that the number of pairs in $\mathcal{I} \times \mathcal{O}$ that remain to be examined seriously is quite small. The data structure we called block sieves, described in the next section, is used to quickly filter out those simply rejectable pairs.

Algorithm PID-BT has a trivial output-sensitive upper bound of $n^{O(1)}|\mathcal{I}| \cdot |\mathcal{O}|$ on the time to generate necessary objects. A tighter analysis of our algorithm would be of great interest. It is also interesting to study if our approach can be applied to the problem of listing relevant potential maximal cliques.

6 Implementation

In this section, we sketch two important ingredients of our implementation. Although both are crucial in obtaining the result reported in Section 7, our work on this part is preliminary and improvements are the subject of future research.

6.1 Data structures

The crucial elementary operation in our algorithm is the following. We have a set $\mathcal{O}$ of feasible O-blocks obtained so far and, given a new feasible I-block $(N(C), C)$, need to find all members $(N(A), A)$ of $\mathcal{O}$ such that $C \subseteq A$ and $|N(C) \cup N(A)| \leq k+1$. As the experimental analysis in the previous section shows, there is only a few such A on average for the tested instances even though $\mathcal{O}$ is usually huge. To support an efficient query processing, we introduce an abstract data structure we call block sieve.

Let G be a graph and k a positive integer. A *block sieve* for graph G and width k is a data structure storing vertex sets of $V(G)$ which supports the following operations.

store(U) : store vertex set U in in the block sieve.

supersets(U) : return the list of entries W stored in the block sieve such that $U \subseteq W$ and $|N(U) \cup N(W)| \leq k+1$.

Data structures for superset query have been studied [20]. The second condition above on the retrieved sets, however, appears to make this data structure new. For each $U \subseteq V(G)$, we define the *margin* of U to be $k+1-|N(U)|$. Our implementation of block sieves described below exploits an upper bound on the margins of vertex sets stored in the sieve.

We first describe how such block sieves with upper bounds on margins are used in our algorithm. Let $\mathcal{O}$ be the current set of O-blocks. We use t block sieves $\mathcal{B}_1, \ldots, \mathcal{B}_t$, each $\mathcal{B}_i$ having a predetermined upper bound m_i on the margins of the sets stored. We have $0 < m_1 < m_2 < \ldots < m_t = k$. We set $m_0 = 0$ for notational ease below. In our implementation, we choose roughly $t = \log_2 k$ and $m_i = 2^i$ for $0 < i < t$. For each $(N(A), A)$ in $\mathcal{O}$, A is stored in $\mathcal{B}_i$ such that the margin $k+1-|N(A)|$ is m_i or smaller but larger than m_{i-1}. When we are given an I-block $(N(C), C)$ and are to list relevant blocks in $\mathcal{O}$, we query all of the t blocks with the operations supersets(C). These queries as a whole return the list of all vertex sets A such that $(N(A), A) \in \mathcal{O}$, $C \subseteq A$, and $|(N(A) \cup N(C))| \leq k+1$.

We implement a block sieve by a trie $\mathcal{T}$. The upper bound m on margin is not used in the construction of the sieve; it is used in the query time. In the following, we assume $V(G) = \{1, \ldots, n\}$ and, by an interval $[i, j]$, $1 \leq i \leq j \leq n$, we mean the set $\{v : i \leq v \leq j\}$ of vertices. Each non-leaf node p of $\mathcal{T}$ is labelled with a non-empty interval $[s_p, f_p]$, such that $s_r = 0$ for the root r, $s_p = f_q + 1$ if p is a child of q, and $f_p = n$ if p is a parent of a leaf. Each edge (p, q) which connects node p and a child q of p, is labelled with a subset $S_{(p,q)}$ of the interval $[s_p, f_p]$. Thus, for each node p, the union of the labels of the edges along the path from the root to p is a subset of the interval $[1, s_p - 1]$, or $[1, n]$ when p is a leaf, which we denote by S_p. The choice of interval $[s_p, f_p]$ for each node p is heuristic. It is chosen so that the number of descendants of p is not too large or too small. In our implementation, the interval size is adaptively chosen from 8, 16, 32, and 64.

Each leaf q of trie $\mathcal{T}$ represents a single set stored at this leaf, namely S_q as defined above. We denote by $S(\mathcal{T})$ the set of all sets stored in $\mathcal{T}$. Then, for each node p of $\mathcal{T}$, the set of sets stored under p is $\{U \in S(\mathcal{T}) \mid U \cap [1, p] = S_p\}$.

We now describe how a query is processed against this data structure. Suppose query U is given. The goal is to visit all leaves q such that $U \subseteq S_q$ and $|N(U) \cup N(S_q)| \leq k+1$. This is done by a depth-first traversal of the trie $\mathcal{T}$. When we visit node p, we have the invariant that $U \cap [1, f_p] \subseteq S_p$, since otherwise no leaf in the subtree rooted at p stores a superset of U. Therefore, we descend from p to a child p' of p only if this invariant is maintained. Moreover, we keep track of the quantity $i(p, U) = |N(U) \cap S_p|$ in order to make further pruning of search possible. For each leaf q below p such that $U \subseteq S_q$, we have $i(q, U) \geq i(p, U)$. Combining this with eauality $|N(U) \setminus N(S_q)| = |N(U) \cap S_q| = i(q, U)$, we have $|N(U) \cup N(S_q)| \geq |N(S_q)| + i(p, U)$. Since we know an upper bound m on the margin $k+1-|N(S_q)|$ of S_q, or lower bound $k+1-m$ on $|N(S_q)|$, we may prune the search under node p if $i(p, U) > m$, since this inequality implies $|N(U) \cup N(S_q)| > k+1$ for every leaf q under p. When we reach a leaf q, we test if $|N(U) \cup N(S_q)| \leq k+1$ indeed holds.

6.2 Safe separators

The notion of safe separators for tree width was introduced by Bodlaender and Koster [6]: a separator S of G is *safe* if completing S into a clique does not change the treewidth of G. If

we find a safe separator S then the problem of deciding tree width of G reduces to that of deciding the treewidth of $G\langle C\rangle$ for each component C associated with S. Preprocessing G into such independent subproblems is highly desirable whenever possible.

The above authors observed that a powerful sufficient condition for safeness can be formulated based on graph minors. A *labelled minor* of G is a graph obtained from G by zero or more applications of the following operations. (1) Edge contraction: choose an edge $\{u, v\}$, replace u and v by a single new vertex and let all neighbors of u and v be adjacent to this new vertex; name the new vertex as either u or v. (2) Vertex deletion: delete a vertex together with all incident edges. (3) Edge deletion.

▶ **Lemma 11** (Bodlaender and Koster [6]). *A separator S of G is safe if, for every component C associated with S, $G[V(G) \setminus C]$ contains clique S as a labelled minor.*

Call a separator *minor-safe* if it satisfies the sufficient condition for safeness stated in this lemma. Bodlaender and Koster [6] showed that if S is a minimal separator and is an almost clique (deleting some single vertex makes it a clique) then S is minor-safe and moreover that the set of all almost clique minimal separators can be found in $O(n^2m)$ time, where n is the number of vertices and m is the number of edges.

We aim at capturing as many minor-safe separators as possible, at the expense of theoretical running time bounds on the algorithm for finding them. Thus, in our approach, both the algorithm for generating candidate separators and the algorithm for deciding minor-safeness are heuristic. For candidate generation, we use greedy heuristic for treewidth such as min-fill and min-degree: the separators in the resulting tree-decomposition are all candidates for safe separators.

When we apply our heuristic decision algorithm for minor-safeness to candidate separator S, one of the following occurs.

1. The algorithm answers "YES". In this case, the required labelled clique minor has been found for every component associated S and hence S is minor-safe.
2. The algorithm answers "DON'T KNOW". In this case, the algorithm has failed to find a labelled clique minor for at least one component, and hence it is not known if S is minor-safe or not.
3. The algorithm aborts, after reaching the prescribed number of execution steps.

Our heuristic decision algorithm works in two phases. Let S be a separator, C a component associated with S, and $R = V(G) \setminus (S \cup C)$. In the first phase, we contract edges in R and obtain a graph B on vertex set $S \cup R'$, where each vertex of R' is a contraction of some vertex set of R and B has no edge between vertices in R'. For each pair u, v of distinct vertices in S, let $N(u, v)$ denote the common neighbors of u and v in graph B. The contractions are performed with the goal of making $|N(u, v) \cap R'|$ large for each missing edge $\{u, v\}$ in S. In the second phase, for each missing edge $\{u, v\}$, we choose a common neighbor $w \in N(u, v) \cap R'$ and contract either $\{u, w\}$ or $\{v, w\}$. The choice of the next missing edge to be processed and the choice of the common neighbor are done as follows. Suppose the contractions in the second phase are done for some missing edges in S. For each missing edge $\{u, v\}$ not yet "processed", let $N'(u, v)$ be the set of common neighbors of u and v that are not yet contracted with any vertex in S. We choose $\{u, v\}$ with the smallest $|N'(u, v) \cap R'|$ to be processed next. Tie-breaking when necessary as well as the choice of the common neighbor w in $N'(u, v) \cap R'$ to be contracted with u or v is done in such a way that the minimum of $|(N'(x, y) \cap R') \setminus \{w\}|$ is maximized over all remaining missing edges $\{x, y\}$ in S.

The performance of these heuristics strongly depends on the instances. For PACE 2017 public instances, they work quite well. Table 3 shows the preprocessing result on the last 10

Table 3 Safe separator preprocessing on PACE 2017 instances.

name	$\|V\|$	$\|E\|$	$tw(G)$	safe separators found	max subproblem	time(secs)
ex181	109	732	18	18	89	0.078
ex183	265	471	11	173	76	0.031
ex185	237	793	14	142	52	0.046
ex187	240	453	10	138	81	0.031
ex189	178	4517	70	6	161	0.062
ex191	492	1608	15	184	132	0.171
ex193	1391	3012	10	791	119	3.17
ex195	216	382	10	114	84	0.015
ex197	303	1158	15	176	56	0.062
ex199	310	537	9	157	131	0.046

of those instances. For each instance, the number of safe separators found and the maximum subproblem size in terms of the number of vertices, after the graph is decomposed by the safe separators found, are listed. The results show that these instances, which are deemed the hardest among all the 100 public instances, are quickly decomposed into manageable subproblems by our preprocessing.

On the other hand, these heuristics have turned out useless for most of the DIMACS graph coloring instances: no safe separators are found for those instances. We suspect that this is not the limitation of the heuristics but is simply because those instances lack minor-safe separators.

7 Performance results

We used our implementation of the PID-BT algorithm to determine the treewidth of benchmark instances. For a given instance, we use our decision procedure with k being incremented one by one, starting from the obvious lower bound, namely the minimum degree of the graph. Binary search is not used because the cost of overshooting the exact treewidth is huge.

The computing environment for the experiment is as follows. CPU: Intel Core i7-7700K, 4.20GHz; RAM: 32GB; Operating system: Windows 10, 64bit; Programming language: Java 1.8; JVM: jre1.8.0_121. The maximum heap size is 6GB by default and is 24GB where it is stated so. The implementation is single threaded, except that multiple threads may be invoked for garbage collection by JVM. The time measured is the CPU time, which includes the garbage collection time.

Table 4 lists the DIMACS graph coloring instances that are newly solved: the previously known upper and lower bounds did not match. For all but three of them, the previous best upper bound has turned out optimal: only the lower bound was weaker. In this experiment, however, no knowledge of previous bounds are used and our algorithm independently determines the exact treewidth.

The results on “queen" instances illustrate how far our algorithm has extended the practical limit of exact treewidth computation. Queen7_7 with 49 vertices is the largest instance previously solved, while queen10_10 with 100 vertices is now solved.

Our implementation also solves all previously solved DIMACS graph coloring instances within 10 seconds per instance and many of them within a second. Moreover, for many of the test instances which it leaves unsolved, it significantly improves the previously best known lower bounds. The details can be found in the full paper.

Table 5 summarizes the result on PACE 2017 public instances. More details can be found in the full paper. The instance which took the longest time (530 seconds) was “ex169” which

Table 4 Newly solved DIMACS graph coloring instances.

name	$\|V\|$	$\|E\|$	tw	time(secs)	prev UB	prev LB
DSJC125.5	**125**	**3891**	**108**	**459**	**108**	56
DSJC250.9	**250**	**27897**	**243**	**0.44**	**243**	212
DSJC500.9	**500**	**112437**	**492**	**14**	**492**	433
DSJR500.5	**500**	**58862**	**246**	**546**	-	**-**
games120†	**120**	**638**	**32**	**94738**	**32**	24
homer†	**561**	**1628**	**30**	**2765**	31	26
miles750	**128**	**2113**	**36**	**0.23**	**36**	35
myciel6	**95**	**755**	**35**	**419**	**35**	29
queen8_8	**64**	**728**	**45**	**4.16**	**45**	25
queen9_9	**81**	**1056**	**58**	**274**	**58**	35
queen8_12	**96**	**1368**	**65**	**649**	**-**	39
queen10_10	**100**	**1470**	**72**	**20934**	**72**	39

Previous upper bounds from [14] and [16]; previous lower bounds from [14] and [8].
† 24GB heap space is used for these instances.

Table 5 Summary of the results on PACE 2017 public instances.

	$t \leq 1$ sec	1 sec $< t \leq 1$ min	1 min $< t \leq 10$ min
the number of instances solved in time t	25	68	7

has 3706 vertices, 42236 edges, and treewidth 22. Considering the fact that this test set has been designed to be challenging for the second competition on treewidth in PACE and that the time allocated for each instance is 30 minutes, we can say that our implementation performs quite well.

Acknowledgment. The author thanks Hiromu Ohtsuka for his help in implementing the block sieve data structure. He also thanks Yasuaki Kobayashi for helpful discussions and especially for drawing the author's attention to the notion of safe separators. This work would have been non-existent if not motivated by the timely challenges of PACE 2016 and 2017. The author is deeply indebted to their organizers, especially Holger Dell, for their dedication and excellent work.

References

1 Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

2 Jeremias Berg and Matti Järvisalo. Sat-based approaches to treewidth computation: an evaluation. In *Tools with Artificial Intelligence (ICTAI), 2014 IEEE 26th International Conference on*, pages 328–335. IEEE, 2014.

3 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1996.

4 Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. A note on exact algorithms for vertex ordering problems on graphs. *Theory of Computing Systems*, 50(3):420–432, 2012.

5 Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On exact algorithms for treewidth. *ACM Transactions on Algorithms (TALG)*, 9(1):12, 2012.

6 Hans L. Bodlaender and Arie M. C. A. Koster. Safe separators for treewidth. *Discrete Mathematics*, 306(3):337–350, 2006.

7 Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2007.

8 Hans L. Bodlaender, Thomas Wolle, and Arie M. C. A. Koster. Contraction and treewidth lower bounds. *J. Graph Algorithms Appl.*, 10(1):5–49, 2006.

9 Vincent Bouchitté and Ioan Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM Journal on Computing*, 31(1):212–232, 2001.

10 Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The first parameterized algorithms and computational experiments challenge. In *LIPIcs – Leibniz International Proceedings in Informatics*, volume 63. Schloss Dagstuhl-Leibniz – Zentrum fürr Informatik, 2017.

11 Fedor V. Fomin, Dieter Kratsch, Ioan Todinca, and Yngve Villanger. Exact algorithms for treewidth and minimum fill-in. *SIAM Journal on Computing*, 38(3):1058–1079, 2008.

12 Fedor V. Fomin and Yngve Villanger. Treewidth computation and extremal combinatorics. *Combinatorica*, pages 1–20, 2012.

13 GitHub repository for TCS-Meiji on PACE2017 Track A submission. `https://github.com/TCS-Meiji/PACE2017-TrackA`. Public since: 2017-05-01.

14 Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 201–208. AUAI Press, 2004.

15 David S. Johnson and Michael A. Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

16 Nysret Musliu. An iterative heuristic algorithm for tree decomposition. In *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, pages 133–150. Springer, 2008.

17 Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

18 Neil Robertson and Paul D. Seymour. Graph minors. XX. wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2):325–357, 2004.

19 Marko Samer and Helmut Veith. Encoding treewidth into SAT. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 45–50. Springer, 2009.

20 Iztok Savnik. Index data structure for fast subset and superset queries. In *International Conference on Availability, Reliability, and Security*, pages 134–148. Springer, 2013.

21 The Parameterized Algorithms and Computational Experiments Challenge. `https://pacechallenge.wordpress.com`. Accessed: 2017-06-30.

Exponential Lower Bounds for History-Based Simplex Pivot Rules on Abstract Cubes*

Antonis Thomas

Department of Computer Science, Institute of Theoretical Computer Science, ETH Zürich, Zürich, Switzerland
athomas@inf.ethz.ch

Abstract

The behavior of the simplex algorithm is a widely studied subject. Specifically, the question of the existence of a polynomial pivot rule for the simplex algorithm is of major importance. Here, we give exponential lower bounds for three history-based pivot rules. Those rules decide their next step based on memory of the past steps. In particular, we study Zadeh's least entered rule, Johnson's least-recently basic rule and Cunningham's least-recently considered (or round-robin) rule. We give exponential lower bounds on Acyclic Unique Sink Orientations of the abstract cube, for all of these pivot rules. For Johnson's rule our bound is the first superpolynomial one in any context; for Zadeh's it is the first one for AUSO. Those two are our main results.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.1 Combinatorics

Keywords and phrases pivot rule, lower bound, exponential, unique sink orientation, zadeh

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.69

1 Introduction

The existence of a polynomial time pivot rule for the simplex algorithm is a major open problem in the theory of optimization. Most known rules have superpolynomial lower bounds by now. For deterministic rules, in particular, it is the case that many of them admit exponential lower bounds. Klee and Minty with their seminal paper [16], already in 1972, gave an exponential lower bound for Dantzig's original pivot rule. Their construction has been heavily studied ever since (for example [10],[3]) and inspired many later lower bounds.

In this paper, we are interested in a family of deterministic pivot rules known as history-based (or having memory). For those, superpolynomial lower bounds seemed to be elusive until recently. Arguably, the most famous history-based rule is due to Zadeh. Known as the *least entered* rule, it was described in 1980 with a technical report that was reprinted in 2009 [26]. This rule keeps a history of how many times each improving direction has been used and, at every step, chooses one that minimizes this history (a tie-breaking rule takes care of ties). The least entered rule was specifically designed to attack constructions similar to the Klee-Minty by using the improving directions in a balanced way (note that in this regard, it is similar to a random walk). With a letter to Klee in the 80s, Zadeh offered a $1000 prize to anyone who can prove polynomial upper or superpolynomial lower bounds for the least entered rule. This prize was claimed in 2011, by Friedmann [6], with a superpolynomial lower bound on actual Linear Programs (LP). No non-trivial upper bounds are known for this rule.

* A full version of the paper is available at https://arxiv.org/abs/1706.09380.

Another interesting rule was suggested by Cunningham [4], known as the *least-recently considered* rule. It fixes an initial ordering on all improving directions and then selects one in a round-robin fashion, starting from the last direction selected. The history here is to remember which was the last used improving direction. Furthermore, the *least-recently basic* rule, which Cunningham attributes to Johnson, was also first discussed in the same paper [4]. That rule selects the improving direction that left the basis least recently (in other words the direction whose opposite was selected least recently). For a detailed exposition on those and many other history-based pivot rules, the interested reader should look at Aoshima *et al.* [1].

We provide *exponential lower bounds*, by means of Acyclic Unique Sink Orientations, for all three aforementioned history-based rules.

Unique Sink Orientations. (USO) is an abstract framework that generalizes LP (and other problems). It was originally described by Stickney and Watson [23] and later revived by Szabó and Welzl [24]. Such abstract frameworks have received lots of attention since the discovery of the Random Facet pivot rule: Kalai [15] and, independently, Matoušek, Sharir and Welzl [19] proved subexponential upper bounds for this rule on LP. It became evident that their analysis made use only of combinatorial properties of LP and, thus, it was possible to extend their upper bounds in a much more abstract setting [8].

The most well-studied such framework is that of USO (e.g. [18],[5],[11] and see also below). Intuitively, a USO is an orientation of the hypercube graph such that every non-empty face has a unique sink (vertex with only incoming edges). The computational problem is to discover the unique global sink by performing vertex evaluations (each one reveals the orientation of the edges incident to the vertex). Commonly, acyclic USO (AUSO) constructions have served as lower bounds for pivot algorithms (e.g. [17], [22], [20], [14]) and our lower bounds are also manifested as AUSO.

Prior work and open questions. Aoshima *et al.* [1] explore the possibility that there exist AUSO on which history-based pivot rules take a Hamiltonian path. They prove, with the help of computers, that Zadeh's pivot rule admits such Hamiltonian paths up to dimension 9 at least. On the contrary, they show that Johnson's rule (among others) does not admit Hamiltonian paths and, so, they ask if it admits exponential paths on AUSO.

Recently, Avis and Friedmann [2] gave the first exponential lower bound for history-based rules. Namely, they prove an exponential lower bound for Cunningham's rule on binary parity games (definitions in [2]). Their constructions translate immediately to linear programs and also AUSO, for which[1] the lower bound is $\Omega(2^{n/5})$. However, they are very complicated and, thus, the authors ask if it is possible to prove exponential lower bounds for this rule, on AUSO, in a simpler manner.

Moreover, they compare their construction to the one for Zadeh's rule [6]. The latter gives a family of non-binary parity games (which correspond to linear programs), where Zadeh's rule takes a subexponential number of steps, of the form $2^{\Omega(\sqrt{n})}$ (where n is the number of variables of the LP). Although binary parity games correspond directly to AUSO, the same is not known for non-binary ones. Hence, Avis and Friedmann ask [2] if superpolynomial lower bounds for Zadeh's rule exist also on AUSO. In addition, Friedmann's lower bound [6] is based on a tie-breaking rule which is *artificial* in the sense that it always works in favor of

[1] The exact translation of binary Parity Games to AUSO is explained in [2]. Roughly, their constructed binary parity game translates to a cube of dimension $5n'$, where the path that the algorithm will take is of length $2^{n'}$. Thus, for n-dimensional AUSO, that is a lower bound of the form $\Omega(2^{n/5})$.

the lower bound designer. It is not described in the paper because, as the author writes, it is "not a natural one". Thus, he raises the question [6] of whether it is possible to obtain a lower bound with a *natural* tie-breaking rule.

Finally, Avis and Friedmann write [2]: "More generally it is of interest to determine whether all of the history based rules mentioned in [1] have exponential behaviour on AUSO".

Our results. With Theorem 4, we give an exponential lower bound for Johnson's rule. This is the *first superpolynomial* lower bound for this algorithm. Moreover, we give an exponential lower bound for Zadeh's rule, with Theorem 8. This has a number of advantages compared to the known construction: Firstly, it is exponential, whereas Friedmann's lower bounds [6] are subexponential (also *not* known to translate to AUSO). Secondly, our constructions are much simpler to describe. Finally, it is based on a tie-breaking rule that is essentially as *simple* as possible: a fixed ordered list. These two lower bounds constitute the *main results* of this paper. With Theorem 3, we give an exponential lower bound for Cunningham's rule. The advantage here is that the construction is significantly simpler; the lower bound also happens to be slightly improved. Theorem 3 serves as a warm-up to the main results by introducing the techniques and notation we use for our constructions.

Therefore, we answer to the positive all the questions described in the previous paragraph. Due to space constraints, many details and some proofs are missing from this extended abstract; a full version with all the details and a more complete analysis can be found at [25].

Our methods. The constructions in this paper are based on the building tools originally developed by Schurr and Szabó [21]; we do, however, introduce some novel ideas needed to deal with history-based pivot rules. Most known inductive lower bound constructions (e.g. [21],[22],[20],[14]) embed copies of the previous construction into the next one, in such a way that the algorithm gets trapped in the previous construction twice. For Zadeh's rule this does not work: it balances the directions being used and it inevitably escapes the second trap (at the next inductive step). To overcome this, we build a trap that consists of a small number of copies, being connected in a careful way which ensures that the algorithm uses the improving directions in a *balanced* fashion: it follows the path of the previous construction, up to making additional "balancing moves" between different copies.

Lower bounds on AUSO. It is not clear if AUSO lower bound constructions (including ours) can be realized as LP. However, the abstract setting allows for simpler proofs that are easy to communicate. We, thus, believe that such constructions are relevant for understanding the behavior of the pivot rules; the ideas could be used for the design of LP-based exponential lower bounds. For example, the first subexponential lower bounds for Random Facet [17] (tight to the upper bound; see also [9]) and for Random Edge [20] (at every step chooses one improving direction at random) were both proved by AUSO constructions. Indeed, for these two rules, subexponential lower bounds have been later proved on actual LP [7]. The most recent lower bound on AUSO was by Hansen and Zwick in 2016 [14], where they improve the subexponential lower bound for Random Edge. Note that for this rule non-trivial exponential upper bounds are known in the general case [13] and under assumptions [12].

2 Preliminaries

Let $[n] = \{1, \ldots, n\}$ and $\pm[n] = \{-n, \ldots, -1, 1, \ldots, n\}$. Let $Q^{[n]} = 2^{[n]}$ be the set of vertices of the n-dimensional hypercube over coordinates in $[n]$. Often we write Q^n (the

superscript indicates the dimension). A vertex of the hypercube $v \in Q^n$ is denoted by the set of coordinates it contains. Generally, with $C \subseteq [n]$ we denote a set of coordinates. Consider two vertices $v, u \in Q^n$. With $v \oplus u$ we denote the symmetric difference of the two sets. Now, let $C \subseteq 2^{[n]}$ and $v \in Q^n$. A *face* of the hypercube, $F(C, v)$, is defined as the set of vertices that are reached from v over the coordinates defined by any subset of C, i.e. $F(C, v) = \{u \in Q^n | v \oplus u \subseteq C\}$. The dimension of the face is $|C|$. We call edges the faces of dimension 1, e.g. $F(\{j\}, v)$. For $k \leq n$ we call a face of dimension k a k-face.

Let ψ denote an orientation of the edges of the hypercube Q^n. Consider two vertices $v, u \in Q^n$ and a coordinate $j \in [n]$. The notation $v \xrightarrow{j} u$ (w.r.t ψ) means that $F(\{j\}, v) = \{v, u\}$ and that the corresponding edge is oriented from v to u in ψ. Sometimes we write $v \to u$, when the coordinate is irrelevant. An edge $v \xrightarrow{j} u$ is forward if $j \in u$ and otherwise we say it is backward. We use $v \rightsquigarrow w$ to denote (that there is) a directed path from v to w.

We now define the concept of *direction*; the algorithms that we study here have memory of the directions that have been used so far. A direction is a signed coordinate. Let $c \in C$ be a coordinate; two different directions correspond to c, $+c$ and $-c$. At a vertex v the direction $+c$ corresponds to a forward edge incident to v and $-c$ to a backward edge. We say that a direction is *available* at vertex v if the corresponding edge is outgoing. Thus, at each vertex if a coordinate is incoming then none of the directions is available and if a coordinate is outgoing then exactly one of the directions is available. Similarly to above, we write $v \xrightarrow{d} u$, for some direction d. Note that if we have $v \xrightarrow{+c} v'$ (similarly $-c$) then at v' neither $+c$ nor $-c$ can be available. Generally, we denote with $D \subseteq \pm[n]$ a set of directions. Given a set of coordinates C, we say that D is the set of directions that corresponds to C to mean $D = \{-c, +c \mid c \in C\}$. Often, we use d to denote a direction without specifying its sign.

Then, ψ is a *Unique Sink Orientation* (USO) of Q^n when every non-empty face has a unique sink. USO can be either cyclic or acyclic (for these we write AUSO). n-AUSO means an AUSO over Q^n. For a USO ψ, we define s_ψ, the outmap function: for every $v \in Q^n$, $s_\psi(v) = \{j \in [n] | v \xrightarrow{j} (v \oplus \{j\})\}$, that is the set of coordinates on which v has an outgoing edge. A sink of a face $F(C, v)$ is a vertex $u \in F(C, v)$, such that $s_\psi(u) \cap C = \emptyset$. The whole cube is a face of itself; thus, there is a unique vertex v, the *global sink* with $s(v) = \emptyset$. In the rest, we write $s(v)$ to denote the outmap of v (ψ will be clear from the context).

The computational problem associated with a USO is to find the global sink. The computational model is the *vertex oracle* model. We have access to an oracle such that when we give it a vertex v, it replies with the outmap $s(v)$ of v. This is the standard computational model in USO literature and all the lower and upper bounds are with respect to it.

We are now ready to state the Product and Reorientation lemmas (due to [21]) which are the building tools for our constructions. The following constitutes an intuitive description of the Product lemma which is relevant to us: Consider an n-AUSO A (oriented hypercube graph) and take 2^m copies of A. For every vertex $v \in A$, take an m-AUSO A_v, the *connecting frame* for v. Each copy of A corresponds to a vertex of the frame A_v and v in this copy of A is connected according to that vertex of A_v. The result is an $(n + m)$-AUSO. Formally:

▶ **Lemma 1** (Product [21]). *Let C be a set of coordinates, $C' \subseteq C$ and $\bar{C'} = C \setminus C'$. Let $\tilde{s}$ be a USO outmap on $Q^{C'}$. For each vertex $u \in Q^{C'}$ we have a USO outmap s_u on $Q^{\bar{C'}}$. Then, the orientation defined by the outmap $s(v) = \tilde{s}(v \cap C') \cup s_{v \cap C'}(v \cap \bar{C'})$ on Q^C is a USO. Furthermore, if $\tilde{s}$ and all s_u are acyclic so is s.*

The Reorientation lemma, which follows, can be intuitively explained this way: if we have a USO and there is a face, such that all the vertices in this face have the same outmap on the edges external to the face, then we can reorient this face according to any other USO.

▶ **Lemma 2** (Reorientation [21]). *Let C be a set of coordinates, $C' \subseteq C$ and $\bar{C}' = C \setminus C'$. Let s be a USO on Q^C and let $\mathcal{F} = F(C', u)$, for some $u \in Q^C$, be a face of Q^C. If, for any two vertices $v, w \in \mathcal{F}$, $s(v) \cap \bar{C}' = s(w) \cap \bar{C}'$ and $\tilde{s}$ is a USO on $Q^{\bar{C}'}$, then the outmap $s'(v) = \tilde{s}(v \cap C') \cup (s(v) \cap \bar{C}')$ for $v \in \mathcal{F}$ and $s'(v) = s(v)$ otherwise is a USO on Q^C.*

3 A warm-up: Cunningham's Rule

▶ **Theorem 3.** *There exists n-AUSO such that Cunningham's rule, with a suitable starting vertex and list, takes a path of length at least $2^{n/4}$.*

We will sketch a proof for the above theorem. But let us start with some general comments and definitions that will apply to all our constructions. Firstly, they are inductive. Let A_i be the ith step of the induction. The base case is A_0. We call C_i a bundle of coordinates; that is the set of coordinates that was added at the ith step of induction (and C_0 are the coordinates of the base case). We also define $C_i^+ = \bigcup_{k=0}^{i} C_k$. Then, D_i denotes the set of directions that corresponds to C_i and similarly for D_i^+. Let v_0^i be the starting vertex for A_i. Consider that there is a token, which is initially on v_0^i, and at every step moves according to the direction that the given algorithm chooses. The path that the token takes from v_0^i to the unique sink, on A_i, is denoted with P_i; its length is denoted with $|P_i|$.

To construct A_{i+1} from A_i, we take 2^m copies of A_i and connect their vertices with m-dimensional connecting frames, for some constant m (Lemma 1). Afterwards, we perform one reorientation (Lemma 2), to install a simple balancing gadget. The token starts at the starting vertex v_0^{i+1} and walks on a path P (in A_{i+1}) until it reaches a vertex that has all coordinates from C_i^+ incoming. This vertex corresponds to the sink of A_i. If we project the path P to only the directions from D_i^+ we get exactly P_i. In the balancing gadget the token will be taken back to the vertex that corresponds to the starting vertex for A_i. The idea is to prove that if we project the rest of the token's path to the global sink, to only the directions from D_i^+, we get again P_i. Thus, $|P_{i+1}| > 2|P_i|$. Let $T(n)$ denote the length of the corresponding paths on an n-AUSO. The recursion we get then is $T(n+m) > 2T(n)$. This gives rise to exponential lower bounds of the form $2^{n/m}$. For Cunningham's and Johnson's rules the constant is $m = 4$ and for Zadeh's $m = 6$.

The lower bound. Consider that the algorithm runs on an n-AUSO. It has an ordered list L that contains all $2n$ directions; let $L[k]$ indicate the kth direction on the list. There is a marker μ of which direction was used last: if direction $L[k]$ was used at the last step then $\mu = k$. At the next step the algorithm will start checking the directions on the list from $L[\mu+1]$ in a cyclic order (so if it reaches $L[2n]$ it continues from $L[1]$) and it chooses the first available one. Initially, $\mu = 2n$ so that the first direction that the algorithm checks is $L[1]$.

We will now give a short sketch of the lower bound. Full details can be found in the full version [25]. Let A_0 be the base case and $L_0 = (+c_0^1, -c_0^2, +c_0^3, -c_0^1, +c_0^4, -c_0^3, +c_0^2, -c_0^4)$. To construct A_{i+1} from A_i we take 2^4 copies of A_i which we connect with three different frames. The two crucial ones F_1 and F_2 are given in Figure 1; F_3 can be found in [25]. The new set of coordinates will be $C_{i+1} = \{c_{i+1}^1, c_{i+1}^2, c_{i+1}^3, c_{i+1}^4\}$.

Let us define some notation in reference to Figure 1. An AUSO is given as a collection of 2-faces on the first two coordinates. All coordinates are labeled. Each square represents a face on coordinates C_i^+. All of these faces, except $\boxed{\text{B}}$, are internally oriented according to A_i (correspond to copies). The numbers are indicating in which order the token will visit them. We refer to these faces in the text; for example, we write $\boxed{1}$ to mean the face: $F(C_i^+, \{c_{i+1}^2\})$. Given a vertex v, we write $v \perp \boxed{1}$ to mean the vertex $v' \in \boxed{1}$, such that $v \cap C_i^+ = v' \cap C_i^+$.

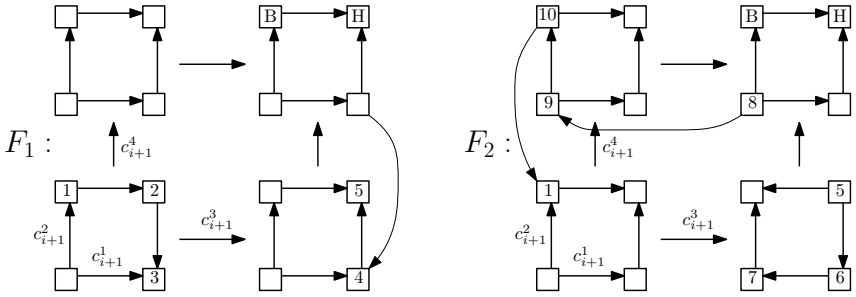


Figure 1 The orientations F_1 and F_2, used as connecting frames, are given in this figure. The 4-dimensional frames are split in 2-faces of coordinates c^1_{i+1} and c^2_{i+1}. The arrows on the other coordinates indicate the orientation of all the edges on this coordinate except when noted differently. For example, in F_2 all edges on coordinate c^3_{i+1} are oriented from left to right except the edge $\boxed{9} \leftarrow \boxed{8}$. Each $\boxed{\cdot}$ represents a face on C^+_i. This notation is valid also for the next figures.

Moreover, we write $\boxed{1} \rightsquigarrow \boxed{5}$ to mean a path from a vertex in $\boxed{1}$ to the corresponding vertex in $\boxed{5}$, using only directions from D_{i+1}. In this case, the exact vertex will be clear from the context. The face $\boxed{\text{B}}$ is the one that contains the balancing gadget, which is installed by use of Lemma 2. In this construction and the one of Section 4, $\boxed{\text{H}}$ is a hypersink (has all edges external to the face incoming). In the construction of Section 5 there is no hypersink.

Let us give a short description of the behavior of Cunningham's rule on A_{i+1}. Firstly, the starting vertex $v_0 = v_0^{i+1} = \{c^2_0, \ldots, c^2_{i+1}\}$. Thus, the token is initially placed in $\boxed{1}$. The list $L_{i+1} = L_i \cdot (+1, -2, +3, -1, +4, -3, +2, -4)$, where $\cdot$ represents concatenation. For simplicity, we write a number $\pm k$ instead of $\pm c^k_{i+1}$. So, the directions from bundle D_k have priority over the ones from bundle $D_{k'}$, if $k < k'$. Let $IN = D^+_i$ and $OUT = D_{i+1}$; that is the set of directions from the previous inductive steps and the current one respectively.

The connecting frame is F_1. The algorithm uses directions from IN and the token moves in $\boxed{1}$. When these are exhausted the token takes a path $\boxed{1} \rightsquigarrow \boxed{5}$. The directions from OUT are, then, exhausted. The algorithm uses a direction from IN, in $\boxed{5}$. At the next vertex the frame changes[2] to F_2. When the directions from OUT will be used again (after the ones from IN) the token will take a path $\boxed{5} \rightsquigarrow \boxed{10} \rightarrow \boxed{1}$. After one step in $\boxed{1}$, on a direction from IN, the frame changes to F_1. The conclusion is that the token will keep moving between $\boxed{1}$ and $\boxed{5}$ until it reaches a vertex v_{s_i}, such that $s(v_{s_i}) \cap C^+_i = \emptyset$. This is not a cycle as the token keeps moving toward v_{s_i} on the IN directions (this was once P_i).

When the token is on s_i the frame will change to F_3. The token will walk to $\boxed{\text{B}}$ and the OUT directions will get exhausted exactly there. Inside $\boxed{\text{B}}$, the algorithm will be forced to take a path back to vertex $v_0 \perp \boxed{\text{B}}$ (the sink of $\boxed{\text{B}}$). The next time the directions from OUT will be used the token will go in the hypersink $\boxed{\text{H}}$. But there, it will be at vertex $v_0 \perp \boxed{\text{H}}$, the coordinates from C_{i+1} will be incoming from there on and, so, it will stay inside $\boxed{\text{H}}$ where it will perform P_i once again. We can conclude that $|P_{i+1}| > 2|P_i|$ can be proved for this construction, which also proves Theorem 3. Details in the full version [25].

[2] At this point the change of frame can be understood with an adversary argument. We play against the algorithm and we choose which frame to reveal at which vertex. This is consistent with Lemma 1.

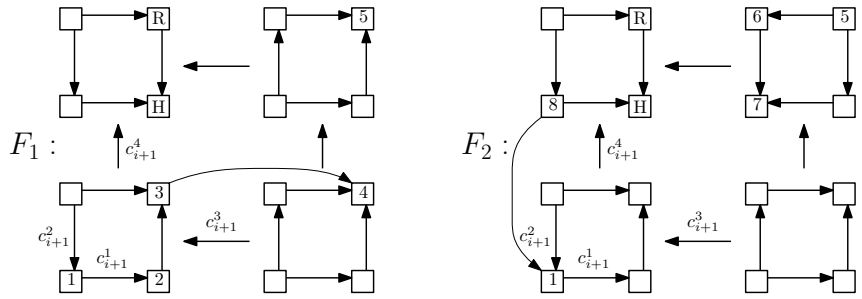


Figure 2 The orientations F_1 and F_2, used as connecting frames, are given in this figure.

4 Exponential lower bound for Johnson's rule

▶ **Theorem 4.** *There exists n-AUSO such that Johnson's rule, with a suitable starting vertex, takes a path of length at least $2^{n/4}$.*

In this section we will prove the above theorem. Let us define Johnson's least-recently basic rule. Consider that the algorithm runs on an n-AUSO. It maintains a history function h which is defined on all $2n$ directions. Let v be the current vertex. Intuitively, the algorithm keeps the following history: Say direction d was used at step x and let $-d$ be the opposite of that direction. Then, at step x we have $h(-d) = x$; this will stay intact until $-d$ is used. On the other hand, $h(d)$ will keep increasing to the current step until $-d$ is used. Formally, for a direction d, $h(d)$ is the last step number when $|d| \in v$ if d is positive and the last step number when $|d| \notin v$ if d is negative. Here, $|d|$ denotes the coordinate that corresponds to direction d. Ties are possible and we assume that they are broken lexicographically. The algorithm chooses from the set of available directions, direction d which minimizes $h(d)$.

The construction. The construction is inductive. Let A_i denote the ith inductive step. The base case A_0 is the 4-dimensional AUSO F_1, shown in Figure 2. The initial set of coordinates is $C_0 = \{c^1_0, c^2_0, c^3_0, c^4_0\}$. The starting vertex is $v_0 = \emptyset$, which is at the vertex labeled $\boxed{1}$ in the figure. Then, the algorithm will go over directions $(+c^1_0, +c^2_0, +c^3_0, +c^4_0, -c^3_0, -c^2_0)$ and will find the sink at $\{c^1_0, c^4_0\}$. Let us now describe how to construct AUSO A_{i+1} from AUSO A_i. Every inductive step adds 4 dimensions. As before, $C^+_j = \bigcup_{k=0}^{j} C_k$. Let the new bundle of coordinates be $C_{i+1} = \{c^1_{i+1}, c^2_{i+1}, c^3_{i+1}, c^4_{i+1}\}$. We take 16 copies of A_i and connect them with the 4-AUSO F_1 and F_2 that appear in Figure 2. In this section, a reset-AUSO (thus, the $\boxed{\text{R}}$ in the figure) will take the role of the balancing gadget.

Let us define what we mean by lexicographic order here: $+c^k_j$ comes before $-c^{k'}_j$ for any k and k'; $+c^k_j$ comes before $+c^{k'}_j$ and $-c^k_j$ comes before $-c^{k'}_j$ if $k < k'$. Finally, d^k_j comes before $d^{k'}_{j'}$ for any k and k' and positive/negative sign, if $j < j'$.

The starting vertex will be $v_0 = \emptyset$ for every inductive step. Then, every positive direction $+c$ initially has $h(+c) = 0$ (until $+c$ is used by the algorithm). At step number 1, one of the positive directions will be used at which point every negative direction $-c$ has $h(-c) = 1$.

With this construction we want to force the algorithm to the following behavior: It starts at $\boxed{1}$ using all the positive directions in lexicographic order. Then it is in $\boxed{5}$, where it will use all the negative directions in lexicographic order. This will continue until the sink of A_i has been reached. It follows that directions from D^+_i are only used when the algorithm is in $\boxed{1}$ or $\boxed{5}$, before the sink of A_i has been discovered. We will later show that this is the case.

Then, we consider the construction as an adversary argument. Firstly, the starting vertex of A_i and the sink of A_i both use F_1 as the connecting frame. Every time the algorithm is in

$\boxed{1}$ and uses a direction from D_i^+, we change (or keep) the connecting frame to F_1. Similarly, when the algorithm arrives in $\boxed{5}$ and uses a direction from D_i^+, we change the connecting frame to F_2. This operation is consistent with Lemma 1: Every vertex is connected with the corresponding frame. The result of this operation is A'_{i+1} which is not the final AUSO.

The final step for the construction of A_{i+1} is to use Lemma 2 to embed a reset-AUSO to the face $\boxed{\mathrm{R}}$. For every $i > 0$, R_i is a $4i$-AUSO (whereas A_i is a $4(i+1)$-AUSO). For the construction of A_{i+1} for A_i we embed R_{i+1} to the face $\boxed{\mathrm{R}}$. R_{i+1} is designed such that it has its sink at vertex $\emptyset$. In addition, it has a path from vertex $\{c_0^1, c_0^4, \ldots, c_i^1, c_i^4\}$ to the vertex $\emptyset$ such that every vertex on this path has only one outgoing edge and the path goes through the negative directions in lexicographic order: $(-c_0^1, -c_0^4, \ldots, -c_i^4, -c_i^4)$.

This reorientation concludes the construction of A_{i+1}. It does not introduce any cycles in A_{i+1}. A formal description of the reset-AUSO and an argument on the acyclicity of A_i can be found in the full version [25]. Note that in $\boxed{\mathrm{H}}$ we still have A_i.

The behavior of Johnson's rule. The behavior of Johnson's rule on the AUSO constructed as above will be described here. We give as much detail as space allows; the rest can be found in the full version [25]. Firstly, we define the tools that we are going to use for this analysis.

Similarly to the previous section, consider a token t. That is a token that starts at the initial vertex v_0 and moves according to the directions that the algorithm chooses. With slight abuse of notation we also use t to refer to the vertex where the token currently lies on. Moreover, we write t_j to mean the set $t \cap C_j^+$; that is, the projection of the vertex t to the set of coordinates C_j^+. Since $t_j \subseteq t$, we call t_j a subtoken.

We say that a coordinate bundle C_j is *active* when $s(t) \cap C_j \neq \emptyset$. Otherwise, we say that C_j is inactive. Note that for both the 4-dimensional frames that we have used, the sink is at the same vertex. This implies that C_j is inactive if and only if token t is such that $t \cap C_j = \{c_j^1, c_j^4\}$. Moreover, we say that token t is in $\boxed{1}_j$ to mean that $t \cap C_j = \emptyset$; t is in $\boxed{5}_j$ when $t \cap C_j = C_j$ and similarly for the rest of the faces $\boxed{\cdot}$ from Figure 2.

For each subtoken t_j, we say that it has reached *its sink* when all bundles in C_j^+ are inactive. This means that $t_j = \{c_0^1, c_0^4, \ldots, c_j^1, c_j^4\}$. *Resetting* C_j^+ is a process that happens when subtoken t_j is at its sink: It takes token t from a vertex where $t \cap C_j^+ = \{c_0^1, c_0^4, \ldots, c_j^1, c_j^4\}$ to a vertex where $t \cap C_j^+ = \emptyset$. Moreover, we say that C_j^+ is *resettable* when:

- t_j is at its sink.
- $h(-c_{j'}^1) < h(-c_{j'}^4) < h(-c_{j+1}^2)$, for every $0 \leq j' \leq j$.

The first bullet in the definition above equivalently means that all bundles in C_j^+ are inactive. Resetting C_j^+ is a process that takes place when (and only when) token t is in the reset-AUSO in $\boxed{\mathrm{R}}_{j+1}$. For now assume that C_j^+ will be reset only when it is resettable; we will prove this later (with Lemma 7). Thus, t_j is on its sink when the resetting process starts. The second bullet of the definition of resettable ensures that token t will not go out of $\boxed{\mathrm{R}}_{j+1}$ before C_j^+ has been reset. During the reset of C_j^+, token t will go over negative directions from D_j^+ in this order: $(-c_0^1, -c_0^4, \ldots, -c_j^1, -c_j^4)$. This is because of the construction of the reset-AUSO R_{j+1}; the algorithm has no other choice. We are ready to state the following lemma (a formal proof can be found in the full version [25]).

▶ **Lemma 5.** *Let $t \cap C_j^+ = \emptyset$. Then, all the positive directions from C_j^+ will be used in the lexicographic order: $(+c_0^1, +c_0^2, +c_0^3, +c_0^4, \ldots, +c_j^1, +c_j^2, +c_j^3, +c_j^4)$.*

The above lemma defines the path that token t will follow until subtoken t_j reaches its sink. For every bundle C_j, the positive directions are used in lexicographic order

$(+c_j^1, +c_j^2, +c_j^3, +c_j^4)$ and the token t goes to $\boxed{5}_j$. After this, we have $h(-c_j^1) < h(-c_j^2) < h(-c_j^3) < h(-c_j^4) < h(d)$ for any positive d from D_j. Then, some directions from C_{j-1}^+ will be used and the frame for C_j will change to F_2. When it is the turn of the negative directions from D_j to be used they will be used consecutively and in lexicographic order $(-c_j^1, -c_j^2, -c_j^3, -c_j^4)$; that is, assuming t_{j-1} has not reached its sink yet. Otherwise, the connecting frame for C_j would be F_1 and the directions $-c_j^1$ and $-c_j^4$ would not be available.

After this, t will be in $\boxed{1}_j$ and, moreover, $h(+c_j^1) < h(+c_j^2) < h(+c_j^3) < h(+c_j^4) < h(d)$ for any negative d from D_j. There, after some directions from C_{j-1}^+ are used, the connecting frame for C_j will be F_1. When it is the turn of the positive directions from D_j to be used they will be used consecutively and in in lexicographic order $(+c_j^1, +c_j^2, +c_j^3, +c_j^4)$ and the token t will go back to $\boxed{5}_j$. We can conclude that t will keep moving from $\boxed{1}_j$ to $\boxed{5}_j$ and reversely until subtoken t_{j-1} reaches its sink. The next corollary follows from this discussion.

▶ **Corollary 6.** *Let C_{j+1} be active.*

1. *If t is in $\boxed{1}_{j+1}$, then the positive directions from D_{j+1} will be used (when it is their turn) consecutively and in lexicographic order $(+c_{j+1}^1, +c_{j+1}^2, +c_{j+1}^3, +c_{j+1}^4)$.*
2. *If t is in $\boxed{5}_{j+1}$ and subtoken t_j has not reached its sink, then the negative directions from D_{j+1} will, similarly, be used (when it is their turn) consecutively as $(-c_{j+1}^1, -c_{j+1}^2, -c_{j+1}^3, -c_{j+1}^4)$.*

It follows that directions from D_j^+ are only used when t is in $\boxed{1}_{j+1}$ or in $\boxed{5}_{j+1}$.

The next lemma is the last ingredient needed (proof can be found in the full version [25]).

▶ **Lemma 7.** *Let C_{j+1} be active. When t_j reaches its sink, C_j^+ is resettable.*

Now consider the path of the token t from v_0 to the sink of A_{i+1}. This AUSO is of dimension $n = 4(i+2)$. By Corollary 6, t will be moving back and forth between $\boxed{1}$ and $\boxed{5}$ until t_i reaches its sink. When that happens, C_i^+ is resettable (by Lemma 7) and, when t enters $\boxed{\mathrm{R}}$, C_i^+ will be reset. Following, t enters $\boxed{\mathrm{H}}$ at vertex $v_0 \perp \boxed{\mathrm{H}}$.

When the algorithm started at $v_0 = \emptyset$, all the positive directions from D_{i+1}^+ where used in lexicographic order. Since it is deterministic, this defines completely the behavior of t. Consider the path P that t will follow from v_0 until t_i reaches its sink, but projected on the coordinates from C_i^+. From Lemma 5, we know that when token t is such that $t \cap C_i^+ = \emptyset$, all the positive directions from D_i^+ will be used in the lexicographic order. At this point C_{i+1} is inactive and t is in $\boxed{\mathrm{H}}$. Therein, it will follow the same path P to the global sink.

Let $T(n)$ denote the length of the path that token t will take from v_0 until it reaches the global sink on a n-AUSO. With the above analysis, we have shown that the recursion $T(n+4) > 2T(n)$ holds. This recursion leads to the proof of Theorem 4.

5 Exponential lower bound for Zadeh's Rule

▶ **Theorem 8.** *There exists n-AUSO such that Zadeh's rule, with a suitable starting vertex and tie-breaking rule, takes a path of length at least $2^{n/6}$.*

In this section we will prove the above theorem. Firstly, let us define formally Zadeh's least entered rule. Consider that the algorithm runs on an n-AUSO. It maintains a history function h which is defined on all $2n$ directions. Given a direction d, $h(d)$ is the number of times the direction d has been used. At the beginning $h(d) = 0$, for all d. At every step the algorithm picks one direction from the set of available ones that minimizes the history function. In addition, there is a tie-breaking rule: this is an ordering of the directions and is

invoked only in case more than one have the minimum history size. As we already mentioned in Section 1, our lower bound construction will have the *simplest* possible tie-breaking rule, an ordered list which will be given explicitly. This is in contrast to the lower bounds from [6].

Secondly, let us define some tools that we will use for the analysis of the algorithm. We have a *balance* function b, which is also defined on all the $2n$ directions. Let d_{max} be the most used direction; then, $b(d) = h(d_{max}) - h(d)$. This means that direction d has been used $b(d)$ less times compared to d_{max}. We say that a direction d is *imbalanced* when $b(d) > 0$ and that D is balanced when $b(d) = 0$, for all $d \in D$. We also define a balance function on any subset of directions: Given set D we define $b(D, d)$ to be the balance of direction d w.r.t. the directions from D, i.e. the defining coordinate is now $d_{max} \in D$.

Furthermore, we define the concept of *saturation*. This is with regards to history and the current vertex in the algorithm run. Given a set of directions $D \subseteq [\pm n]$ and a vertex v we say that v is D-saturated when for every $d \in D$ with $b(d) > 0$, the direction d is not available for v. It follows that if at vertex v set D is balanced, then v is D-saturated.

The construction. The construction is inductive. Let A_i denote the ith inductive step. The base case, A_0, is a 6-AUSO. Due to the lack of space we do not define it here, but we will mention and utilize some of its properties. A complete description of the base case can be found in the full version [25]. Every inductive step adds 6 new dimensions. As before the bundle of coordinates C_i is the one added at the ith step of induction (and C_0 are the ones of the base case). Also, with D_i we denote the directions that correspond to C_i and, similarly, for D_i^+. Thus, the AUSO A_i is $6(i+1)$-dimensional and the coordinates that describe it are in the set C_i^+. For each A_i, the starting vertex is $v_0^i = \{c_0^2, \ldots, c_i^2\}$.

Let us now describe how to construct AUSO A_{i+1} from AUSO A_i. Let the new bundle of coordinates be C_{i+1}. We call IN the set of directions D_i^+ and OUT the set of directions D_{i+1}. At every inductive step the tie-breaking rule will be formed such that the directions from IN have priority over the ones from OUT. Thus, directions D_k have priority over the ones from $D_{k'}$, if $k < k'$. For simplicity, we write number $\pm k$ to mean direction $\pm c_{i+1}^k$.

The starting vertex for A_{i+1} is $v_0 = v_0^{i+1} = \{c_0^2, \ldots, c_{i+1}^2\}$. Assume that P_i (the path the token takes in A_i) is known to us. Similarly to the previous sections, this can be interpreted as an adversary argument. To construct A_{i+1} we take $2^6 = 64$ copies of A_i and use three different 6-AUSO as connecting frames, utilizing Lemma 1. The crucial frames are given in Figure 3. For vertices that are not on P_i it does not matter which frame we use. For vertices that are on the path P_i we choose the connecting frame according to the following rule:

(1) Vertices that are not D_i^+-saturated (w.r.t. P_i) we connect with F_1.

(2) Vertices that are D_i^+-saturated (w.r.t. P_i) we connect with F_2.

(3) For the sink s_i of A_i we use F_3.

The latter is a 6-AUSO that has the same path $\boxed{1} \rightsquigarrow \boxed{12}$ as F_1, has its sink in $\boxed{12}$ (so the uppermost edge on coordinate c_{i+1}^1 is backward) and all other edges are forward (figure in [25]).

The result of this operation is A'_{i+1}. It remains to perform one reorientation. Namely, the balance-AUSO will be embedded in the face $\boxed{\text{B}}$, shown in Figure 3. The balance-AUSO is a uniform $6(i+1)$-AUSO which has its sink at the vertex v_0^i (all edges are oriented towards v_0^i). Formally, the outmap of vertex $v = v_0 \perp \boxed{\text{B}}$ is such that $s(v) \cap C_i^+ = \emptyset$. This reorientation will not introduce cycles; a formal proof can be found in the full version [25].

In reference to Figure 3, let us present the intuitive idea: The token will walk (in a projected way) along P_i once while walking between $\boxed{1}$ and $\boxed{12}$. Then, it will go back to

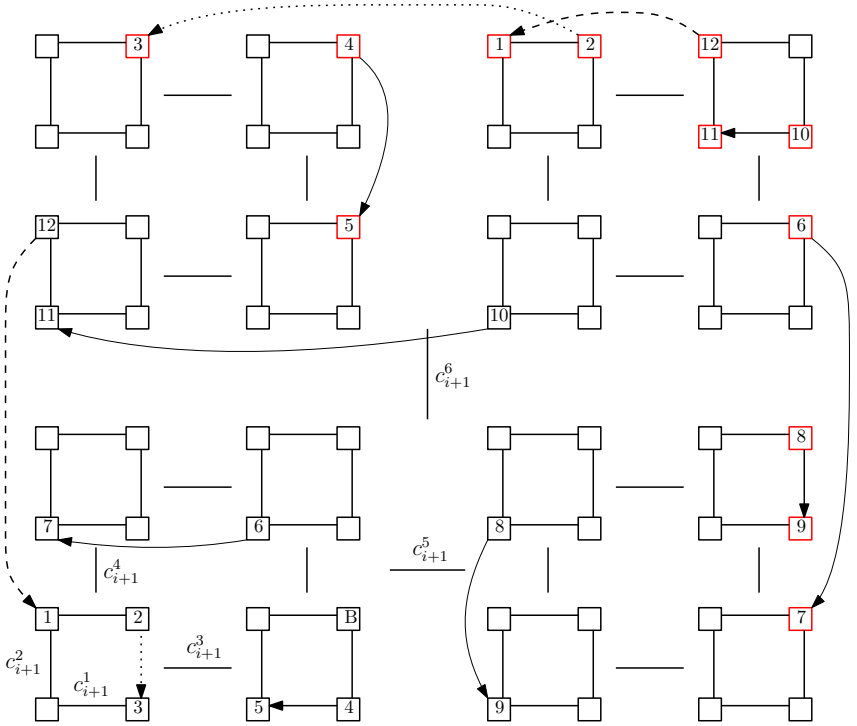


Figure 3 Both orientations F_1 and F_2 are given in this figure. For simplicity, only the orientations of the backward edges are explicitly drawn; every other edge is forward. The frame F_1 includes the *dashed* backward edges but not the dotted ones; F_2 includes the *dotted* backward edges but not the dashed ones. The solid backward edges are included in both frames.

the start of P_i in the balance-AUSO B. Then, it will walk the path P_i once again while walking between 1 and 12 .

Below, we give two crucial properties that will hold for our construction. The first one is about the base case A_0 (of which a detailed description can be found in the full version [25]).

(i) There is at least one $(\pm[6])$-saturated vertex, other than the starting vertex v^0_0, in A_0. In addition, the sink s_0 is at least two vertices away from the last vertex on the path P_0 that was $(\pm[6])$-saturated.

Property (i) will be utilized in the proof of Lemma 9. The second property holds for every inductive step A_k of the construction $0 \leq k \leq i+1$.

(ii) When the token reaches the sink s_k of A_k there are exactly $4(k+1)$ negative coordinates imbalanced. Let $IM_k = \{-c^3_0, -c^4_0, -c^5_0, -c^6_0, \ldots, -c^3_k, -c^4_k, -c^5_k, -c^6_k\}$. For every $d \in IM_k$ we have $b(d) = 1$ and for every other d we have $b(d) = 0$.

Property (ii) holds for the base case (details in the full version [25]). Then, we will argue in the step-by-step analysis that it also holds for every inductive step of the construction. Also note that if the token takes the directions in IM_k from the sink s_k, it will go to the the starting vertex v^k_0. Such a path is not available in any of the connecting frames; however, a path which spans exactly those directions is available in the balance-AUSO.

We will now analyze the behavior of Zadeh's rule on AUSO A_{i+1}. Firstly, let us define the tie-breaking ordered list T_{i+1}:

$$T_{i+1} = T_i \cdot (+1, -2, +3, -1, +4, -3, +5, -4, +6, -5, +2, -6).$$

As before a number $\pm k$ indicates the direction $\pm c^k_0$. Secondly, we state two lemmas that will be used in the analysis that comes below. We include proofs for those in the full version [25].

▶ **Lemma 9.** *Let token t be at an IN-saturated vertex v, such that $s(v) \cap C_i^+ \neq \emptyset$. Then, $\exists d \in IN$ such that $v \xrightarrow{d} v'$ and v' also has $s(v') \cap C_i^+ \neq \emptyset$. Moreover, v' is not IN-saturated.*

▶ **Lemma 10.** *Let the token t be at a vertex v as in the lemma above. Then there is a vertex v' that comes after v on P_{i+1} and such that v' is IN-saturated.*

Step-by-step analysis. We are ready to give a description for the behavior of the algorithm on A_{i+1}, in as much detail as the space allows; a very careful analysis can be found in the full version [25]. Initially, the token is at the starting vertex v_0. Let us denote with d_{max}^{OUT} the direction that maximizes history over the OUT directions; similarly, we define d_{max}^{IN}. Assume that the token is at an IN-saturated vertex and $b(d_{max}^{OUT}) > 0$. Then, the algorithm will use directions from OUT until it reaches a vertex that is OUT-saturated.

When at $\boxed{1}$, directions from IN will be used, since they have priority in T_{i+1}. After some steps, an IN-saturated vertex v_s will be reached, by Lemma 10. At v_s, we have that $b(d_{max}^{OUT}) > 0$. The connecting frame will be F_2. The directions from OUT will be utilized and the token will take a path $\boxed{1} \rightsquigarrow \boxed{12}$, where it will reach $v_s \perp \boxed{12}$. Then, we have $b(-6) = 1$ and for every other direction $d \in OUT$, $b(d) = 0$; also, $b(d_{max}^{OUT}) = 0$. Because the frame is F_2, the dashed edge is not available: $v_s \perp \boxed{12} \leftarrow v_s \perp \boxed{1}$. Thus, $v_s \perp \boxed{12}$ is OUT-saturated. One direction from IN will be used; by Lemma 9 the next vertex is not IN-saturated. The frame will then be F_1, and direction -6 will be used. The token is back to $\boxed{1}$ and $b(OUT, d) = 0$, for every $d \in OUT$, $b(d_{max}^{OUT}) = 1$ and $b(d_{max}^{IN}) = 0$. The token will keep moving between $\boxed{1}$ and $\boxed{12}$ in the way thus described, until it reaches a vertex v_{s_i}, such that $s(v_{s_i}) \cap C_i^+ = \emptyset$ (v_{s_i} corresponds to the sink of A_i). The latter will be evaluated in $\boxed{1}$, by Lemma 9. Then, we have that $b(d) = 1$, for every $d \in IM_i$, by Property (ii) (which holds inductively).

The frame for v_{s_i} is F_3. The token will go over $+1$ and $+3$ to $\boxed{\text{B}}$. Therein, it will take a path $v_{s_i} \perp \boxed{\text{B}} \rightsquigarrow v_0 \perp \boxed{\text{B}}$ for which it will use exactly the directions from IM_i; afterwards, IN is balanced. The token will take a path $v_0 \perp \boxed{\text{B}} \rightsquigarrow v_0 \perp \boxed{8} \rightsquigarrow v_0 \perp \boxed{12}$. All the vertices on this path are IN-saturated because IN is balanced. The frame will be F_2 and, so, the dashed edge is not available: $v_0 \perp \boxed{12} \leftarrow v_0 \perp \boxed{1}$. At this point $b(-3) = b(-4) = b(-5) = b(-6) = 1$ and for every other $d \in OUT$, $b(d) = 0$. From now on, all the steps that the token will take using directions from IN will be consistent with the path P_i. This is because IN is balanced and the token is at a vertex that corresponds to v_0^i (the starting vertex of A_i).

The token will keep moving between $\boxed{1}$ and $\boxed{12}$, in a similar way as described above, until it reaches a vertex v_{s_i} such that $s(v_{s_i}) \cap C_i^+ = \emptyset$. The latter will be evaluated in $\boxed{1}$, by Lemma 9. The main difference to the situation of the previous paragraphs is in the balance vector. After the token reaches an IN-saturated vertex in $\boxed{1}$, the balance vector will be $b(OUT, -4) = b(OUT, -5) = b(OUT, -6) = 1$ and $b(d_{max}^{OUT}) = 1$. This is also the case when v_{s_i} is reached. But then, the connecting frame is F_3 and a path $v_{s_i} \perp \boxed{1} \rightsquigarrow v_{s_i} \perp \boxed{12}$ will be taken. The global sink will be exactly at $v_{s_i} \perp \boxed{12}$. After the sink has been reached, we have $b(d_{max}^{OUT}) = 0$ and, thus, $b(-3) = b(-4) = b(-5) = b(-6) = 1$. Thus, Property (ii) will also be satisfied for the new inductive step.

With the above analysis, we have proved that the path P_{i+1} will have length that is larger than twice the length of path P_i. Therefore, we obtain the recursion $T(n+6) > 2T(n)$ which leads to the proof of Theorem 8.

6 Conclusions

In this paper, we have constructed AUSO on which the three pivot rules of interest can take exponentially long paths. Several interesting problems remain open: First and foremost

is settling if Zadeh's and Johnson's rules admit exponential lower bounds even on linear programs. Moreover, it remains open to decide if Zadeh's rule admits Hamiltonian paths on AUSO, a direction suggested by the authors of [1]. Finally, we are interested in exponential lower bounds for all the history-based rules that are discussed in [1]. We believe that our methods can be used to prove exponential lower bounds on AUSO for all of those rules.

References

1 Yoshikazu Aoshima, David Avis, Theresa Deering, Yoshitake Matsumoto, and Sonoko Moriyama. On the existence of Hamiltonian paths for history based pivot rules on acyclic unique sink orientations of hypercubes. *Discrete Applied Mathematics*, 160(15):2104–2115, 2012. doi:10.1016/j.dam.2012.05.023.

2 David Avis and Oliver Friedmann. An exponential lower bound for cunningham's rule. *Mathematical Programming*, pages 1–35, 2016. doi:10.1007/s10107-016-1008-4.

3 József Balogh and Robin Pemantle. The Klee-Minty random edge chain moves with linear speed. *Random Structures & Algorithms*, 30(4):464–483, 2007. doi:10.1002/rsa.20127.

4 William H Cunningham. Theoretical properties of the network simplex method. *Mathematics of Operations Research*, 4(2):196–208, 1979. doi:10.1287/moor.4.2.196.

5 Jan Foniok, Bernd Gärtner, Lorenz Klaus, and Markus Sprecher. Counting unique-sink orientations. *Discrete Applied Mathematics*, 163, Part 2:155–164, 2014. doi:10.1016/j.dam.2013.07.017.

6 Oliver Friedmann. A subexponential lower bound for Zadeh's pivoting rule for solving linear programs and games. In *IPCO 2011*, pages 192–206, 2011. doi:10.1007/978-3-642-20807-2_16.

7 Oliver Friedmann, Thomas Dueholm Hansen, and Uri Zwick. Subexponential lower bounds for randomized pivoting rules for the simplex algorithm. In *STOC 2011*, pages 283–292, 2011. doi:10.1145/1993636.1993675.

8 Bernd Gärtner. A subexponential algorithm for abstract optimization problems. *SIAM J. Comput.*, 24(5):1018–1035, 1995. doi:10.1137/S0097539793250287.

9 Bernd Gärtner. The random-facet simplex algorithm on combinatorial cubes. *Random Structures & Algorithms*, 20(3):353–381, 2002. doi:10.1002/rsa.10034.

10 Bernd Gärtner, Martin Henk, and Günter M. Ziegler. Randomized simplex algorithms on Klee-Minty cubes. *Combinatorica*, 18(3):349–372, 1998. doi:10.1007/PL00009827.

11 Bernd Gärtner and Antonis Thomas. The complexity of recognizing unique sink orientations. In *STACS 2015*, pages 341–353, 2015. doi:10.4230/LIPIcs.STACS.2015.341.

12 Bernd Gärtner and Antonis Thomas. The niceness of unique sink orientations. In *APPROX/RANDOM 2016*, pages 30:1–30:14, 2016. doi:10.4230/LIPIcs.APPROX-RANDOM.2016.30.

13 Thomas Dueholm Hansen, Mike Paterson, and Uri Zwick. Improved upper bounds for random-edge and random-jump on abstract cubes. In *SODA 2014*, pages 874–881, 2014. doi:10.1137/1.9781611973402.65.

14 Thomas Dueholm Hansen and Uri Zwick. Random-edge is slower than random-facet on abstract cubes. In *ICALP 2016*, pages 51:1–51:14, 2016. doi:10.4230/LIPIcs.ICALP.2016.51.

15 Gil Kalai. A subexponential randomized simplex algorithm (extended abstract). In *STOC 1992*, pages 475–482, 1992. doi:10.1145/129712.129759.

16 Victor Klee and George J. Minty. How good is the simplex algorithm? *Inequalities III*, pages 159–175, 1972.

17 Jiří Matoušek. Lower bounds for a subexponential optimization algorithm. *Random Structures & Algorithms*, 5(4):591–607, 1994. doi:10.1002/rsa.3240050408.

18 Jiří Matoušek. The number of unique-sink orientations of the hypercube*. *Combinatorica*, 26(1):91–99, 2006. doi:10.1007/s00493-006-0007-0.

19 Jiří Matoušek, Micha Sharir, and Emo Welzl. A subexponential bound for linear programming. *Algorithmica*, 16(4/5):498–516, 1996. doi:10.1007/BF01940877.

20 Jiří Matoušek and Tibor Szabó. RANDOM EDGE can be exponential on abstract cubes. *Advances in Mathematics*, 204(1):262–277, 2006. doi:10.1016/j.aim.2005.05.021.

21 Ingo Schurr and Tibor Szabó. Finding the sink takes some time: An almost quadratic lower bound for finding the sink of unique sink oriented cubes. *Discrete & Computational Geometry*, 31(4):627–642, 2004. doi:10.1007/s00454-003-0813-8.

22 Ingo Schurr and Tibor Szabó. Jumping doesn't help in abstract cubes. In *IPCO 2005*, pages 225–235, 2005. doi:10.1007/11496915_17.

23 Alan Stickney and Layne Watson. Digraph models of Bard-type algorithms for the linear complementarity problem. *Math. Oper. Res.*, 3(4):322–333, 1978. doi:10.1287/moor.3.4.322.

24 Tibor Szabó and Emo Welzl. Unique sink orientations of cubes. In *FOCS 2001*, pages 547–555, 2001. doi:10.1109/SFCS.2001.959931.

25 Antonis Thomas. Exponential lower bounds for history-based simplex pivot rules on abstract cubes. *CoRR*, abs/1706.09380, 2017. URL: https://arxiv.org/abs/1706.09380.

26 Norman Zadeh. What is the worst case behavior of the simplex algorithm. *Polyhedral computation*, 48:131–143, 2009.

Maxent-Stress Optimization of 3D Biomolecular Models*†

Michael Wegner[1], Oskar Taubert[2], Alexander Schug[3], and Henning Meyerhenke[4]

1 **Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
2 **Department of Physics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
3 **Steinbuch Centre for Computing, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
4 **Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**

Abstract

Knowing a biomolecule's structure is inherently linked to and a prerequisite for any detailed understanding of its function. Significant effort has gone into developing technologies for structural characterization. These technologies do not directly provide 3D structures; instead they typically yield noisy and erroneous distance information between specific entities such as atoms or residues, which have to be translated into consistent 3D models.

Here we present an approach for this translation process based on maxent-stress optimization. Our new approach extends the original graph drawing method for the new application's specifics by introducing additional constraints and confidence values as well as algorithmic components. Extensive experiments demonstrate that our approach infers structural models (*i. e.,* sensible 3D coordinates for the molecule's atoms) that correspond well to the distance information, can handle noisy and error-prone data, and is considerably faster than established tools. Our results promise to allow domain scientists nearly-interactive structural modeling based on distance constraints.

1998 ACM Subject Classification G.2.2 Graph Theory, G.1.6 Optimization

Keywords and phrases Distance geometry, protein structure determination, 3D graph drawing, maxent-stress optimization

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.70

1 Introduction

Context. Proteins are biomolecular machines that fulfill a large variety of tasks in living systems, be it as reaction catalysts, molecular sensors, immune responses, or driving muscular activity [40]. Knowing a protein's 3D structure is a requirement for any detailed understanding of its function, and functional or structural disorder can lead to disease. Structure resolution techniques have made strong progress in recent years: biomolecules that were inaccessible a decade ago can now be structurally resolved, as exemplified by the rapid growth of structural

* The full version of this paper is available at `https://arxiv.org/abs/1706.06805`.
† The work by MW and HM was partially supported by grant ME 3619/3-1 within German Research Foundation (DFG) Priority Programme 1736. AS acknowledges support by the *Helmholtz Impuls- und Vernetzungsfonds* and a Google Research Award. HM and AS acknowledge support by KIT's Young Investigator Network YIN.

databases [31]. The resolution techniques, however, do not directly provide structural information as 3D coordinates. Instead, *e. g.,* X-ray crystallography yields a diffraction pattern which has to be translated into a structural model. Similarly, Nuclear Magnetic Resonance (NMR) techniques measure coupled atomic spins, which can be translated into pairwise distances between specific atoms. These distances are typically incomplete, *i. e.,* not all spatially adjacent atom pairs are detected [37]. Also computational tools, such as co-evolutionary analysis of multiple-sequences alignments of protein [35, 39, 29] or RNA families [11] can provide distance constraints between residues for biomolecular structure prediction. Conceptually, one can understand the information provided by these techniques as incomplete and erroneous parts of the complete distance matrix.

Motivation. Our goal is to provide an efficient and effective method to compute a full structural 3D model from incomplete and/or noisy distances. For this task, physics-based approaches are computationally prohibitive. Molecular dynamics-based approaches require weeks on supercomputers [17] and stochastic global optimization techniques [33] still days on medium-sized clusters. To lower computational costs, one can use more coarse-grained force-fields [30], yet the computations still require hours to days depending on the input size.

For interactive or nearly-interactive work, however, running times of a few seconds would be desirable. This would support a quick back-and-forth between, *e. g.,* assigning NMR chemical shifts to determine pairwise atomic distances and follow-up structural modeling [37]. These structural models allow then, in turn, improved NMR shift assignment and a repetition of the loop. One forgoes describing the detailed physics and finds a near-optimal solution which respects all distance constraints. Exemplary tools which solve this distance geometry problem are DGSOL [26] and DISCO [23]. Challenges for such tools are efficiency, quality, and support for variations such as the ability to deal with noise, error, or distance intervals.

Outline and Contribution. In this paper we transfer a maxent-stress optimization approach from 2D graph drawing [16] to computing a 3D model from (incomplete) distance information. To this end, we exploit the resemblance of the objective functions (see Section 2), and extend the basic model and algorithm shown in Section 2.3 by specifics of the biological application. Our algorithmic adaptations and extensions, as well as details regarding their implementation, are presented in Section 3. Extensive experiments (see Section 4) reveal that our algorithm is significantly faster than other competitive algorithms; at the same time its solution quality is very often better than the results of the best competitor. In particular in our most realistic instances, we outperform our competitors (i) in terms of quality by providing more accurate structural models in (ii) consistently high agreement with reference models and requiring (iii) only about 5–10% of the running time. This stays true even when provided with noisy input data. We further extend our algorithm to support a weighted problem variant that allows to specify how certain a distance interval is and obtain very promising experimental results for this setting as well. To our knowledge, our algorithm is the first to support this new variant.

2 Preliminaries

2.1 Problem Definition

We model a biomolecule as a graph $G = (V, E)$, where the set V of n vertices models the atoms and the set E of m edges their relations. Distance information is given separately for all pairs $\{u, v\} \in S \subseteq V \times V$ by a distance matrix $D \in \mathbb{R}^{n \times n}_{\geq 0}$. For this purpose d_{vw} denotes

the distance between vertices v and w – or is set to `nil` if the distance is unknown (for pairs $\notin S$). We are interested in finding an embedding of G into $\mathbb{R}^3$, *i. e.,* 3D coordinates for the vertices, that respects the distances for S. This problem is known as (3D) *distance geometry problem* (DGP). In line with previous work, we account for inexact distance information due to measurement errors by introducing an interval in which the actual distance is contained. This modified DGP is called *interval distance geometry problem* [20]:

▶ **Definition 1** (Interval Distance Geometry Problem (iDGP))**.** Let a simple undirected graph $G = (V, E)$, a distance interval function $d = [l, u]$ with $d : E \to \mathbb{R}^2$, and an integer $k > 0$ be given. Then determine whether there is an embedding $x : V \to \mathbb{R}^k$ such that

$$\forall \{v, w\} \in E : l_{vw} \leq \|x_v - x_w\| \leq u_{vw}, \tag{1}$$

where l_{vw} and u_{vw} are lower and upper bounds for the distance of the edge $\{v, w\}$.

Here and in the following, k equals 3. Then DGP is prefixed by an 'M' for 'molecular'. Note that the (M)DGP is contained in the i(M)DGP by setting the lower and upper bound of each interval to be equal to the actual distance. Saxe [32] showed that deciding whether a valid embedding exists (in the DGP sense) is strongly NP-complete for $k = 1$ and strongly NP-hard for $k > 1$. Interestingly, the problem becomes solvable in polynomial time if all distances are given [6, 9, 12]. Since solving the decision problem (finding a valid embedding) is difficult and even not always possible, we continue by considering the embedding task as an optimization problem, to be solved by heuristics. As a measure of error, one could use the *largest distance mean error* (LDME) defined as: $\text{LDME}(x) = \sqrt{\frac{1}{|E|} \sum_{\{v,w\} \in E} \max(l_{vw} - \|x_v - x_w\|, \|x_v - x_w\| - u_{vw}, 0)^2}$. An embedding x that has an LDME value of 0 is obviously a solution of the iDGP as each distance constraint is met. One could thus minimize the LDME of the embedding found. To be closer to the biophysical application and real-world data, we actually use the *root mean square deviation* (RMSD) to *evaluate* our solutions. The RMSD compares the embedding against a reference structure:

$$\text{RMSD}(x, x') = \min \sqrt{\frac{1}{|V|} \sum_{v \in V} \|x_v - x'_v\|^2}, \tag{2}$$

with x_v and x'_v being the coordinates of the embedding and the reference structure, respectively. The minimum value is over all possible spatial translations and rotations of both superimposed structures. RMSD values < 1.5 Å approach the structure resolution limit of experimental wet-lab techniques (NMR and X-ray) [40]. While error functions that test the capability of the algorithm to match the constraints (such as LDME) can be useful from an optimization point of view, a good value does not necessarily mean that the embedding reproduces the molecular structure. In particular, the algorithm must be able to handle noisy and erroneous data. In the end, the structure must also be physically meaningful. The RMSD addresses this challenge and is a standard measure of (dis)similarity in structural molecular biology by directly assessing the usefulness in a real-life scenario[40]. We will therefore rely on the RMSD to compare algorithms.

2.2 Related Work

There exist many algorithms for solving the MDGP optimization problem. Yet, for most algorithms the required running time is either very high or the solution quality is in the meantime rather low. Also, some algorithms currently do not support iMDGP – which limits

their use in a real-world scenario. Due to space constraints the description of related work focuses on two tools, DGSOL and DISCO, which we use in our experimental comparison as they are publicly available and established in the distance geometry research community – cf. a book on distance geometry edited by Mucherino *et al.* [27] and an even more recent survey by the same authors [20]. For a much broader overview we refer to this book and survey. Mentioned running times (in seconds) are based on experimental data in previous works and are thus not necessarily completely comparable.

Liberti *et al.* [20] found in their survey from 2014 that general-purpose global optimization solvers like Octave's *fsolve* or spatial branch-and-bound techniques are not able to solve the problem for more than 10 vertices in a reasonable amount of time. One reason is that the objective function has a large number of local minima.

Moré and Wu [25] implemented an algorithm called DGSOL[1] that transforms the objective function gradually into a smoother function that approximates the original function and has fewer local minima. The algorithm builds a hierarchy of increasingly smooth functions by iteratively applying a transformation. In a next step, DGSOL employs Newton's method as a local optimization on the smoothest function and then traces this solution back to the original objective function, applying a local optimization on each level. Liberti *et al.* state that the algorithm "has several advantages: it is efficient, effective for small-to-medium-sized instances, and, more importantly, can be naturally extended to solve iMDGP instances" [20, p. 23]. On the downside, on large-scale instances the solution quality decreases (while the running time remains reasonable). An et al. [2, 1] use a different continuation approach which improves the solution quality compared to DGSOL. In their experiments the running time of their algorithm for proteins larger than 1 500 atoms lies between 500 and 1 200s. The double variable neighborhood search with smoothing (DVS) algorithm by Liberti et al. [19] combines the ideas of DGSOL and variable neighborhood search into one algorithm. In their comparison to DGSOL the quality of DVS was significantly better, but the running time two orders of magnitude slower already for small inputs.

Biswas *et al.* [5] proposed the DAFGL algorithm that decomposes the graph into clusters by running the symmetric reverse Cuthill-McKee algorithm on the distance matrix. The subproblems are solved with a semidefinite programming (SDP) formulation. DAFGL is capable of solving the iMDGP. While its solution quality are mostly reasonable, one has to consider that as much as 70% of distances smaller than 6 Å were provided with added noise. Also, larger instances increase the running time rapidly. The two largest molecules (PDB: 1toa and 1mqq, see Table 1) took already 2 654s and 1 683s with only modest to poor solution quality (RMSD: 3.2 Å and 9.8 Å) to solve even though the algorithm makes use of a distributed SDP solver. Leung and Toh [18] proposed the DISCO[2] algorithm that is an advancement to the DAFGL algorithm by Biswas *et al.* [5]. If the problem is small enough, DISCO solves the problem with an SDP approach and refines the obtained solution with regularized gradient descent. Otherwise, DISCO splits the graph into two subgraphs and solves the problem recursively. DISCO uses the symmetric reverse Cuthill-McKee algorithm to cluster the vertices initially. In a second step DISCO tries to minimize the edge cut between different subgraphs by placing a vertex v into the subgraph where most of its neighbors are placed. The algorithm puts some vertices in both subgraphs (overlapping atoms) to later stitch the two embedded subgraphs together. Its authors tested DISCO also in the iMDGP setting (*i. e.,* DISCO supports inexact distances). The results indicate that DISCO is able to

[1] Publicly available at `http://www.mcs.anl.gov/~more/dgsol/` (accessed on April 4, 2017).
[2] Publicly available at `http://www.math.nus.edu.sg/~mattohkc/disco.html` (accessed on April 4, 2017).

compute the structure of proteins with very sparse distance data and high noise in 412s for a protein having 3 672 atoms. Fang and Toh [14] presented some enhancements to DISCO for the iMDGP setting by incorporating knowledge about molecule conformations to improve the robustness of DISCO. Their experiments show that their changes indeed lead to better solutions (about 50–70%) with the cost of increased running times (also about 50–70%).

2.3 MaxEnt-Stress Optimization

We aim at developing an algorithm for iMDGP (and its extension wiMDGP, see Section 3.2) with a significantly lower running time than previous algorithms and with solutions of good quality. Our main idea is to use an objective function proposed by Gansner *et al.* [16] for planar graph drawing, called maxent-stress (short for *maximal entropy stress*). As the name suggests, it is composed of two parts, a stress part that penalizes deviations from the prescribed distances (with a quadratic penalty, possibly weighted) and an entropy part that penalizes vertices for getting too close to each other (atoms cannot overlap):

$$\min_x \sum_{vw \in S} \omega_{vw}(\|x_v - x_w\| - d_{vw})^2 \; - \alpha H(x), \tag{3}$$

where $H(x) = -\operatorname{sgn}(q) \sum_{vw \notin S} \|x_v - x_w\|^{-q}, q > -2$, is the entropy term, ω_{vw} a weighting factor for edge $\{v, w\}$, $\alpha \geq 0$ a user-defined control parameter, and $\operatorname{sgn}(q)$ the signum function with the special case that $sgn(0) = 1$.

To minimize function (3), Gansner *et al.* [16] derive a solution from successively solving Laplacian linear systems of the form $Lx = b$ for x. A noteworthy feature of this successive iteration towards a local minimum is that the solution of the current iteration depends on the solution of the previous iteration, since the current right-hand side is computed from the solution in the previous iteration. Note that the computation of distances between vertex pairs not in S is not required for function (3). Instead, vertex pairs not in S are related to each other via the entropy term, which enters the right-hand side, too. If the parameter q is set to be smaller than zero, the entropy term of vertex u acts as a sum of attractive forces on u. Conversely, the term acts as a sum of repulsive forces if q is larger than zero.

The Gansner *et al.* algorithm typically needs several iterations to converge. In this process the entropy weighting factor α has a strong influence in the maxent-stress model: a high value will cause the vertices to expand into space indefinitely while a low value will cause no entropy influence. The maxent-stress algorithm therefore starts with $\alpha = 1$ and gradually reduces this value to $\alpha = 0.008$ with a rate of 0.3. For each value of α, a maximum of 50 linear system solves are performed and Gansner *et al.* set q to 0 except when the graph has more than 30% degree-1 vertices (then $q \leftarrow 0.8$). Note that in this entropy context they assume $\|x\|^0 = \ln \|x\|$. If the relative difference $\|x' - x\|/\|x\|$ between two successive solutions x and x' is below 0.001, the algorithm is terminated.

More implementation details (*e. g.,* the approximation of the entropy term in case of $|S| \in \mathcal{O}(n)$) can be found in Section 3.3.

3 New Algorithm and its Implementation

Now we describe the adaptations made to the generic maxent-stress algorithm in order to deal with iMDGP and an extension called wiMDGP. For more technical details we refer the interested reader to our code[3].

[3] `https://algohub.iti.kit.edu/parco/NetworKit/NetworKit-MaxentStress`, main source folder `networkit/cpp/viz`. An updated standalone version is planned to be published in the future.

3.1 Adapting the Maxent-stress Algorithm for iMDGP

Recall that for iMDGP we are given a graph $G = (V, E)$ and the distance intervals $d = [l, u] : E \to \mathbb{R}^2_{\geq 0}$. We then want to find an embedding $x : V \to \mathbb{R}^3$ that respects the intervals as well as possible. Note that, in line with previous work, we assume the set S of known distances to be equal to E here. We also assume the edges in E to be unweighted.

As Gansner *et al.*'s maxent-stress algorithm [16] cannot cope with intervals, we solve the iMDGP by first running our implementation of the maxent-stress algorithm with an adapted distance $d' : E \to \mathbb{R}$ that is defined by $d'_{vw} \coloneqq (l_{vw} + u_{vw})/2$. One might expect that, in the resulting embedding, the distances should be roughly in the middle of their interval. This would already be a valid solution to the iMDGP. However, our preliminary experiments show that the output of the maxent-stress algorithm still violates distance constraints even on smaller graphs if called this way. We therefore apply local optimizations to the layout computed by the maxent-stress algorithm. One optimization is based on simulated annealing, while the other is a simple local optimization algorithm.

Optimization of the Embedding with Simulated Annealing. Simulated annealing (SA) is a well-known metaheuristic that can escape local optima by probabilistically accepting neighbor solutions that are worse than the current one, see *e. g.* Talbi [38].

Our SA algorithm is sketched as Algorithm 1 in the full version of this paper [41]. The SA metaheuristic is often especially powerful if the initial solution is randomly chosen and can then jump between local minima. In our setting we receive an embedding from the maxent-stress algorithm as input; its global structure should be already quite good and only some of the given distances are not in their desired intervals. Therefore, our SA algorithm is only used to overcome rather narrow local minima instead of jumping to a completely different solution. The constants in Algorithm 1 have been manually chosen in informal experiments. The outermost loop breaks after a certain number of unsuccessful improvement attempts or if the SA temperature is really low. The second loop iterates until an equilibrium w. r. t. the current temperature is reached – here controlled by the number of iterations and modifications. As we do not want to get completely different solutions for reasons mentioned above, we choose a low start temperature and decrease it rather quickly.

Within the innermost loop a new neighbor solution is computed. In fact, we use parallelism here to reduce the running time of the algorithm. While this can lead to some data races if the position of a vertex is altered by more than one thread, we did not experience any significant decrease in quality. The number of total iterations, steps with no improvement and the number of modifications are all chosen rather small to keep the running time low.

The main ingredients of the innermost loop are

(i) the local error criterion,
(ii) the local optimizer that computes a neighbor solution, and
(iii) the acceptance function.

We define $\text{error}_{vw}(x)$ as $\text{error}_{vw}(x) \coloneqq \max\{l_{vw} - \|x_v - x_w\|, \|x_v - x_w\| - u_{vw}, 0\}^2$. and the local error of an edge $\{v, w\}$ as:

$$\text{localError}(\{v, w\}, x) \coloneqq \text{error}_{vw}(x) + \sum_{u \in N(v) \setminus \{w\}} \text{error}_{vw}(x) + \sum_{u \in N(w) \setminus \{v\}} \text{error}_{uw}(x),$$

where $N(v)$ denotes the neighborhood (*i. e.* the set of incident vertices) of v.

In each iteration a new neighbor solution is computed for an edge $\{v, w\}$. To this end, we apply a force-based approach that takes the edge lengths to their neighbors and the edge length of $\{v, w\}$ into account. The idea is to model the local system similarly to the spring

embedder model [13] and the force-directed algorithm by Fruchterman and Reingold [15]. The difference to those algorithms is that we have to deal with an interval for the length of an edge. In our local force optimization step, we only change the positions of v and w while keeping adjacent vertices fixed. We say an edge $\{v, w\}$ is *violating* its distance constraint if the $\text{error}_{vw}(x)$ is larger than 10^{-9} (and not exactly 0 due to numerical reasons). In our spring model only the violating edges account for a repulsive or attractive force, while the other edges do not take part in the force calculation. In our model each spring has its equilibrium state in an interval that corresponds to the interval of the respective edge it models.

For an optimization on edge $\{v, w\}$, the forces acting on v and w are a combination of attractive and repulsive forces: $f(v) \coloneqq f_{rep}(v) + f_{attr}(v)$ and $f(w) \coloneqq f_{rep}(w) + f_{attr}(w)$. The repulsive and attractive forces for a vertex v are defined as $f_{rep}(v) \coloneqq \sum_{w \in N_{rep}(v)} (x_v - x_w) \cdot \frac{l_{wv}^2}{\|x_w - x_v\|^2}$ and $f_{attr}(v) \coloneqq \sum_{w \in N_{attr}(v)} (x_w - x_v) \cdot \frac{u_{wv}^2}{\|x_w - x_v\|^2}$, where $N_{rep}(v) \subseteq N(v)$ is the set of neighbors of v that are too close to v (*i. e.,* the edge is shorter than its lower bound) and $N_{attr}(v) \subseteq N(v)$ is the set of neighbors of v that are too far away (*i. e.,* the edge is longer than its upper bound). Finally, the acceptance function always accepts improving changes. Error-increasing changes are probabilistically accepted according to the Boltzmann distribution based on the local error (as in many cases [38]).

A Simple Local Optimization Algorithm. In addition to our SA optimization algorithm, we propose another simple local optimization algorithm. During one iteration the algorithm sorts the edges by their error (*i. e.,* the deviation from the edge's given distance interval) in descending order. For an edge $\{v, w\}$ having a length that is not in the given distance interval, the algorithm either prolongates or shortens the edge length such that it is exactly as long as the upper or lower bound given by the distance interval, respectively. We only accept the change if we reduce the local edge error. If we change the length of an edge $\{v, w\}$, we lock the other incident edges of v and w for the remainder of the current iteration to prevent an oscillating effect. We perform a maximum of 50 iterations or less if there is no improvement between two successive iterations. Pseudocode of the method is shown as Algorithm 2 in the full version of this paper [41].

3.2 Intervals with Confidence Values: wiMDGP

Some distances can be measured more accurately than others in common biomolecular experimental methods. To account for this, we add a confidence to each interval. Such a confidence states how certain it is that the actual distance is contained in this interval, leading us to the following problem definition:

▶ **Definition 2** (Weighted Interval Distance Geometry (Optimization) Problem (wiDGP))**.** Let a simple undirected graph $G = (V, E)$, a distance interval function $d = [l, u]$, a confidence function $p : E \to \mathbb{R}$, and an integer $k > 0$ be given. Then minimize the following function:

$$\sum_{\{v,w\} \in E} \omega_{vw} \cdot \text{error}_{\{v,w\}}(x), \tag{4}$$

where the weight ω_{vw} depends on the edge's confidence value c_{vw}.

In order to support wiMDGP, we adapt the maxent-stress algorithm as well as the other two optimization algorithms. For wiMDGP we can use the weights ω_{vw} from Eq. (3), the maxent-stress optimization problem, as a penalty that increases the error of an edge if the confidence that the distance lies in the interval is high. After some manual parameter

tuning, we have chosen the following function to define the weighting factors: $\omega_{vw} := 1 + 5 \exp^{-5(1-c_{vw})}$, where c_{vw} denotes the confidence for edge $\{v, w\}$. This way, the weight of an edge is roughly in the interval $[1, 6]$ and increases rapidly between 0.7 and 1. A confidence between 0 and 0.6 only tells us that we cannot be very certain about the distance and thus, the error term should not vary too much. For higher confidence values, we can be quite certain that the distance interval is correct, so we need to penalize the errors of such edges significantly higher. Choosing a larger interval for the weights turned out not to be beneficial in terms of solution quality in preliminary experiments.

Both our SA and simple local optimization algorithm use an adapted error function for an edge that includes the weighting function above. In our simple local optimizer, we additionally change the sorting of the edges such that edges with higher confidence are considered first. Confidence ties are broken by choosing the edge with larger error first.

3.3 Implementation Details

Initial layout. For computing the initial layout, we implemented three algorithms. In addition to PivotMDS [7], which was used by Gansner *et al.* [16], these are two random vertex placement algorithms. The first one is a very simple method and places the coordinates randomly in a k-dimensional hypercube with predefined side length.

The second one does include some of the distance information provided by the input. Given an edge $\{v, w\}$ and the coordinates of vertex v, we place w at the boundary of a k-dimensional hypersphere with radius d_{vw} and centered at v. Some more details can be found in the full version [41].

Finally, PivotMDS is an approximation algorithm for multidimensional scaling that is based on sampling; for a detailed description the reader is referred to Brandes and Pich [7].

Preliminary experiments indicated that PivotMDS and the random hypersphere approach fare similarly well. Since PivotMDS turned out to be more robust in terms of solution quality when applied to protein instances, we use it in all the following experiments. Its slower speed is more than outweighed by the more expensive maxent-stress algorithm.

Approximating the entropy term. The entropy term in Eq. (3) iterates over all elements not in S. As the set S is usually sparse, this computation would thus require quadratic running time. Thus, it is important to approximate the distances required for the entropy calculations. We implemented both the well-known Barnes-Hut approximation (also used by Gansner *et al.*) as well as well-separated pair decomposition (WSPD) [8].

Additionally, we evaluate the entropy lazily: instead of computing the entropy term in each iteration, we recompute it only when the function $\lfloor 5 \log i \rfloor$ changes, where i is the iteration number. We expect the entropy term to significantly change more frequently at the beginning of a new iteration; thus, we use a function that causes the algorithm to recompute the entropy more often at lower iteration numbers. Lipp *et al.* [21] use the same function for reducing the running time of their WSPD algorithm. This "lazy" computation of the entropy term significantly reduces the running time while the quality does not deteriorate much.

Solving the Laplacian linear systems. Recall that Gansner *et al.* derived an iteration of subsequent Laplacian linear systems for optimizing maxent-stress. They use the conjugate gradient method (CG) as a Laplacian solver in their implementation. The conjugate gradient method has superlinear time complexity. That is why we use lean algebraic multigrid (LAMG) instead, a fast solver proposed by Livne and Brandt [22] with linear empirical running time. We use our C++ implementation of LAMG; it is available in NetworKit and has been used

Table 1 Proteins for distance geometry benchmarks and their basic properties [4]. Listed are the protein data bank (PDB) code [31], and the size in atoms (vertices), amino acid residues, the number of edges equivalent to covalent bonds (= bonds edges) and the number of edges with atoms closer to each other than 5Å without being a covalent bond (= contact edges).

Protein	# atom/ vertices	# residues	bond edges	contact edges
1ptq	402	50	412	3987
1lfb	641	78	654	6320
1gpv	735	87	696	7208
1f39	767	101	788	7621
1ax8	1003	130	1016	10527
1rgs	2015	264	2053	20731
1toa	2138	277	2181	23168
1kdh	2846	356	2904	30655
1bpm	3671	481	3744	41027
1mqq	5510	675	5665	62396

for other Laplacian graph problems before [3]. An alternative multilevel approach for solving the linear systems exists [24], but is harder to adapt to the present scenario.

Optimization Workflow. We combine our two optimization algorithms into one workflow: the solution found by the SA algorithm can often be further improved by a subsequent run of our simple local optimization algorithm. Sometimes it happens that the SA algorithm only finds a slightly worse solution compared to the maxent-stress algorithm. In this case we ignore the SA solution and only run the simple optimization algorithm.

4 Experiments

In this section we present a representative subset of our experiments and their results. To this end, we implemented our algorithm in C++ based on NetworKit [36], an open-source toolkit for graph algorithms and in particular interactive large-scale network analysis. We call our algorithm MOBi (for **M**axent-stress **O**ptimization of **Bi**omolecular models) and compare it with DGSOL [25] (C/Fortran code) and DISCO [18] (compiled Matlab code), two of the very few publicly available established tools that can handle inexact inputs with intervals. Experimental settings are further detailed in the full version of this paper [41].

4.1 Instances

To test the accuracy and efficiency of our approach in a real-world setting, we work on 10 proteins of different sizes [4] taken from the protein data bank (PDB) [31], see Table 1. These proteins range from small globular proteins with 50 amino acids to large proteins with about 700 amino acids. We only consider *ATOM* entries in the file, which provide atomic coordinates. Also, we only work on the first chain in the case of multiple protein chains. Based on the coordinates of a protein, we can construct an instance for the various distance geometry problems. For each experiment we actually create three instances per protein (*i. e.,* different contact distance information) to eliminate effects of particularly good/bad sets of input data. Also, each instance (*i. e.,* same contact distance information) is re-run three times to eliminate stochasticity effects. The displayed data per protein are averaged over these three instances and three respective runs.

We use a percentage p of all atom distances $\{v, w\} \in \binom{V}{2}$ for which $d_{vw} = \|x_v - x_w\|$ is below a cut-off distance of 5 Å, where x denotes the coordinates from the protein file. We use 5 Å because this approximates the distance that can be determined by NMR experiments [42, 37] and since it is a typical cutoff in determining so-called contact maps (*i. e.* binary matrices that store only adjacencies whose length is below the cutoff) [34, 28]. To construct an instance for iMDGP, we introduce the interval $[d_{vw} - \underline{\epsilon}, d_{vw} + \overline{\epsilon}]$, where $\underline{\epsilon}, \overline{\epsilon} = d_{vw} \cdot \mathcal{N}(0, \sigma^2)$ and σ is the standard deviation. We denote instances created this way as *normal*-iMDGP instances.

In contrast, the *bonds*-iMDGP more closely reflects standard NMR experiments by taking protein biochemistry into account. As all covalent bonds of a protein are known, instances can be assumed to have full knowledge of exact distances for these bonds: Chemically, the length of covalent bonds fluctuates very little, hence the interval for these bonds edges $\{v, w\}$ consists of a single distance $[d_{vw}, d_{vw}]$. In addition to the distance information of the bonds, we add more distances the same way as for constructing a *normal*-iMDGP instance.

4.2 Results & Discussion

To quantify the structure determination quality of the different approaches, we compare them by RMSD. For the *normal*-iDGP test instances, we observe low RMSD for MOBi and DISCO, while DGSOL performs significantly worse (cf. Tables 2 and 3 as well as Table 4 in Appendix A.3 of the full version [41]. As expected, the RMSD gets higher if less distance information is provided (the instances in Tables 2 and 3 provide only 50% and 30% of the contact edges, respectively, while the instances in Table 4 provide 70%). The RMSD values do not increase, as one might expect *a priori*, necessarily with instance size (number of atoms/ vertices). Instead, these instances have more edges, which might make the embedding computationally more demanding but also of good quality. Indeed, MOBi and DISCO yield good embeddings regardless of system size. DGSOL performs worse for larger systems. Also, MOBi is more consistent than DISCO and performs best in nearly all instances, in particular for those with less information (cf. instances 1gpv and 1rgs in Table 3). Similarly, the running times of MOBi are about an order of magnitude faster than DGSOL and DISCO, with DGSOL being slightly faster than DISCO. There is an overall trend of increased running times with system size, but some instances seem particular hard to compute (*e. g.* instances 1gpv, 1rgs, and 1toa in Table 3). Gaussian noise on the intervals does not significantly alter the results: given a relatively high amount of distance information, MOBi and DISCO produce embeddings of very high quality with RMSD < 1 Å (see Tables 4, 5 and 6 in Appendix A.3 in the full version of this paper [41]. It should be noted, though, that in Table 5 DISCO yields the majority of best results in terms of solution quality – but MOBi is usually not far behind.

For the *bonds*-iMDGP instances, we display the results only for MOBi (DISCO and DGSOL show the same respective trends as above) as a heatmap in Figure 1. For very low amounts of additional distance information (1–2%) in addition to the bonds, MOBi is unable to provide high-quality embeddings as displayed by RMSD values > 5 Å. When provided as little as 8–12% distance information in addition to the bonds, the structure determination quality becomes below 3 Å RMSD, *i. e.,* it approaches the wet-lab resolution. Interestingly, the amount of Gaussian noise does not strongly influence the embedding quality. Exemplary structure embeddings are displayed in the full version of this paper [41]. While the overall quality appears good here as well, one can see that most structural errors are found at the surface, where fewer edges are available.

The experimental results for wiMDGP are shown in Table 7 in the full version [41]. As these instances cannot be directly compared against other cases, we merely report that MOBi

Table 2 Performance results on 50% $\sigma = 0.1$ *normal*-iDGP instances. Best results in bold font.

	RMSD / Å			time / s		
Protein	MOBi	DGSOL	DISCO	MOBi	DGSOL	DISCO
1ptq	0.46	8.05	**0.45**	**1.56**	9.85	13.70
1lfb	**0.87**	10.51	1.00	**2.73**	22.99	25.29
1gpv	**0.68**	14.35	0.91	**7.51**	120.35	128.54
1f39	**0.53**	16.45	0.69	**5.55**	70.22	70.80
1ax8	**0.51**	12.23	0.55	**3.91**	39.42	48.49
1rgs	**0.60**	17.56	1.05	**7.10**	112.49	155.52
1kdh	**0.88**	19.41	1.06	**8.64**	173.70	186.00
1toa	**0.48**	23.72	0.86	**14.14**	181.96	311.92
1bpm	**0.48**	21.73	0.50	**12.94**	221.18	264.02
1mqq	**0.34**	23.56	0.40	**18.22**	381.65	519.58

Table 3 Performance results on 30% $\sigma = 0.1$ *normal*-iDGP instances. Best results in bold font.

	RMSD / Å			time / s		
Protein	MOBi	DGSOL	DISCO	MOBi	DGSOL	DISCO
1ptq	**1.05**	9.29	1.09	**1.38**	12.39	13.30
1lfb	**1.50**	11.68	1.53	**2.79**	26.76	22.86
1gpv	**1.29**	16.29	3.48	**6.64**	112.08	114.63
1f39	**1.06**	17.27	1.74	**5.18**	82.75	78.07
1ax8	**1.07**	12.95	1.34	**3.85**	47.51	44.43
1rgs	**1.37**	18.03	5.54	**40.33**	120.89	120.59
1toa	**1.00**	23.47	1.28	**13.10**	204.94	325.64
1kdh	**1.46**	20.33	1.51	**8.28**	174.67	169.41
1bpm	**0.93**	22.08	1.37	**11.49**	237.23	255.87
1mqq	**0.82**	23.45	0.96	**16.36**	216.39	529.09

produces high quality solutions, typically with RMSD < 1.0 Å and comparable running times to the other instances with MOBi. To truly assess the use of this wiMDGP implementation on real-world data, one would have to work on curated wet-lab experimental data [42, 40] or co-evolutionary signals [35]. This is clearly outside the scope of this paper.

Overall, in particular for limited and noisy distance information, MOBi provides consistently embeddings with higher quality and does so at significantly lower running times than both DISCO and DGSOL. On average (geometric means over quotients for each of the Tables 2 to 6), MOBi is between 13x and 20x faster than DISCO. At the same time its RMSD values are on average 17% to 41% better – except for Table 5, where DISCO is 12% better. DGSOL is not competitive in terms of solution quality and also 6x to 13x slower.

For *normal*-iDGP instances and very high amounts of distance information, the MOBi embeddings provide RMSD < 1 Å, which is below the typical resolution of NMR or X-ray [40]; even providing only $p = 30\%$ leads to high quality embeddings. Both MOBi and DISCO perform considerably better than older algorithms such as[5], where some RMSD > 5 Å were reported. In the more realistic scenario of *bonds*-iMDGP instances with all bond edges provided, only few contact edges (8–12%) can already lead to high quality embeddings with MOBi. Thus, we are confident that our algorithm will lead to improved interpretation of wet-lab experiments in particular in cases with sparse data, such as sparse NMR experiments.

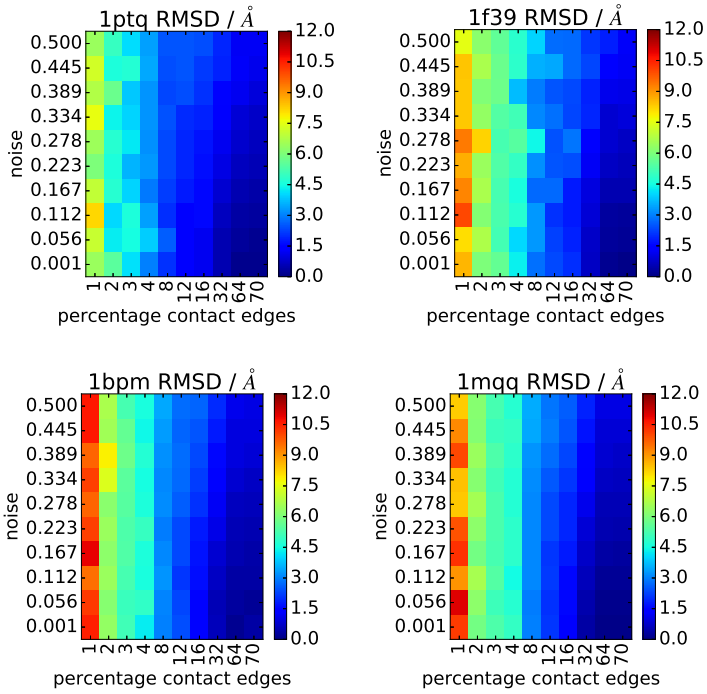


Figure 1 Quality results for bonds-iDGP instances. The horizontal axis shows the amount of contact edges in addition to bond edges provided, the vertical axis varies the σ of Gaussian noise.

5 Conclusions

This paper provides a significant step towards nearly-interactive protein structure determination. To this end, we implemented the maxent-stress algorithm [16] and incorporated first of all a faster Laplacian solver. Based on this implementation we extended the graph drawing algorithm to handle distance geometry problems where the distances are either exact or come in the form of intervals, optionally with some confidence. Comparing our algorithm with two publicly available competitors shows that we are able to significantly outperform both of them in terms of running time, while usually providing embeddings with higher quality. For the more realistic bonds-iDGP instances, our algorithm is able to compute high quality protein structures with limited and noisy information. Most errors can be found at the surface of the proteins, where only few edges can guide the optimization process.

While some related work can provide even higher solution quality (*e. g.* [14]), it can only do so at the expense of an enormous increase in running time. The strength of our work is the combination of low running time, good and consistent solution quality, and genericity.

The evaluation of our algorithm on real-world instances whose distance matrices are derived from chemical bonds and real NMR experiments is ongoing and shows very promising results, too. In the future it also seems promising to use our algorithm for bootstrapping more sophisticated and thus more expensive algorithms for protein structure determination (such as refining the resulting structures in physics-based force fields similar to [35, 10]). Moreover, further improvements of the resulting structures could be achieved by re-weighting edges by their density; such an approach would consider surfaces more strongly.

Acknowledgments. The authors thank Michael Kovermann (University of Konstanz) for fruitful discussions.

References

1 Le Thi Hoai An. Solving large scale molecular distance geometry problems by a smoothing technique via the gaussian transform and d.c. programming. *Journal of Global Optimization*, 27(4):375–397, 2003. doi:10.1023/a:1026016804633.

2 Le Thi Hoai An and Pham Dinh Tao. Large-scale molecular optimization from distance matrices by a d.c. optimization approach. *SIAM Journal on Optimization*, 14(1):77–114, jan 2003. doi:10.1137/s1052623498342794.

3 Elisabetta Bergamini, Michael Wegner, Dimitar Lukarski, and Henning Meyerhenke. Estimating current-flow closeness centrality with a multigrid laplacian solver. In *Proc. 7th SIAM Workshop on Combinatorial Scientific Computing, CSC 2016*, pages 1–12. SIAM, 2016. doi:10.1137/1.9781611974690.ch1.

4 Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, Talapady N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The protein data bank. *Nucleic Acids Research*, 28(1):235–242, Jan 2000. doi:10.1093/nar/28.1.235.

5 Pratik Biswas, Kim-Chuan Toh, and Yinyu Ye. A distributed SDP approach for large-scale noisy anchor-free graph realization with applications to molecular conformation. *SIAM Journal on Scientific Computing*, 30(3):1251–1277, jan 2008. doi:10.1137/05062754x.

6 Leonard M. Blumenthal. *Theory and Applications of Distance Geometry*, volume 347. Oxford, 1953.

7 Ulrik Brandes and Christian Pich. Eigensolver methods for progressive multidimensional scaling of large data. In *Graph Drawing*, pages 42–53. Springer Science + Business Media, 2007. doi:10.1007/978-3-540-70904-6_6.

8 Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90, jan 1995. doi:10.1145/200836.200853.

9 Gordon M. Crippen, Timothy F. Havel, et al. *Distance Geometry and Molecular Conformation*, volume 74. Research Studies Press Taunton, UK, 1988.

10 Angel E. Dago, Alexander Schug, Andrea Procaccini, James A. Hoch, Martin Weigt, and Hendrik Szurmant. Structural basis of histidine kinase autophosphorylation deduced by integrating genomics, molecular dynamics, and mutagenesis. *Proceedings of the National Academy of Sciences*, 109(26):E1733–E1742, 2012.

11 Eleonora De Leonardis, Benjamin Lutz, Sebastian Ratz, Simona Cocco, Rémi Monasson, Alexander Schug, and Martin Weigt. Direct-coupling analysis of nucleotide coevolution facilitates rna secondary and tertiary structure prediction. *Nucleic acids research*, 43(21):10444–10455, 2015.

12 Qunfeng Dong and Zhijun Wu. A linear-time algorithm for solving the molecular distance geometry problem with exact inter-atomic distances. *Journal of Global Optimization*, 22(1/4):365–375, 2002. doi:10.1023/a:1013857218127.

13 Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:146–160, 1984.

14 Xingyuan Fang and Kim-Chuan Toh. Using a distributed SDP approach to solve simulated protein molecular conformation problems. In *Distance Geometry*, pages 351–376. Springer Science + Business Media, nov 2012. doi:10.1007/978-1-4614-5128-0_17.

15 Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, Nov 1991. doi:10.1002/spe.4380211102.

16 Emden R. Gansner, Yifan Hu, and Stephen North. A maxent-stress model for graph layout. *IEEE Transactions on Visualization and Computer Graphics*, 19(6):927–940, jun 2013. doi:10.1109/tvcg.2012.299.

17 Oliver F. Lange, Nils-Alexander Lakomek, Christophe Farès, Gunnar F. Schröder, Korvin F. A. Walter, Stefan Becker, Jens Meiler, Helmut Grubmüller, Christian Griesinger, and

Bert L. De Groot. Recognition dynamics up to microseconds revealed from an rdc-derived ubiquitin ensemble in solution. *science*, 320(5882):1471–1475, 2008.

18 Ngai-Hang Z. Leung and Kim-Chuan Toh. An SDP-based divide-and-conquer algorithm for large-scale noisy anchor-free graph realization. *SIAM Journal on Scientific Computing*, 31(6):4351–4372, jan 2010. doi:10.1137/080733103.

19 Leo Liberti, Carlile Lavor, Nelson Maculan, and Fabrizio Marinelli. Double variable neighbourhood search with smoothing for the molecular distance geometry problem. *Journal of Global Optimization*, 43(2-3):207–218, aug 2009. doi:10.1007/s10898-007-9218-1.

20 Leo Liberti, Carlile Lavor, Nelson Maculan, and Antonio Mucherino. Euclidean distance geometry and applications. *SIAM Review*, 56(1):3–69, jan 2014. doi:10.1137/120875909.

21 Fabian Lipp, Alexander Wolff, and Johannes Zink. Faster force-directed graph drawing with the well-separated pair decomposition. In *Graph Drawing and Network Visualization – 23rd International Symposium, GD 2015, Los Angeles, CA, USA, September 24-26, 2015, Revised Selected Papers*, volume 9411 of *LNCS*, pages 52–59. Springer, 2015. doi:10.1007/978-3-319-27261-0_5.

22 Oren E. Livne and Achi Brandt. Lean algebraic multigrid (LAMG): Fast graph laplacian linear solver. *SIAM Journal on Scientific Computing*, 34(4):B499–B522, Jan 2012. doi:10.1137/110843563.

23 Jeffrey W. Martin, Anthony K. Yan, Chris Bailey-Kellogg, Pei Zhou, and Bruce R. Donald. A geometric arrangement algorithm for structure determination of symmetric protein homooligomers from noes and rdcs. *Journal of Computational Biology*, 18(11):1507–1523, 2011.

24 Henning Meyerhenke, Martin Nöllenburg, and Christian Schulz. Drawing large graphs by multilevel maxent-stress optimization. In *Graph Drawing and Network Visualization – 23rd International Symposium, GD 2015, Los Angeles, CA, USA, September 24-26, 2015, Revised Selected Papers*, volume 9411 of *LNCS*, pages 30–43. Springer, 2015. doi:10.1007/978-3-319-27261-0_3.

25 Jorge J. Moré and Zhijun Wu. Global continuation for distance geometry problems. *SIAM Journal on Optimization*, 7(3):814–836, aug 1997. doi:10.1137/s1052623495283024.

26 Jorge J. Moré and Zhijun Wu. Distance geometry optimization for protein structures. *Journal of Global Optimization*, 15(3):219–234, 1999.

27 Antonio Mucherino, Carlile Lavor, Leo Liberti, and Nelson Maculan, editors. *Distance Geometry: Theory, Methods, and Applications*. Springer Science + Business Media, 2013. doi:10.1007/978-1-4614-5128-0.

28 Jeffre K. Noel, Paul C. Whitford, and Onuchic Jose N. The shadow map: A general contact definition for capturing the dynamics of biomolecular folding and function. *J Phys Chem B*, 116(29):8692–8702, 2013. doi:10.1021/jp300852d.

29 Sergey Ovchinnikov, Hahnbeom Park, Neha Varghese, Po-Ssu Huang, Georgios A. Pavlopoulos, David E. Kim, Hetunandan Kamisetty, Nikos C. Kyrpides, and David Baker. Protein structure determination using metagenome sequence data. *Science*, 355(6322):294–298, 2017.

30 Carol A. Rohl, Charlie E. M. Strauss, Kira M. S. Misura, and David Baker. Protein structure prediction using rosetta. *Methods in enzymology*, 383:66–93, 2004.

31 Peter W. Rose, Andreas Prlić, Chunxiao Bi, Wolfgang F. Bluhm, Cole H. Christie, Shuchismita Dutta, Rachel Kramer Green, David S. Goodsell, John D. Westbrook, Jesse Woo, Jasmine Young, Christine Zardecki, Helen M. Berman, Philip E. Bourne, and Stephen K. Burley. The rcsb protein data bank: views of structural biology for basic and applied research and education. *Nucleic Acids Research*, 43(D1):D345, 2015. doi:10.1093/nar/gku1214.

32 James B. Saxe. *Embeddability of weighted graphs in k-space is strongly NP-hard*. Carnegie-Mellon University, Department of Computer Science, 1980.

33 A. Schug, T. Herges, and W. Wenzel. Reproducible protein folding with the stochastic tunneling method. *Physical review letters*, 91(15):158102, 2003.

34 Alexander Schug and José N. Onuchic. From protein folding to protein function and biomolecular binding by energy landscape theory. *Current opinion in pharmacology*, 10(6):709–714, 2010.

35 Alexander Schug, Martin Weigt, José N. Onuchic, Terence Hwa, and Hendrik Szurmant. High-resolution protein complexes from integrating genomic information with molecular simulation. *Proceedings of the National Academy of Sciences*, 106(52):22124–22129, 2009.

36 Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, Dec 2016.

37 J. B. Stothers. *Carbon-13 NMR Spectroscopy: Organic Chemistry, A Series of Monographs*, volume 24. Elsevier, 2012.

38 El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

39 Guido Uguzzoni, Shalini John Lovis, Francesco Oteri, Alexander Schug, Hendrik Szurmant, and Martin Weigt. Large-scale identification of coevolution signals across homo-oligomeric protein interfaces by direct coupling analysis. *Proceedings of the National Academy of Sciences*, 114(13):E2662–E2671, 2017.

40 D. Voet and J. G. Voet. *Biochemistry, 4th Edition*. John Wiley & Sons, 2010.

41 Michael Wegner, Oskar Taubert, Alexander Schug, and Henning Meyerhenke. Maxent-stress optimization of 3D biomolecular models. *arXiv preprint arXiv:1706.06805*, Jun 2017. URL: `https://arxiv.org/abs/1706.06805`.

42 Kurt Wüthrich. Protein structure determination in solution by nmr spectroscopy. *Journal of Biological Chemistry*, 265(36):22059–22062, 1990.