20th International Symposium on Experimental Algorithms

SEA 2022, July 25–27, 2022, Heidelberg, Germany

Edited by

Christian Schulz

Bora Uçar

LIPIcs – Vol. 233 – SEA 2022 www.dagstuhl.de/lipics

Editors

Christian Schulz
Heidelberg University, Germany
christian.schulz@informatik.uni-heidelberg.de

Bora Uçar
CNRS, Laboratoire LIP, Lyon, France
bora.ucar@ens-lyon.fr

ACM Classification 2012
Theory of computation → Design and analysis of algorithms

ISBN 978-3-95977-251-8

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-251-8.

Publication date
July, 2022

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 https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.SEA.2022.0

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

https://www.dagstuhl.de/lipics

Contents

Papers

20th International Symposium on Experimental Algorithms (SEA 2022).
Editors: Christian Schulz and Bora Uçar
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

We are pleased to present the collection of papers accepted for presentation at the 20th edition of the International Symposium on Experimental Algorithms (SEA 2022) which was held in Heidelberg from 25th July 2022 to 27th July 2022.

SEA, previously known as Workshop on Experimental Algorithms (WEA), is an international forum for researchers in the area of the design, analysis, and experimental evaluation and engineering of algorithms, as well as in various aspects of computational optimization and its applications (telecommunications, transport, bioinformatics, cryptography, learning methods, etc.). The symposium aims at attracting papers from both the Computer Science and the Operations Research/Mathematical Programming communities. Submissions to SEA are requested to present significant contributions supported by experimental evaluation, methodological issues in the design and interpretation of experiments, the use of heuristics and meta-heuritics, or application-driven case studies that deepen the understanding of the complexity of a problem. A main goal of SEA is also the creation of a friendly environment that can lead to and ease the establishment or strengthening of scientific collaborations and exchanges among attendees. For this reason, the symposium solicits high-quality original research papers (including significant work-in-progress) on any aspect of experimental algorithms.

Each submission that was made to SEA 2022 was reviewed by at least three Program Committee members or external reviewers. After a careful peer review and evaluation process, 24 papers were accepted for presentation and for inclusion in the LIPIcs proceedings, according to the reviewers' recommendations. The acceptance rate was 49%. The scientific program of the symposium also includes presentations by three keynote speakers: Tobias Achterberg (R&D Gurobi, Germany), Cynthia A. Phillips (Sandia National Laboratories, US) and Paul Spirakis (University of Liverpool, England and University of Patras, Greece). The conference reintroduced a best paper award. The best paper was selected by a selection committee formed by Kathrin Hanuaer, Simon Puglisi, Alex Pothen (chair of the best paper selection committee), and Julian Shun. Based on the committee's careful assessment, the best paper was selected to be "Fast Succinct Retrieval and Approximate Membership using Ribbon" by Peter C. Dillinger, Lorenz Hubschle-Schneider, Peter Sanders and Stefan Walzer. We congratulate the authors and thank the selection committee.

The 20th edition of SEA was organized by the Algorithm Engineering group at Heidelberg University. We thank Catherine Proux-Wieland, Marcelo Fonseca Faraj and Ernestine Großmann for their help with the organization of the symposium. We also thank the faculty of mathematics and computer science for providing us with the facilities for the conference. Moreover, we would like to thank the SEA steering committee for giving us the opportunity to host SEA 2022. Thanks are also due to the editors of the ACM Journal of Experimental Algorithmics for their interest in hosting a special issue of the best papers presented at SEA 2022. We would like to thank Google for providing us with financial support as well as HGS MathComp for providing travel support for young scientists. Finally, we express our gratitude to the members of the Program Committee as well as numerous subreviews for their support, collaboration, and excellent work.

Heidelberg, July 2022
Christian Schulz and Bora Uçar

20th International Symposium on Experimental Algorithms (SEA 2022).
Editors: Christian Schulz and Bora Uçar
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Steering Committee

- Edoardo Amaldi (Politecnico di Milano, Italy)
- David A. Bader (New Jersey Institute of Technology US)
- Josep Diaz (Universitat Politecnica de Catalunya, Spain)
- Giuseppe F. Italiano (University of Rome Tor Vergata, Italy)
- Klaus Jansen (University of Kiel, Germany)
- Kurt Mehlhorn (Max-Planck-Institut für Informatik, Germany)
- Ian Munro (University of Waterloo, Canada)
- Sotiris Nikoletseas (Patras University, Greece)
- Jose Rolim (University of Geneva, Switzerland)
- Pavlos Spirakis (University of Liverpool, UK)

20th International Symposium on Experimental Algorithms (SEA 2022).
Editors: Christian Schulz and Bora Uçar

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

Organization

Program Chairs

- Christian Schulz, Heidelberg University (Germany)
- Bora Uçar, CNRS and LIP, ENS de Lyon (France)

Program Committee

- Petra Berenbrink, Universität Hamburg (Germany)
- Thomas Bläsius, Karlsruhe Institute of Technology (Germany)
- Matteo Ceccarello, Freie Universität Bozen (Italy)
- Stephane Chretien, University of Lyon II (France)
- David Coudert, INRIA (France)
- Simone Faro, Università di Catania (Italy)
- D'Angelo Gianlorenzo, Gran Sasso Science Institute (Italy)
- Marc Goerigk, University of Siegen (Germany)
- Susana Ladra, Universidade Da Coruna (Spain)
- Alexander van der Grinten, Humboldt-Universität zu Berlin (Germany)
- Kathrin Hanauer, University of Vienna (Austria)
- Juha Kärkkäinen, University of Helsinki (Finland)
- M. Oğuzhan Külekci Istanbul Technical University (Turkey)
- Stefano Leucci, University of L'Aquila (Italy)
- Marco Lübbecke, RWTH Aachen (Germany)
- Gonzalo Navarro, University of Chile (Chile)
- Rolf Niedermeier, Technische Universtät Berlin (Germany)
- Nicolas Nisse, INRIA (France)
- Manuel Penschuck, Goethe University Frankfurt (Germany)
- Cynthia A. Phillips, Sandia National Laboratories (US)
- Marcin Pilipczuk, University of Warsaw (Poland)
- Mustafa Pınar, Bilkent Univeristy (Turkey)
- Michael Poss, LIRMM (France)
- Alex Pothen, Purdue University (US)
- Simon Puglisi, University of Helsinki (Finland)
- Sebastian Schlag, Apple (US)
- Stefan Schmid, Technical University of Berlin (Germany)
- Melanie Schmidt, HHU Düsseldorf (Germany)
- Shikha Singh, Williams College (US)
- Julian Shun, Massachusetts Institute of Technology (US)
- Bertrand Simon, CNRS (France)
- Darren Strash, Hamilton College (US)
- Sabine Storandt, University of Konstanz (Germany)
- Stefan Szeider, Technical University Vienna (Austria)
- Yihan Sun, University of California (US)
- Annegret Wagler, Universitaire des Cézeaux (France)
- Norbert Zeh, Dalhousie University in Halifax (Canada)

20th International Symposium on Experimental Algorithms (SEA 2022).
Editors: Christian Schulz and Bora Uçar

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

Best Paper Committee

- Kathrin Hanauer, University of Vienna (Austria)
- Alex Pothen (chair), Purdue University (US)
- Simon Puglisi, University of Helsinki (Finland)
- Julian Shun, Massachusetts Institute of Technology (US)

External Reviewers

Thorsten Koch, Marcus Wilhelm, Esmaeil Delfaraz, Hélène Toussaint, Till Fluschnik, Shangdi Yu, Letong Wang, Yan Gu, Alessio Conte, Jessica Shi, Manuel Cáceres, Mirko Rossi, Changwan Hong, Adrián Gómez Brandón, Diego Arroyuelo, Marcelo Fonseca Faraj, Ripon Chakrabortty, Adrian Feilhauer, Ernestine Großmann, Dustin Cobas, Tomohiro Koana, Arie Koster, Daniel Karapetyan, Soumen Maity, Lijun Chang, Christopher Weyand, S M Ferdous, Stefano Scafiti, Malin Rau, Svetlana Kulagina, Javier Marenco, Yuji Shinano, Diego Diaz, Cristian Urbina, Janis S. Neufeld, Xiaojun Dong.

Discrete Hyperbolic Random Graph Model

Dorota Celińska-Kopczyńska
Institute of Informatics, University of Warsaw, Poland

Eryk Kopczyński
Institute of Informatics, University of Warsaw, Poland

Abstract

The hyperbolic random graph model (HRG) has proven useful in the analysis of scale-free networks, which are ubiquitous in many fields, from social network analysis to biology. However, working with this model is algorithmically and conceptually challenging because of the nature of the distances in the hyperbolic plane. In this paper, we propose a discrete variant of the HRG model (DHRG) where nodes are mapped to the vertices of a triangulation; our algorithms allow us to work with this model in a simple yet efficient way. We present experimental results conducted on networks, both real-world and simulated, to evaluate the practical benefits of DHRG in comparison to the HRG model.

2012 ACM Subject Classification Theory of computation → Random network models; Theory of computation → Routing and network design problems; Human-centered computing → Social network analysis

Keywords and phrases hyperbolic geometry, scale-free networks, routing, tessellation

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.1

Supplementary Material *Software (Source Code and Data)*: `https://figshare.com/articles/software/Discrete_Hyperbolic_Random_Graph_Model_code_and_data_/16624369`

Funding This work has been supported by the National Science Centre, Poland, grant DEC-2016/21/N/HS4/02100.

Acknowledgements We would like to thank all the referees for their comments which have greatly improved the paper.

1 Introduction

Hyperbolic geometry has been discovered by 19th century mathematicians wondering about the nature of parallel lines. One of the properties of this geometry is that the amount of an area in the distance d from a given point is exponential in d; intuitively, the metric structure of the hyperbolic plane is similar to that of an infinite binary tree, except that each vertex additionally connects to two adjacent vertices on the same level.

Recently, hyperbolic geometry has proven useful in modeling hierarchical data [17, 19]. In particular, it has found application in the analysis of scale-free networks, which are ubiquitous in many fields, from network analysis to biology [21]. In the hyperbolic random graph model (HRG), we place the nodes randomly in a hyperbolic disk; nodes that are in a close neighbourhood are more likely to be connected. The properties of a HRG, such as its power-law degree distribution or high clustering coefficient, are similar to those of real-world scale-free networks [13]. Due to high clustering coefficients, HRG is more accurate than earlier models such as Preferential Attachment [1] in modeling real-world networks.

Perhaps the two most important algorithmic problems related to HRGs are *sampling* (generate a HRG) and *MLE embedding*: given a real-world network $H = (V, E)$, map the vertices of H to the hyperbolic plane in such a way that the edges are predicted as accurately as possible. The quality of this prediction is measured with *log-likelihood*, computed with

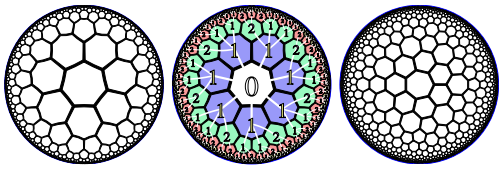


Figure 1 (a) Order-3 heptagonal tiling (G_{710}). (b) Its growth. (c) Bitruncated heptagonal tiling (G_{711}).

the formula $\sum_{v,w} \log p(v,w)$, where $p(v,w)$ is the probability that the model correctly predicts the existence or not of an edge (v,w). Those problems are non-trivial: even simply computing the log-likelihood, using a naive algorithm would require time $O(|V|^2)$. The original paper [21] used an $O(|V|^3)$ algorithm for embedding. Efficient algorithms have been found for generating HRGs and the closely related Geometric Inhomogeneous Random Graphs in expected time $O(|V|)$ [6, 3, 22, 27, 25, 10] and for MLE embedding real-world scale-free networks into the hyperbolic plane in time $\tilde{O}(|V|)$ [4], which was a major improvement over previous algorithms [20, 26]. Embedding has practical applications in link prediction [24] and routing [5, 2].

This paper introduces and experimentally studies the discrete version of the HRG model (DHRG). In the DHRG model, we use a tessellation rather than the hyperbolic plane. Instead of the hyperbolic distance between points, we use the number of steps between two tiles in the tessellation. Our approach has the following advantages:

- Avoiding numerical issues. DHRG is not based on the tuple of coordinates, which makes it immune to serious precision errors resulting from the exponential expansion. This way we solve a fundamental issue for hyperbolic embeddings [2, 23].
- Algorithmic simplicity. In DHRG we find efficient algorithms for sampling, computing the log-likelihood, and improving an embedding by generalizing similar algorithms for trees. Working with DHRG does not require a good understanding of hyperbolic geometry, combating a major drawback of previous approaches.

The first potential objection to our approach is that discrete distances are inaccurate. This inaccuracy comes from two sources: different geometry (distances in Euclidean square grid correspond to the taxicab metric, which is significantly different from the usual Euclidean metric) and discreteness. It is challenging to rigorously study the theoretical effects of discretization on the properties of our model. However, according to our experiments, these issues do not turn out to be threatening – the HRG embeddings are large enough to render discreteness insignificant, and discrete distances are a good approximation of the actual hyperbolic distances (better than in the case of Euclidean geometry).

Our experiments on artificial networks show that using DHRG improves the success rate of greedy routing in 77% of the cases (depending on the parameters). A DHRG embedding can be efficiently improved by moving the vertices so that the log-likelihood becomes better. Our procedure yields about 10% improvement of log-likelihood on state-of-the-art HRG embeddings on real graphs. This result is supported by our extensive simulations on artificial graphs.

2 Prerequisities

In this section, we briefly introduce hyperbolic geometry and the HRG model. A more extensive introduction to hyperbolic geometry can be found, e.g., in [8].

Figure 1 shows the order-3 heptagonal tessellation of the hyperbolic plane in the Poincaré model. In the hyperbolic metric, all the triangles, heptagons, and hexagons on each picture are actually of the same size. The points on the boundary of the disk are infinitely far from the center. The area of a hyperbolic circle of radius r is exponential in r.

The hyperbolic plane (in the Minkowski hyperboloid model) is $\mathbb{H}^2 = \{(x, y, z) : z > 0, z^2 - x^2 - y^2 = 1\}$. The distance between two points $a = (x, y, z)$ and $a' = (x', y', z')$ is $\delta(a, a') = \operatorname{acosh}(zz' - xx' - yy')$. Most introductions to hyperbolic geometry use the Poincaré disk model; however, the Minkowski hyperboloid model is very useful in computational hyperbolic geometry, because the essential operations are simple generalizations of their Euclidean or spherical counterparts. In particular, rotation of a point v by angle α is given by

$$\begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} v,$$

while a translation by x units along the X axis is given by

$$\begin{pmatrix} \cosh x & 0 & \sinh x \\ 0 & 1 & 0 \\ \sinh x & 0 & \cosh x \end{pmatrix} v.$$

We can easily map the Minkowski hyperboloid model to the Poincaré disk model (e.g., for visualization purposes) using stereographic projection:

$$(x', y') = (x/(z + 1), y/(z + 1)).$$

In the hyperbolic polar coordinate system, every point is represented by two coordinates (r, ϕ), where

$$P(r, \phi) = (\cos(\phi)\sinh(r), \sin(\phi)\sinh(r), \cosh(r)).$$

Here, r is distance from the central point $(0, 0, 1)$, and ϕ is the angular coordinate.

▶ **Definition 1.** *The* **hyperbolic random graph model** *has four parameters: n (number of vertices), R (radius), T (temperature), and α (dispersion parameter). Each vertex $v \in V = \{1, \ldots, n\}$ is independently randomly assigned a point $\mu(v) = P(r_v, \phi_v)$, where the distribution of ϕ_v is uniform in $[0, 2\pi]$, and the density of the distribution of $r_v \in [0, R]$ is given by $f(r) = \frac{\alpha \sinh(\alpha r)}{\cosh(\alpha R)-1}$. Then, for each pair of vertices $v, w \in V$, they are independently connected with probability $p(\delta(\mu(v), \mu(w)))$, where $\delta(x, y)$ is the distance between $x, y \in \mathbb{H}^2$, and $p(d) = \frac{1}{1+e^{(d-R)/2T}}$.*

The parameter α controls the power-law exponent $\beta = 2\alpha + 1$ [13]. The parameter T, typically chosen to be in $[0, 1]$, regulates the importance of underlying geometry, and thus the clustering coefficient: with T very close to 0, an edge exists iff $\delta(\mu(v), \mu(w)) < R$, while with larger values of T missing short edges and existing long edges are possible. In [13] and [4], R equals $2\log n + C$, where C is a parameter adjusting the average degree of the resulting graph.

An **MLE embedder** is an algorithm which, given a network $H = (V, E)$, finds a good embedding of H in the hyperbolic plane, i.e., parameters R, T, and α, and a mapping $\mu : V \to \mathbb{H}^2$. The quality of the embedding is measured with *log-likelihood*, computed with the formula

$$\log L(\mu) = \sum_{v<w\in V} \log p_{\{v,w\}\in E}(\delta(\mu(v), \mu(w))),$$

where $p_\phi(d) = p(d)$ if ϕ is true and $1 - p(d)$ if ϕ is false. While not a goal by itself, we can expect that a better embedding (in terms of log-likelihood) will perform better in the applications, such as link prediction, greedy routing, visualization, etc.

Now, we explain the structure of a hyperbolic tessellation, on the example of the order-3 heptagonal tiling (G_{710} from Figure 1b). Let $\delta(t_1, t_2)$ be the distance between two tiles ($\delta(t_1, t_2) = 1$ iff t_1 and t_2 are adjacent). Let $\delta_0(t) = \delta(t, t_0)$ be the distance of tile t from the center tile t_0. We denote the set of tiles t such that $\delta_0(t) = d$ with R_d; for $d > 0$ it is a cycle (in Figure 1b sets R_d are marked with colored rings). Except for the central tile, every tile has one or two adjacent tiles in the previous layer (called left and right parent, p_L and p_R), two adjacent tiles in the same layer (left and right sibling, s_L and s_R), and the remaining tiles in the next layer (children). By connecting every tile to its (right) parent, we obtain an infinite tree structure. The numbers 0, 1, 2 denote the *type* of the tile, which is the number of parents. We can implement the function $Adj(t, i)$ returning the pointer to the i-th neighbor to tile t, clockwise starting from the parent in amortized time $O(1)$ by using a lazily generated representation of the tessellation, where each tile is represented by a node holding pointers to the parent node, children nodes (if already generated), and siblings (if already known). Since every tile has at least two (non-rightmost) children, this structure grows exponentially. To gain intuition about hyperbolic tessellations, we recommend playing HyperRogue [15] as its gameplay focuses on the crucial concepts of exponential growth and distances in the tessellation graph.

In a Euclidean tessellation, the distance x between the center of two adjacent tiles can be arbitrary; the same tessellation can be as coarse or fine as needed. This is not the case for hyperbolic tessellations. In the case of G_{710}, x must be the edge length of a triangle with angles $\pi/7$, $\pi/7$, and $2\pi/3$, which we can find using the hyperbolic cosine rule. If the central tile is at Minkowski hyperboloid coordinates $(0, 0, 1)$, the coordinates of every other tile t can be found by composing translations (by x units) and rotations (by multiples of $2\pi/7$).

3 Our contribution

Here we introduce the discrete hyperbolic random graph model (DHRG), which is the discrete version of the HRG model (Definition 1). We map vertices $v \in V$ not to points in the continuous hyperbolic plane but the tiles of our tessellation, i.e., $\mu : V \to D_R$, where D_R is the set of all tiles in distance at most R.

▶ **Definition 2.** *A* **discrete hyperbolic random graph (DHRG)** *with parameters n, R, T, and α is a random graph $H = (V, E)$ constructed as follows:*

- *The set of vertices is $V = \{1, \ldots, n\}$,*
- *Every vertex $v \in V$ is independently randomly assigned a tile $\mu(v) \in D_R$, in such a way that the probability that $\mu(v) = w \in R_d$ is proportional to $\frac{e^{d\alpha}}{|R_d|}$;*
- *Every pair of vertices $v_1, v_2 \in V$ are independently connected with an edge with probability $p(\delta(\mu(v_1), \mu(v_2)))$, where $p(d) = \frac{1}{1+e^{(d-R)/2T}}$.*

Note that the definition permits $\mu(v_1) = \mu(v_2)$ for two different vertices $v_1, v_2 \in V$. This is not a problem; such vertices v_1 and v_2 are not necessarily connected, nor do they need to have equal sets of neighbors. This may happen when two vertices are too similar to be differentiated by our model.

Determining the relation between discrete and continuous distances theoretically is challenging. We find this relation experimentally. We compute the hyperbolic distance R_d between $h_0 = (0, 0, 1)$ and T_d, where T_d is a randomly chosen tile such that $\delta_0(T_d) = d$. We get

$R_d = c_1 d + c_2 + X_d$, where $c_1 \approx 0.9696687$, $c_2 \approx 0.0863634$, and X_d is a random variable with a bell-shaped distribution, expected value $EX_d = o(1)$ and variance Var $X_d = \Theta(d)$. (See Appendix A for the example detailed results for G_{710}.) Note that the error is much better than for Euclidean hexagonal grid, where a similar formula holds, but with Var $X_d = \Theta(d^2)$. This is because in the Euclidean plane, the stretch factor depends on the angle between the line $[h_0, T_d]$ and the grid, which remains constant along the whole line; on the other hand, in the hyperbolic plane, this angle is not constant, and its values in sufficiently distant fragments of that line are almost independent. Let n_d be the number of tiles in distance d; they are every second Fibonacci number multiplied by 7, and thus $n_d = \Theta(\gamma^d)$ for $\gamma = \frac{3+\sqrt{5}}{2}$. The radius of a hyperbolic disk which has the same area as the union of all tiles in distance at most d is $\log(\gamma)d + O(1)$; our coefficient c_1 is close to $\log(\gamma) \approx 0.9624237$, but slightly larger.

Let j be a function which maps every tile of our tessellation to the coordinate of its center. DHRG mappings can be converted to HRG by composing μ with j, and the other conversion can be done by finding the tile containing $\mu(v)$ for each $v \in V$. Since our experimental results show that the discrete distances are very good approximations of the continuous distances (up to the multiplicative constant c_1 and additive constant c_2), we expect the desired properties of HRGs, such as the high clustering coefficient and the power-law degree distribution, to still be true in DHRGs. The parameters α, R and T of the DHRG model will be obtained from the HRG parameters by dividing them by c_1.

4 Algorithms for DHRG

In this section, we present our algorithms for working with the DHRG model. Our algorithms will be efficient under the assumption $R = O(\log n)$ and $m = o(n^2/R)$.

▶ **Proposition 3.** *There is a canonical shortest path between every pair of tiles (t_1, t_2). If t_2 is to the right from t_1, this canonical shortest path consists of: a number of right parent edges; at most one right sibling edge; and a number of non-leftmost child edges. If t_1 is to the right from t_2, the canonical path is defined symmetrically. For any pair of tiles (t_1, t_2), the distance $\delta(t_1, t_2)$ can be found in time $O(\delta(t_1, t_2))$.*

The algorithm for finding $\delta(t_1, t_2)$ works as follows: for every d starting from $\min(\delta_0(t_1), \delta_0(t_2))$ and going downwards, we find the leftmost and rightmost ancestors of t_1 and t_2 in R_d. If one of the ancestors of t_1 matches one of the ancestors of t_2, we return $\delta_0(t_1) + \delta_0(t_2) - 2d$; if they do not match but are adjacent, we return $\delta_0(t_1) + \delta_0(t_2) - 2d + 1$.

In our application, we will need to efficiently find the distance between a tile t and all tiles $u \in A$. We will do it using the following data structure:

▶ **Definition 4.** *A* distance tally counter *is a structure with the following operations:*

- INIT*, which initializes the multiset of tiles A to empty.*
- ADD*(u,x), which adds the tile u to the multiset A with multiplicity x (which can be negative).*
- COUNT*(t), which, for tile t, returns an array T such that $T[d]$ is the number of elements of A in distance d from t.*

▶ **Theorem 5.** *There is an implementation of distance tally counter where all the operations are executed in $O(R^2)$, where R is the maximum distance from the central tile.*

Proof (sketch). A **segment** is a pair of tiles of form $[p_L^k(t), p_R^k(t)]$ for some tile t and $k \geq 0$. The notation p_R^k here denotes the k-th iteration, i.e., the rightmost k-th ancestor in this case. For a segment $s = [v_L, v_R]$, let $p(s) = [p_L(v_L), p_R(v_R)]$ be the parent segment. In the case of G_{710}, either $p_L^k(t) = p_R^k(t)$, or they are neighbors. The algorithm from Proposition 3 can be seen as follows: we start with two segments $[t_1, t_1]$ and $[t_2, t_2]$, and then apply the parent segment operation to each of them until we obtain segments which are close. To construct an efficient distance tally counter, we need to tally the ancestor segments for every $u \in A$.

For every segment s, we keep an array a_s, where $a_s[d]$ is the number of $u \in A$ such that $p^d[u, u] = s$. The operation Add updates these arrays in time $O(R^2)$. The operation Count(t) constructs $s_d = p^d[t, t]$ for $d = 0, \ldots, \delta_0(t)$, and uses the information in segments intersecting or adjacent to s_d to count the number of elements of u for which the distance algorithm would return every possible distance. We need to make sure that we do not count the same $u \in A$ twice (for different values of d). However, this can be done by temporarily subtracting from $a_s[d]$ entries which correspond to u's which have been already counted; see Appendix B for details. ◀

Other tessellations than the order 3 heptagonal tessellation are possible. The order-3 octagonal grid, G_{810}, is coarser. Finer tessellations can be obtained by applying the Goldberg-Coxeter construction, such as G_{711} from Figure 1; their growth is less extreme than for G_{710}, and thus the distance between segment ends, as well as the number of sibling edges in the canonical path, may be greater than 1. However, the algorithms generalize; see Appendix B.

Computing the likelihood. Computing the log-likelihood in the continuous model is difficult, because we need to compute the sum over $O(n^2)$ pairs; a better algorithm was crucial for efficient embedding of large real-world scale-free networks [4]. The algorithms above allow us to compute it quite easily and efficiently in the DHRG model. To compute the log-likelihood of our embedding of a network H with n vertices and m edges, such that $\delta_0(v) \leq R$ for each $v \in V$, we:

- for each d, compute Pairs$[d]$, which is the number of pairs (v, w) such that $\delta(v, w) = d$. The distance tally counter allows doing this in a straightforward way (Add$(\mu(v),1)$ for each $v \in V$ followed by Count$(\mu(v))$ for each $v \in V$), in time $O(nR^2)$.
- for each d, compute Edges$[d]$, which is the number of pairs (v, w) connected by an edge such that $\delta(v, w) = d$. This can be done in time $O(mR)$ simply by using the distance algorithm for each of m edges.

After computing these two values for each d, computing the log-likelihood is straightforward. One of the advantages over [4] is that we can then easily compute the log-likelihood obtained from other values of R and T, or from a function $p(d)$ which is not necessarily logistic.

Improving the embedding. A continuous embedding of good quality can be obtained by first finding an approximate embedding and then improving it using a *spring embedder* [14]. Imagine there are attractive forces between connected pairs of vertices, and repulsive forces between unconnected pairs. The embedding m changes in time as the forces push the vertices towards locations in such a way that the quality of the embedding is improved. Computationally, spring embedders are very expensive – there are $\Theta(n^2)$ forces, and potentially many steps of simulation could be necessary.

On the contrary, our approach allows to improve DHRG embeddings easily. We use a local search algorithm. Suppose we have computed the log-likelihood and on the way we have computed the vectors Pairs and Edges, as well as the distance tally counter where

every $\mu(v)$ has been added. Let $v' \in V$ be a vertex of our embedding, and w be a tile. Let μ' be the new embedding given by $\mu'(v') = w$ and $\mu'(v) = \mu(v)$ for $v \neq v'$. Our auxiliary data lets us compute the log-likelihood of μ' in time $O(R^2 + R\deg(w))$.

We try to improve the embedding in the following way: in each iteration, for each vertex $v \in V$, consider all neighbors of $\mu(v)$, compute the log-likelihood for all of them, and if for some μ' we have $\log L(\mu') > \log L(\mu)$, replace μ with μ'. Each iteration takes time $O(R^2 n + Rm)$.

5 Experimental setup

The setup of the experiments is as follows. First, we map a network to the hyperbolic plane using the hyperbolic embedder based on the algorithm from [4] (for brevity, we will call it BFKL). This is the embedding stage. We start with a default parameters for the embedder (R_0, T_0, α_0). This way we obtain the placement of the nodes in the hyperbolic plane, on which we estimate HRG predictive model with the same R_0, T_0, α_0 as the embedder (prediction stage). We compute the log-likelihood (L_1). Usually, L_1 is even worse than the log-likelihood of the naïve Erdös-Rényi-Gilbert model where each edge exists with probability $m/\binom{n}{2}$. This is because the influence of the parameter T on the quality of the embedding is small [20] and BFKL uses a small value of $T = 0.1$ that does not necessarily correspond to the network. The prediction stage is irreplaceable here – the log-likelihood is computed for the predictive model that takes the placement of the nodes as given. That is why, we should be still able to obtain a better log-likelihood during the prediction stage by estimating HRG models with different parameters (R_1, T_1); for brevity, the log-likelihoods obtained via such an optimization will be called "the best log-likelihoods". L_2 is the best log-likelihood obtained with the default embedding. As it proxies the best possible outcome for the default embedding, it will serve as the benchmark scenario in our experiment.

Now we need to check if using DHRG improves the quality of the embedding. To this end, we convert our embedding into the DHRG model, by finding the nearest tile of our tessellation for each $v \in V$. L_3 is the best discrete log-likelihood for a new embedding (computed with the logistic function). We call this phase *discretization*.

Next, we try our *local search* algorithm (20 iterations) and compute the best discrete log-likelihood after local search (L_5).[1] Finally, we proceed to the *de-discretization* phase, i.e., convert our mapping back to the HRG model and find the best log-likelihood L_7.

We denote the running times as t_m (converting HRG to DHRG), t_l (computing Pairs and Edges), t_e (local search). For comparison, we also include the time of computing the best continuous log-likelihood t_c using a parallelized $O(n^2)$ algorithm[2], and the time of computing the log-likelihood by the BFKL algorithm (t_b). All the times are measured on Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz with 96 GB RAM. Most computations use a single core, except the continuous log-likelihood values (L_2 and L_7) which use 8 cores. Our implementation, using the RogueViz non-Euclidean geometry engine [16], as well as the results of our experiments can be found at `https://figshare.com/articles/software/Discrete_Hyperbolic_Random_Graph_Model_code_and_data_/16624369`.

[1] We do 20 iterations because further iterations tend to move less and less vertices, and thus their effect on the quality of embedding is minimal.

[2] We compute the distance for every pair of nodes. This way we create the array d, where $d[n]$ is the number of distances in the interval $[n\varepsilon, (n+1)\varepsilon)$; we take $\varepsilon = 10^{-4}$. This computation is parallelized. The array d allows computing a very good approximation of the log-likelihood for any value of parameters R and T in time $O(R/\varepsilon)$.)

Table 1 Experimental results on real-world networks. Facebook (Fb), Slashdot (Sd), Amazon (Am), Google (Go), and Patents (Pa) networks from SNAP database; F09 and F11 are GitHub networks. MB is the amount of memory in megabytes, and time is in seconds.

name	n	m	R	α	grid	$-L_2$	$\frac{L_3}{L_2}$	$\frac{L_5}{L_3}$	$\frac{L_7}{L_2}$	MB	t_m [s]	t_l [s]	t_e [s]	t_c [s]	t_b [s]
Fb	4309	88234	12.57	0.755	G_{710}	176131	1.04	0.93	0.97	40	0.196	0.03	10	0.35	0.048
Fb	4309	88234	12.57	0.755	G_{810}	176131	1.07	0.92	0.98	54	0.183	0.03	8	0.5	0.048
F09	74946	537972	20.90	0.855	G_{710}	3954627	1.04	0.86	0.90	2010	5.432	1.16	222	131	0.896
F09	74946	537972	20.90	0.855	G_{810}	3954627	1.06	0.84	0.90	1866	4.634	0.81	176	128	0.896
Sd	77352	327431	26.00	0.610	G_{710}	2091651	1.25	0.72	0.92	2659	5.326	1.05	201	130	0.292
Sd	77352	327431	26.00	0.610	G_{810}	2091651	1.27	0.71	0.92	2253	4.618	0.78	158	126	0.292
Am	334863	925872	24.11	0.995	G_{710}	6957174	1.04	0.86	0.91	5677	23.34	5.40	721	2690	1.444
Am	334863	925872	24.11	0.995	G_{810}	6957174	1.04	0.85	0.90	4868	19.76	3.92	576	2811	1.444
F11	405270	2345813	26.34	0.715	G_{710}	20028756	1.22	0.76	0.93	9995	30.36	7.36	1349	3715	5.216
F11	405270	2345813	26.34	0.715	G_{810}	20028756	1.22	0.76	0.93	8940	25.84	5.38	1113	3636	5.216
Go	855804	4291354	26.06	0.865	G_{710}	22762281	1.30	0.75	0.98	18226	64.75	16.05	2363	16618	3.560
Go	855804	4291354	26.06	0.865	G_{810}	22762281	1.32	0.75	0.99	15314	54.31	10.93	1823	15818	3.560
Pa	3764118	16511741	28.74	0.995	G_{810}	—	—	0.90	—	66396	250.6	73.65	9335	—	41.24

6 Experiments on real-world networks

Table 1 contains detailed results of the experiments. The networks we use come mostly from SNAP database [18]. To benchmark our algorithm on a large network, we additionally study undirected social networks with power-law-like scale behavior, representing the following relations that occured between 2009 and 2011 in GitHub [9] (see Appendix C for the details). It makes sense to use finer tessellations for smaller graphs and coarser tessellations for larger ones. In most cases, we conduct our experiments on two tessellations: G_{710} (order-3 heptagonal) and the coarser G_{810} (order-3 octagonal). For the Facebook graphs, we also try finer tessellations; see Appendix D for the details. Finer tessellations give better log-likelihoods, but a too dense grid dramatically decreases the performance without giving significant benefits.

The parameters n, m, R and α come from the BFKL embedder (n is the number of vertices, m is the number of edges). Discretization worsens the log-likelihoods; for smaller networks L_3 are usually slightly worse than L_2, but this is not surprising. First, our edge predictor has lost some precision in the input because of our tesselation's discrete nature. Second, the original prediction was based on the hyperbolic distance r while our prediction is based on the tesselation distance d, and the ratio of r and d depends on the direction. The whole procedure improves the log-likelihood by up to 10%. For the patents network, computing L_2 and L_7 was not feasible; L_3 was 208618134.

The current version uses a significant amount of RAM. It should be possible to improve this by better memory management (currently vertices and segments which are no longer used or just temporarily created are not freed), or possibly path compression.

When it comes to the running time, computing Edges and Pairs takes negligible time (t_l) when compared to the network size. Even for as large network as Patents, those operations took slightly over a minute (Table 1); for most of the networks analyzed, they took a few seconds. Converting time t_m increases with the size of the network, for most of the networks analyzed we need less than a minute. In comparison to computing log-likelihood as performed by BFKL, our solution is comparable in time; they are of the same order of magnitude. The

longest time is needed for local search. Not surprisingly, the larger graph, the longer it takes to find improvements. However, their spring embedder working in quadratic time is much slower than our local search.[3]

7 Experiments on simulated graphs

7.1 Log-likelihood

Our experiments on real-world graphs showed that discretization, local search, and de-discretization improves the quality of the embedding in terms of log-likelihood. In this section we conduct simulations to see whether we can generalize those observations.

We use the generator included with [4] to generate HRGs with the following parameters: varying n, $\alpha = 0.75$, $T = 0.1$, $R = 2\log(n) - 1$. These are the default values of parameters used by this generator. For every value of n considered, we generate 1000 graphs. For each of the generated graphs H, we embed H using BFKL, compute L_2, convert to DHRG on tessellation G_{710}, improve the embedding (up to 20 iterations), convert back to HRG, and compute L_7. We also compute L_g, which is the log-likelihood of the originally generated embedding (groundtruth).

Figure 2 shows the density graph of L_2/L_g and L_7/L_g for every n. Since L_g is negative, larger values of these ratios are worse. We know that an optimal embedder should achieve log-likelihood at least as good as L_g. We find that our algorithm yields a significant improvement.

Table 2 Changes in log-likelihood after applying our procedure to BFKL embedders. Percent of improvements signifies the percent of the cases the log-likelihood increased (improved). Average and median improvements computed conditionally on improvement. In the "rel. to 0" columns, we present the values of $100 \cdot (1 - L_7/L_2)$; in the "groundtruth" column, we present the values of $100 \cdot (1 - (L_g - L_7)/(L_g - L_2))$.

n	improved	rel. to 0		groundtruth	
		avg	median	avg	median
100	84.6	12.8	11.9	36.2	22.0
200	95.2	12.5	12.1	20.6	20.4
300	97.2	11.8	10.6	18.6	17.5
500	99.1	11.5	11.1	18.0	17.5
1000	99.6	11.4	11.0	17.2	16.8
2000	100.0	11.0	10.7	16.2	15.9
3000	100.0	10.9	10.7	15.9	15.8
4000	100.0	10.6	10.4	15.4	15.2
5000	100.0	10.5	10.3	15.0	14.7
10000	100.0	10.3	10.2	14.8	14.7
15000	100.0	10.0	9.9	14.5	14.4

[3] We have also ran the BFKL spring embedder on the Facebook graph for $T = 0.54336$ and seed 123456789, reporting the log-likelihood of -131634, better than ours. However, this result appears incorrect; our implementation reports the original log-likelihood of $L_1 = -211454$, and can improve it to $L_7 = -157026$. Computing the log-likelihood incorrectly may negatively impact the quality of BFKL embedding.

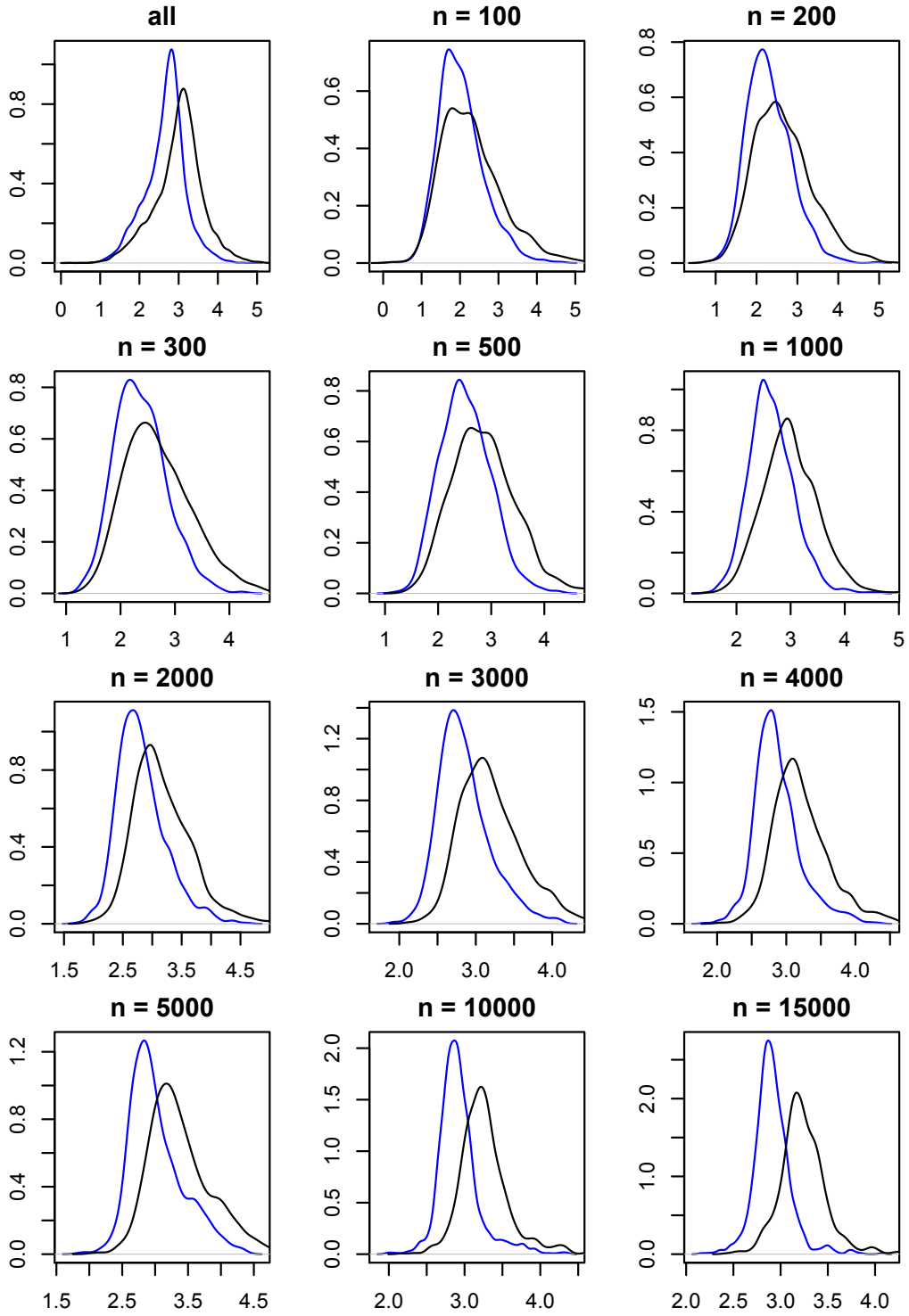


Figure 2 Density of L_2/L_g (black) and L_7/L_g (blue) for $T = 0.1$.

According to the data in Table 2, we notice that our procedure leads to better embeddings than the pure BFKL embedder no matter the size of the graph. Our procedure yields log-likelihoods that are closer to the log-likelihood of the groundtruth. The improvement towards groundtruth is not stable; with the increase of the graph the BFKL embeddings converge, so our improvements become less prominent (around 15% for large graphs). However, the improvements are statistically significant (p-values are always 0.00 for paired Wilcoxon test with alternative hypothesis that the values of log-likelihood increased after our procedure).

In real-life cases, hardly do we know the groundtruth; comparison of log-likelihoods for BFKL embedder and our procedure resembles what would we do with real data (columns rel. to zero in Table 2). In such a case, we may expect that our procedure should improve the result by average by 10%. This result seems stable no matter the size of the graph.

7.2 Greedy routing

One potential application of hyperbolic embedding is greedy routing [5, 2]. A node v obtains a packet to node w; if w is not directly connected to v, v needs to select one of its neighbors through which the packet will be forwarded. In the greedy routing approach, we use the embedding to select the connected node which is the closest to the goal w. Greedy routing fails if, at some point in the chain, none of the connected nodes is closer to w than v itself. In [5] a hyperbolic embedding of the Internet is constructed, yielding 97% success rate of greedy routing. This is much better than a similar algorithm based on geographical placement of nodes (14%). High success rate is also robust with respect to link removals [5]. While the MLE method of finding embedding yields worse results than embeddings constructed specifically for the purpose of greedy routing [2], it is interesting to see how good our methods are according to this metric.

Table 3 summarizes the changes in success rates after our procedure. We discuss the conditional changes (improvement on the condition of the improvement or deterioration on the condition of the deterioration), because we find the absolute changes possibly misleading for small networks. Percent of changes proxies us the probability of the effect. If the effect occurs, we know what to expect without the bias of the counter-effect. The success rates of the original embedding are around 93% on average. Discretization usually worsens the success probability. This appears to be caused by the fact that two neighbors of v can be in the same distance to w (because of discretization), while originally the more useful node is closer. We notice that if the deterioration due to discretization occurs, the average percentage deterioration decreases with the increasing size of the graph. This is expected, since the distances are larger in larger graphs. Meanwhile, the median deterioration (if the deterioration occurs) is stable at around 3%. This means that with the increasing size of the network we face serious deteriorations after discretization less often. The effect is statistically significant (p-values of Wilcoxon paired tests with the alternative hypotheses that the success rate is lower after the discretization are always 0.00). The whole procedure improves the results in more than 45% of cases, however the change for bigger graphs is not statistically significant (p-values for paired Wilcoxon tests with the alternative hypotheses that the success rates increased after our procedure are greater than 10%). For large graphs (over 4000 vertices) in about half of the cases the success rate after the procedure is not worse than the original one.

To reduce the negative effect of discretization, we also perform the same experiment using the G_{711} grid. The results are shown in Table 3. Contrary to the G_{710} tessellation, usage of the finer tessellation for routing (when all three steps are performed) improves the success rate, and the effect is statistically significant. The general directions of the effects resemble the case of coarser grid. Discretization leads to a statistically significant decrease in the success rate; the bigger graphs, the less frequent a noticeable deterioration.

Table 3 Changes in the success rate of greedy routing. Average and median computed conditionally on change (improvement on the condition of the improvement or deterioration on the condition of the deterioration). P-values for Wilcoxon paired tests.

T tiling	n	deterioration (discretization) %	med	avg	improvement (local search) %	med	avg	improvement (three steps) %	med	avg	p-value
	100	95.3	3.28	5.14	81.1	2.55	3.71	56.4	1.11	1.81	0.001
	200	97.3	3.27	4.70	82.5	2.24	3.10	53.8	0.99	1.35	0.037
	300	99.0	3.25	4.37	85.7	1.97	2.51	55.3	0.91	1.21	0.000
	500	99.7	3.19	4.34	87.9	1.81	2.38	55.7	0.71	1.00	0.005
	1000	100.0	3.25	3.92	90.8	1.66	1.99	54.0	0.51	0.70	0.071
T=0.1	2000	100.0	3.27	3.73	94.3	1.58	1.73	51.6	0.40	0.50	0.806
G_{710}	3000	100.0	3.27	3.61	98.6	1.53	1.68	49.7	0.33	0.40	0.999
	4000	100.0	3.33	3.52	98.6	1.55	1.63	49.0	0.29	0.37	0.999
	5000	100.0	3.31	3.52	98.4	1.52	1.61	46.9	0.27	0.36	0.999
	10000	100.0	3.27	3.37	99.8	1.56	1.60	42.1	0.19	0.23	1.000
	15000	100.0	3.37	3.39	99.9	1.61	1.62	32.2	0.16	0.19	1.000
	100	86.4	1.65	2.76	72.5	1.67	2.59	61.0	1.00	1.68	0.00
	200	90.8	1.67	2.49	78.9	1.61	2.25	63.1	0.94	1.42	0.00
	300	94.5	1.59	2.30	82.0	1.42	1.97	65.9	0.84	1.19	0.00
	500	98.2	1.57	2.16	86.8	1.35	1.69	67.6	0.80	1.04	0.00
	1000	99.4	1.65	1.96	93.9	1.23	1.42	70.4	0.59	0.73	0.00
T=0.1	2000	99.8	1.61	1.82	94.3	1.16	1.27	72.6	0.48	0.60	0.00
G_{711}	3000	99.8	1.64	1.75	96.4	1.15	1.21	74.2	0.42	0.49	0.00
	4000	100.0	1.65	1.76	98.1	1.11	1.19	74.0	0.40	0.45	0.00
	5000	100.0	1.63	1.75	98.3	1.15	1.20	78.2	0.40	0.46	0.00
	10000	100.0	1.61	1.67	99.6	1.17	1.19	82.5	0.36	0.38	0.00
	15000	100.0	1.64	1.66	99.7	1.19	1.20	87.0	0.32	0.33	0.00
	300	99.9	3.88	4.53	88.5	2.27	2.72	60.7	1.04	1.37	0.00
	500	100.0	3.60	4.26	88.6	1.74	2.15	59.4	0.84	1.11	0.00
	1000	100.0	3.39	3.73	88.1	1.25	1.48	55.9	0.65	0.78	0.00
T=0.7	2000	100.0	3.21	3.47	87.7	0.99	1.15	52.9	0.46	0.59	0.02
G_{710}	3000	100.0	2.87	3.00	84.4	0.72	0.84	46.7	0.30	0.42	0.98
	4000	100.0	3.22	3.31	89.7	0.80	0.88	38.3	0.28	0.37	1.00
	5000	100.0	2.98	3.02	88.6	0.63	0.69	36.1	0.19	0.24	1.00
	10000	100.0	2.95	2.96	90.6	0.55	0.60	29.1	0.13	0.22	1.00
	15000	100.0	3.23	3.23	95.5	0.73	0.75	24.4	0.12	0.16	1.00
	300	98.3	1.86	2.25	81.1	1.56	1.90	63.5	1.00	1.32	0.00
	500	99.5	1.81	2.14	85.9	1.30	1.57	68.1	0.94	1.12	0.00
	1000	99.9	1.65	1.86	84.0	0.97	1.11	67.5	0.62	0.75	0.00
T=0.7	2000	100.0	1.61	1.71	87.1	0.75	0.83	67.3	0.44	0.55	0.00
G_{711}	3000	100.0	1.40	1.48	83.4	0.53	0.61	64.1	0.32	0.21	0.00
	4000	100.0	1.61	1.64	90.5	0.56	0.62	61.7	0.28	0.36	0.00
	5000	100.0	1.48	1.50	88.8	0.45	0.49	59.5	0.25	0.28	0.00
	10000	100.0	1.47	1.48	94.1	0.41	0.46	61.0	0.18	0.22	0.00
	15000	100.0	1.61	1.61	98.4	0.53	0.53	65.1	0.17	0.19	0.00

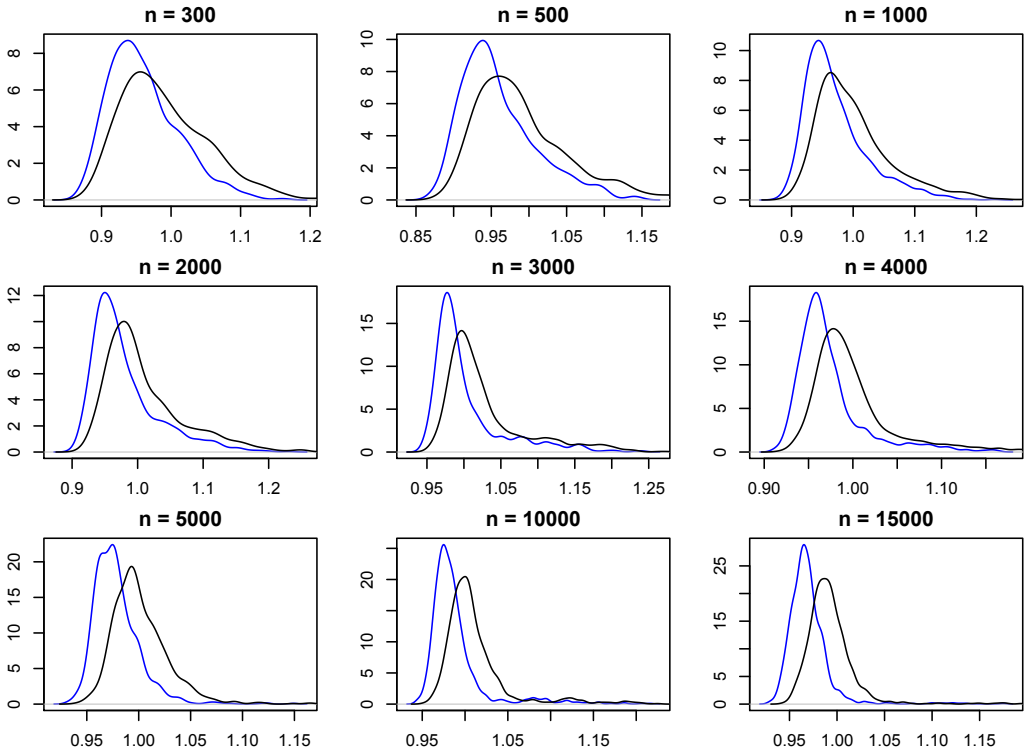


Figure 3 Density of L_2/L_g (black) and L_7/L_g (blue) for $T = 0.7$.

7.3 Changing the temperature

Real-world graphs are considered to have larger temperature T than 0.1. We have also experimented with changing the temperature T to 0.7. This value of T has been used for mapping the Internet [5].

For $T = 0.7$, the embedder and local search actually tend to achieve a better result than the groundtruth. For all sizes of graphs, we observe about 3% improvement in the absolute value of log-likelihood on average $(1 - L_2/L_7)$ (Figure 3). This effect is statistically significant (p-values for paired Wilcoxon tests with alternative hypotheses that the values of log-likelihood increased after our procedure always 0.00).

One could argue that the temperature plays critical role for the success of the greedy routing. With a higher temperature, the links are less predictable, and thus we can expect a lower success rate. We conducted the simulations to check if the insights change in the case of the value of temperature more typical to real-world networks. Table 3 contains the results of the experiments with respect to the size of the graph.

Our insights driven from simulations for $T = 0.7$ resemble the conclusions for the case of $T = 0.1$. In the case of the coarse tessellation (Table 3) the possibility of improvement depends on n. For small and medium-sized graphs the effect is statistically significant; the larger the graph, the less probable statistically significant improvement (p-values of Wilcoxon paired test greater that any conventional significance levels). With the finer tessellation (Table 3), no matter the size of the graph, the post-procedure improvement is statistically significant.

We also checked if the change of the tessellation significantly improves the success rate. To this end, we perfomed paired Wilcoxon tests with the alternative hypotheses that the success rate increased after using a finer tessellation. In all the cases, the p-values were 0.00, so the effect of the tessellation is statistically significant.

8 Conclusion

We introduced the discrete version of the HRG model, which allows efficient algorithms while avoiding numerical issues. We also presented the result of the experimental evaluation of this model. We analyzed both the real-world networks and 20,000 artifical ones, paying special attention to the possible application of the model in greedy routing. Our experimental evaluation shows that we achieve a good approximation of log-likelihood in the HRG model and that using local search significantly improves its log-likelihood, even when converting back to HRG. This is visible both in real-world and in simulated networks. A similar procedure also slightly improves the success rate of greedy routing when a sufficiently fine tessellation is used. The choice of the tessellation seems to be crucial for the success rate for all tested values of the parameters.

References

1 Albert-László Barábasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. doi:10.1126/science.286.5439.509.

2 Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, and Anton Krohmer. Hyperbolic embeddings for near-optimal greedy routing. In *Algorithm Engineering and Experiments (ALENEX)*, pages 199–208, 2018.

3 Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. Efficiently generating geometric inhomogeneous and hyperbolic random graphs. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 21:1–21:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.21.

4 Thomas Bläsius, Tobias Friedrich, Anton Krohmer, and Sören Laue. Efficient embedding of scale-free graphs in the hyperbolic plane. In *European Symposium on Algorithms (ESA)*, pages 16:1–16:18, 2016.

5 Marián Boguñá, Fragkiskos Papadopoulos, and Dmitri Krioukov. Sustaining the internet with hyperbolic mapping. *Nature Communications*, 1(6):1–8, September 2010. doi:10.1038/ncomms1063.

6 Karl Bringmann, Ralph Keusch, and Johannes Lengler. Geometric inhomogeneous random graphs. *Theoretical Computer Science*, 2018. doi:10.1016/j.tcs.2018.08.014.

7 Károly Böröczky. Gömbkitöltések állandó görbületű terekben I. *Matematikai Lapok*, 25:265–306, 1974.

8 James W. Cannon, William J. Floyd, Richard Kenyon, and Walter R. Parry. Hyperbolic geometry. In *In Flavors of geometry*, pages 59–115. University Press, 1997. Available online at http://www.msri.org/communications/books/Book31/files/cannon.pdf.

9 Dorota Celińska. Information and influence in social network of Open Source community. In *9th Annual Conference of the EuroMed Academy of Business*, 2016.

10 Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. *Journal of Parallel and Distributed Computing*, 131:200–217, 2019. doi:10.1016/j.jpdc.2019.03.011.

11 Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=2487085.2487132.

12 Ilya Grigorik. Github Archive. `https://www.githubarchive.org/`, 2012.

13 Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. Random hyperbolic graphs: Degree sequence and clustering. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming*, pages 573–585, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

14 Stephen G. Kobourov and Kevin Wampler. *Non-euclidean spring embedders*, pages 207–214. IEEE Computer Society, 2004. `doi:10.1109/INFVIS.2004.49`.

15 Eryk Kopczyński, Dorota Celińska, and Marek Čtrnáct. HyperRogue: Playing with hyperbolic geometry. In *Proceedings of Bridges : Mathematics, Art, Music, Architecture, Education, Culture*, pages 9–16, Phoenix, Arizona, 2017. Tessellations Publishing.

16 Eryk Kopczyński and Dorota Celińska-Kopczyńska. RogueViz: non-Euclidean geometry engine for visualizations, games, math art, and research, October 2021. URL: `https://github.com/zenorogue/hyperrogue/`.

17 John Lamping, Ramana Rao, and Peter Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 401–408, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co. `doi:10.1145/223904.223956`.

18 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

19 Tamara Munzner. Exploring large graphs in 3d hyperbolic space. *IEEE Computer Graphics and Applications*, 18(4):18–23, 1998. `doi:10.1109/38.689657`.

20 Fragkiskos Papadopoulos, Rodrigo Aldecoa, and Dmitri Krioukov. Network geometry inference using common neighbors. *Phys. Rev. E*, 92:022807, August 2015. `doi:10.1103/PhysRevE.92.022807`.

21 Fragkiskos Papadopoulos, Maksim Kitsak, M. Angeles Serrano, Marian Boguñá, and Dmitri Krioukov. Popularity versus Similarity in Growing Networks. *Nature*, 489:537–540, September 2012.

22 Manuel Penschuck. Generating practical random hyperbolic graphs in near-linear time and with sub-linear memory. In *SEA*, 2017.

23 Frederic Sala, Chris De Sa, Albert Gu, and Christopher Re. Representation tradeoffs for hyperbolic embeddings. In *Proc. ICML*, pages 4460–4469, Stockholmsmässan, Stockholm Sweden, 2018. PMLR. URL: `http://proceedings.mlr.press/v80/sala18a.html`.

24 Zeynab Samei and Mahdi Jalili. Application of hyperbolic geometry in link prediction of multiplex networks. *Scientific Reports*, 9(1):12604, August 2019. `doi:10.1038/s41598-019-49001-7`.

25 Moritz von Looz. *High-Performance Graph Algorithms*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2019. `doi:10.5445/IR/1000095908`.

26 Moritz von Looz, Henning Meyerhenke, and Roman Prutkin. *Generating Random Hyperbolic Graphs in Subquadratic Time*, pages 467–478. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. `doi:10.1007/978-3-662-48971-0_40`.

27 Moritz von Looz, Mustafa Safa Ozdayi, S. Laue, and Henning Meyerhenke. Generating massive complex networks with hyperbolic geometry faster in practice. *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2016.

A Tessellation distances versus hyperbolic distances

Table 4 Tessellation distances (d) versus expected hyperbolic distances (ER_d): expected value and variance.

d	n_d	ER	Var(R)
0	1	0.00000000	0.00000000
1	7	1.09054966	0.00000000
2	21	2.02973974	0.02467308
3	56	2.99594181	0.03923368
4	147	3.96512066	0.05370197
5	385	4.93471877	0.06823850
6	1008	5.90437726	0.08279668
7	2639	6.87404448	0.09735968
8	6909	7.84371297	0.11192362
9	18088	8.81338165	0.12648772
10	47355	9.78305035	0.14105185
11	123977	10.75271905	0.15561599
12	324576	11.72238775	0.17018013
13	849751	12.69205646	0.18474427
14	2224677	13.66172516	0.19930841
15	5824280	14.63139386	0.21387255
16	15248163	15.60106257	0.22843669

Table 4 contains the example detailed results for G_{710} (see Section 3). The asymptotic values are obtained in the following way: the difference $ER_d - ER_{d-1}$ converges very quickly to 0.9696687, and the difference $ER_d - 0.9696687d$ converges very quickly to 0.0863634.

B Proofs and pseudocodes

Generalization of Proposition 3. For $D(G) > 1$, there is one more type of a canonical path possible, where w is a parent of v, but neither the leftmost nor rightmost one.

The idea of the algorithm is to find the shortest canonical path. Suppose that $\delta_0(v) = d' + \delta_0(w)$, where $d' \geq 0$. For each i starting from 0 we compute the endpoints of the segments $P^{d'+i}(v)$ and $P^i(w)$. We check whether these segments are in distance at most γ on the ring; if no, then we can surely tell that we need to check the next i; if yes, we know that the shortest path can be found on one of the levels from i to $i + \lfloor \gamma/2 \rfloor$. We compute the length of all such paths and return the minimum.

The pseudocode of our algorithm is given below. It uses five integer variables a_i, d_i, d and four tile variables l_i, r_i $(i = 1, 2)$. Variables a_i, d_i, l_i and r_i are modified only by the function push(i), which lets us keep the following invariant: $\delta_0(l_i) = \delta_0(r_i) = d_i$, $l_i = p_L^{a_i}(v_i)$, $r_i = p_R^{a_i}(v_i)$. By $v + k$, where v is a tile, we denote the k-th right sibling of v.

The lines (16-17) deal with the special case for $D(G) > 1$ mentioned above.

The main loop in lines (19-23) deals with the other cases. At all times d is the currently found upper bound on $\delta(v, w)$. It is easy to check that the specific shortest path given in Proposition 3 will be found by our algorithm.

Every iteration of every loop increases a_1 or a_2, and an iteration can occur only if $a_1 + a_2 < \delta(v, w)$. Therefore, the algorithm runs in time $O(\delta(v, w))$.

```
1.  function DISTANCE(v_1, v_2):
2.      for i ∈ {1, 2}:
3.          l_i := v_i
4.          r_i := v_i
5.          d_i := δ_0(v_i)
6.          a_i := 0
7.      function push(i):
8.          a_i := a_i + 1
9.          d_i := d_i − 1
10.         l_i := p_L(l_i)
11.         r_i := p_R(r_i)
12.     while d_1 > d_2 :
13.         push(1)
14.     while d_2 > d_1 :
15.         push(2)
16.     for i ∈ {1, 2} if v_i ∈ [l_i, r_i] :
17.             return a_{3−i}
18.     d := ∞
19.     while a_1 + a_2 < d:
20.         for i ∈ {1, 2} for k ∈ {0, ..., D(G)} if l_i = r_{3−i} + k :
21.             d := min(d, a_1 + a_2 + k)
22.         push(1)
23.         push(2)
24.     return d
```

◀

Proof of Theorem 5 for the general case. A segment is **good** if it is of the form $P^d([v, v])$ for some $v \in V$ and $d \in \mathbb{N}$. In our algorithm the operation ADD(w) will update the information in the good segments of the form $P^d([w, w])$, and the operation COUNT(v) will follow the algorithm from Proposition 3, but instead of considering the single segment $[w, w]$, it will count all of them, by using the information stored in the segments close to $P^d([v, v])$. Our algorithm will optimize by representing all the good segments coming from tiles v added to our structure.

We call a tile or good segment *active* if it has been already generated, and thus is represented as an object in memory. For each active tile $v \in V$ we keep two lists $L_L(v), L_R(v)$ of active segments S such that v is respectively the leftmost and rightmost element of S. Each active segment S also has a pointer to $P(S)$, which is also active (and thus, all the ancestors of S are active too), and a dynamic array of integers $a(S)$. Initially, there are no active tiles or good segments; when we activate a segment S, its $a(S)$ is initially filled with zeros. The value of $a(S)[i]$ represents the total $f(w)$ for all tiles w which yield the segment S after i operations of the algorithm from the proof of Proposition 3, i.e., $a(S)[i] = \sum_{w:P^i[w,w]=S} f(w)$.

The operation ADD(w, k) works as follows: for each $i = 0, \ldots, \delta_0(v)$, we simply add k to $a(P^i(S))[i]$. In the pseudocode below, we assume that $P(S)$ returns **null** if S is the root segment.

```
1.  function ADD(w, x):
2.      S := [w, w]
3.      i := 0
4.      while S ≠ null:
5.          a(S)[i] = a(S)[i] + x
6.          S := P(S)
7.          i := i + 1
```

The operation COUNT(v) activates v and $S = [v, v]$ together with all its ancestors. We return the vector A obtained as follows. We look at $P^i(S)$ for $i = 0, \ldots, \delta_0(v)$, and for each $P^i(S)$, we look at close good segments q' on the same level, lists $L_L(w), L_r(w)$ for all w in distance at most γ from $P_i(S)$. The intuition here is as follows: the algorithm from Proposition 3, on reaching $p^{i_1}(v) = S$ and $p^{i_2}(w) = S'$, would find out that these two pairs are close enough and return $i_1 + i_2 + \delta(S, S')$; in our case, for each c such that $a(S')[c] \neq 0$, we will instead add $a(S')[c]$ to $A[a_1 + \delta(S, S') + c]$.

We have to be careful that, if we count some vertex v when considering the pair of segments (S, S'), we do not count it again when considering the pair of segments $(P^j(S), P^j(S'))$. This is done in lines 18–21 in the pseudocode below. By S^L and S^R we respectively denote the leftmost and rightmost vertex of the segment S.

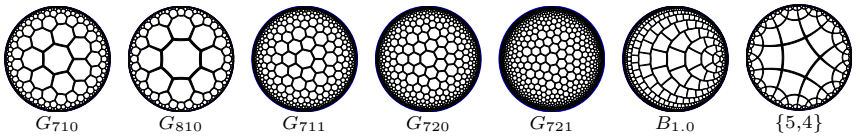
```
1.  function COUNT(v):
2.      U = ∅
3.      for each active S' ∋ v:
4.          insert(U, (S', 0))
5.      d := 0
6.      S := [v, v]
7.      while S ≠ null :
8.          for i ∈ 0, ..., D(G) :
9.              for each S' ∈ L_R(S^R + i):
10.                 insert(U, (S', d + δ(S, S')))
11.             for each S' ∈ L_L(S^L − i):
12.                 insert(U, (S', d + δ(S, S')))
13.         d := d + 1
14.         S := P(S)
15.     T = []
16.     for each (S', d) ∈ U:
17.         for each i: T[d + i] = T[d + i] + a(S')[i]
18.         S'' := P(S')
19.         if (S'', d') ∈ U for some d':
20.             for each i: T[d' + i] = T[d' + i] − a(S')[i]
21.             break
22.     return T                                              ◀
```

C Details of the GitHub dataset

In GitHub convention, *following* means a registered user agreed to be sent notifications about other user's activity within the service. We represent this relationship using the following graph $\mathcal{G}_f$. There is an edge in $\mathcal{G}_f$ between A and B if and only if A follows B. Mechanisms behind the creation of this network involve users' popularity and the similarity, which suggests underlying hyperbolic geometry of $\mathcal{G}_f$. $\mathcal{G}_f$ also shows power-law-like scale behavior [9]; we believe it is a useful benchmark for our analysis. Since the complete download of GitHub data is impossible, our dataset is combined from two sources: GHTorrent project [11] and GitHubArchive project [12]. The analyzed networks contain information about the following relationships in two snapshots: F09 covers relationships that occurred in the service from 2008 to 2009 and F11 covers the same during 2008-2011 period.

Table 5 Experimental results on the Facebook network ($L_2 = -176131$).

grid	L_3	L_5	L_7	MB	#it	t_m [s]	t_l [s]	t_e [s]
G_{810}	-187738	-172018	-172585	46	37	0.180	0.027	14.17
G_{710}	-182721	-170074	-170873	40	29	0.194	0.030	12.13
G_{711}	-179125	-167991	-168445	61	23	0.281	0.058	17.82
G_{720}	-179977	-168105	-168817	98	71	1.025	0.094	91.87
G_{721}	-178108	-167407	-167824	146	99*	1.359	0.208	282.0
G_{753}	-177254	-166889	-167648	1050	99*	4.446	3.059	4999
B_2	-180354	-168055	-168338	47	15	1.278	0.037	17.17
$B_{1.1}$	-180112	-169019	-168134	54	11	1.513	0.041	4.362
$B_{1.0}$	-179554	-168717	-168214	53	59	1.555	0.042	8.830
$B_{0.9}$	-179500	-168973	-168282	56	45	1.607	0.042	22.56
$B_{0.5}$	-179742	-168906	-168017	62	7	2.158	0.046	6.182
$\{5,4\}$	-195952	-173641	-175671	38	20	0.159	0.024	5.700



D Choice of the tessellation

Table 5 presents the experimental results of running our algorithm on the Facebook social circle network graph for various tessellations. We can obtain a coarser grid than G_{710} by using octagons instead of heptagons (G_{810}), and a finer grid by using the Goldberg-Coxeter construction, which adds extra hexagons to the order-3 heptagonal tessellation (G_{7ab}). We can also use a tessellation B_x based on the binary tiling [7] (where x is the width of the tile), or $\{5,4\}$, where four pentagons meet in a vertex. Log-likelihoods are named as in Section 5:
L_2, L_7 – continuous best log-likelihoods
L_3, L_5 – best discrete log-likelihoods, logistic function
L_4, L_6 – best discrete log-likelihood, arbitrary function of distance (used for local search)
The column #it presents the number of iterations of local search; * denotes that we have stopped the process after this number of iterations, while no * denotes that the local search could not improve the log-likelihood any further. MB is the amount of memory in megabytes, and time is in seconds.
As we can see, finer tessellations give better log-likelihoods, but a too dense grid dramatically decreases the performance without giving significant benefits. Tessellation B_x does not yield significantly better results, despite its circles' greater similarity to continuous ones. The results of $\{5,4\}$ are relatively bad; it approximates distances worse than three-valent tessellations.

Solving and Generating Nagareru Puzzles

Masakazu Ishihata
NTT Communication Science Laboratories, Kyoto, Japan

Fumiya Tokumasu
National Institute of Technology, Nara College, Nara, Japan

Abstract

Solving paper-and-pencil puzzles is fun for people, and their analysis is also an essential issue in computational complexity theory. There are some practically efficient solvers for some NP-complete puzzles; however, the automatic generation of interesting puzzle instances still stands out as a complex problem because it requires checking whether the generated instance has a unique solution. In this paper, we focus on a puzzle called Nagareru and propose two methods: one is for implicitly enumerating all the solutions of its instance, and the other is for efficiently generating an instance with a unique solution. The former constructs a ZDD that implicitly represents all the solutions. The latter employs the ZDD-based solver as a building block to check the uniqueness of the solution of generated instances. We experimentally showed that the ZDD-based solver was drastically faster than a CSP-based solver, and our generation method created an interesting instance in a reasonable time.

2012 ACM Subject Classification Computing methodologies → Combinatorial algorithms; Theory of computation → Generating random combinatorial structures; Mathematics of computing → Graph algorithms

Keywords and phrases Paper-and-pencil puzzle, SAT, CSP, ZDD

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.2

Supplementary Material *Software (Source Code and Data)*:
`https://github.com/masakazu-ishihata/Nagareru`
archived at `swh:1:dir:6c6604df4d55d8e2f019fdfe5e19225060e8ce83`

1 Introduction

Paper-and-pencil puzzles are a type of logic puzzle; a player gradually fills in parts of a solution on the puzzle board without violating any rules and eventually constructs a single consistent solution. Solving a puzzle is much fun for puzzle fans, but it has also attracted the extensive attention of theoretical computer scientists because of an interest in computational complexity [13]. They have been competing to prove the computational complexity of various puzzles, and the following is just a small selection of the list of puzzles that have so far proved to be NP-complete to solve: Bag (Corral) [3], Cross Sum [23], Country Road [5], Dosun-Fuwari [8], Herugolf [7], Hiroimono [2], Makaro [7], Moon-or-Sun [9], Nagareru [9], Nurikabe [4], Nurimeizu [9], Nurimisaki [10], Sashigane [10], Slitherlink [23], Sudoku (Number Place) [23], Tatamibari [1], Yajilin [5], Yosenabe [6], and more. The above series of studies is essential from the point of view of computational complexity theory; however, not so crucial for puzzle fans because it does not directly help them enjoy puzzles more. In contrast, the automatic generation of puzzle instances is one of the most promising computer science techniques for puzzle fans. They believe that one of the necessary conditions for an interesting puzzle instance is that the instance admits precisely one solution. However, for some puzzles, checking the uniqueness of solutions of an instance is an equally or more difficult task than solving the instance. Given an instance and its solution, finding another solution is called

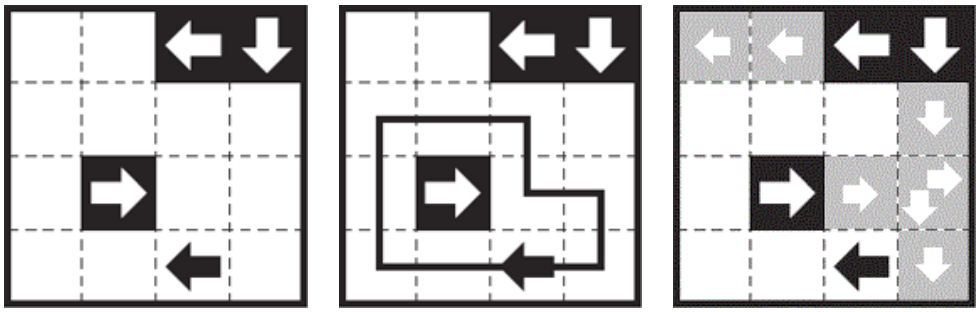
another solution problem (ASP) [21]. The ASP of an NP-complete problem is not necessary NP-complete; however, it has been shown that many puzzles are NP-complete not only in finding one solution but also in finding another solution [7, 10, 9].

In contrast to the theoretical difficulties of solving puzzles, practically efficient puzzle solvers have been proposed. One of the most popular approaches for solving puzzles is formulating a puzzle instance as a SAT problem, or its variants, including a constraint satisfaction problem (CSP) and satisfiability modulo theories (SMT), and solving it by a general constraint solver. For instance, Sugar [20], one of the latest CSP solvers, can solve a wide range of real-world instances of various puzzles, including a part of the above list [18]. Furthermore, some methods for generating puzzle instances have been proposed [22] that employ a SAT/CSP/SMT-based puzzle solvers as their building block to check the uniqueness of the generated instance; however, it has been reported that such a generator is too slow to generate a realistic instance (e.g., a 10×10 grid) because it calls the solver so many times.

Another promising approach to solving puzzles is using zero-suppressed decision diagrams (ZDDs) [14]. A ZDD is a compact graph representation of a set family and provides a variety of queries, including counting, sampling, and set operations, in linear time for its size. The frontier-based search (FBS) [11] is a meta-algorithm for constructing a ZDD representing constrained subgraphs of a target graph. Many FBS examples for various constraints have been proposed, e.g., trees, cycles, simple paths, and more complex constraints [12, 15]. Once a puzzle is formulated as a constrained subgraph finding problem, one can construct a ZDD-based puzzle solver by designing the FBS for the problem. For instance, Slitherlink is a puzzle played on a graph $G = (V, E)$ to find a single cycle $C \subseteq E$ consistent with given all *hints*, where a hint is a pair of an edge set $H \subseteq E$ and a positive number n and restricts C to $|C \cap H| = n$. Hence, Slitherlink can be formulated as a cycle finding problem with some cardinality constraints of some edge sets. It has been reported that a ZDD-based Slitherlink solver [24] performs faster than a CSP-based one [19], even though the former implicitly enumerates all the solutions, whereas the latter finds only one solution. In addition, the ZDD-based solver is helpful to generate puzzle instances because it can compute the number of solutions by the counting query of ZDDs. ZDD-based puzzle instance generators have been proposed for Slitherlink [24], Numberlink [24], and Minesweeper [17].

We focus on a puzzle called Nagareru [16], which has recently been proven to be NP-complete to solve and find another solution [9], and propose practically efficient methods for solving and generating its instance. Nagareru is a puzzle to draw a cycle that satisfies certain constraints like Slitherlink, but the cycle must have a global orientation consistent with some local orientation constraints (detailed rules are explained later), unlike Slitherlink. The FBS for Nagareru cannot be realized by combining existing FBS examples; namely, a new FBS is desired to solve Nagareru puzzles. The main contributions of this paper are threefold. First, we propose a ZDD-based Nagareru solver; we formulate Nagareru as a constraint cycle finding problem and propose the FBS for the constraints. Second, we propose an efficient Nagareru instance generator that employs the ZDD-based solver as its building block. Third, we empirically show that our solver outperforms a CSP-based solver and also that our generator creates an interesting instance in a reasonable time. Note that our generator is very different from those for other puzzles because the definition of "interesting" depends strongly on the target puzzle.

The rest of this paper is organized as follows: In Section 2, we review the rules of Nagareru and formulate it as a constrained cycle finding problem. We formulate a problem to find a constrained cycle as CSP in Section 3. In Section 4, we propose a new FBS for constructing a ZDD that implicitly enumerates all the constrained cycles; namely, it represents all the



(a) Instance. (b) Solution. (c) Instance with winds.

Figure 1 The grid board (a) is an instance of Nagareru, and the grid board (b) indicates its solution. The grid board (c) is the same instance as (a) with gray cells representing winds.

solutions of a Nagareru instance. In addition, we propose a new efficient generator of an "interesting" Nagareru instance that employs our ZDD-based solver to check the uniqueness of the solution in Section 5. We show the experimental results of the above methods in Section 6 and then conclude this paper in Section 7.

2 Problem Definition

2.1 Nagareru Puzzles

Nagareru is a paper-and-pencil puzzle played according to the following rules on a grid board [16]:

1. Draw a line to make a single continuous loop.
2. The line passes through the centers of cells, horizontally, vertically, or turning. It cannot cross itself, branch off, or go through the same cell twice.
3. The line must go through the white cells with a black arrow, and when you go along the arrows of the loop, that becomes the direction of all of the loop.
4. The white arrows in the black cells show that wind is blowing in the direction of the arrow till it reaches another black cell or the border. In cells where the wind blows, the line cannot advance against the wind.
5. When the line enters a cell where the wind blows, it must move at least one cell in that direction. When the line is blown like this (bent by a side wind), it cannot progress to or enter cells to hit the borders or enter black cells.

For example, the left grid board of Figure 1 is an instance of Nagareru, and the middle grid board indicates its solution.

We begin by formulating an instance of Nagareru. For any positive integer $n \in \mathbb{Z}_+$, let $[n] \equiv \{1, \ldots, n\}$. Given $w, h \in \mathbb{Z}_+$, a $w \times h$ gird board consists of w columns and h rows; namely, it has $w \times h$ cells. For any $w' \in [w]$ and $h' \in [h]$, let $i = w' + w(h' - 1)$ refer to the cell in the w'th column from the left and the h'th row from the top. For any $i \in [wh]$, let $\mathrm{adj}(i) \subset [wh]$ be a set of adjacent cells of i. Let $D \equiv \{\mathsf{Up}, \mathsf{Down}, \mathsf{Left}, \mathsf{Right}, \mathsf{No}\}$ denote a set of directions, where No indicates non-directional. For any $d \in D \setminus \{\mathsf{No}\}$, let $d^{-1} \in D$ denote the opposite direction of d. For any $i, j \in [wh]$, let $\mathrm{rel}(i, j) \in D$ denote the relative direction from i to j if i and j are adjacent each other, and $\mathrm{rel}(i, j) = \mathsf{No}$ if otherwise. For any $i \in [wh]$ and $d \in D \setminus \{\mathsf{No}\}$, there exists at most one adjacent cell $j \in \mathrm{adj}(i)$ satisfying $\mathrm{rel}(i, j) = d$, denoted by i_d, where $i_d = \mathsf{Null}$ denotes $\mathrm{rel}(i, j) \neq d$ for any $j \in \mathrm{adj}(i)$. Then, an instance of Nagareru is defined as follows:

▶ **Definition 1** (An instance of Nagareru). *Let $W \subseteq [wh] \times (D \setminus \{\mathsf{No}\})$ be white cells with (black) arrows and $B \subseteq [wh] \times D$ be black cells with (white) arrows. $P \equiv (w, h, W, B)$ is an instance of Nagareru on a $w \times h$ grid board if P satisfies $\forall\{(i,d),(i',d')\} \subseteq W \cup B, i \neq i'$.*

▶ **Theorem 2** (Hardness of finding a solution of Nagareru [9]). *For input Nagareru instance P, checking whether P admits a solution or not is NP-complete.*

Let $\omega \equiv (\omega_1, \ldots, \omega_L) \in [wh]^L$ be a cell sequence of L-length. For any black cell $(i,d) \in B$ and $j \in [wh]$, ω is a *wind path* from i to j if ω satisfies the following conditions:

- $\omega_1 = i$ and $\omega_L = j$,
- $\forall l \in [L-1], \forall d' \in D, (\omega_{l+1}, d') \notin B$: ω has no other black cell than (i,d),
- $\forall l \in [L-1], \mathrm{rel}(\omega_l, \omega_{l+1}) = d$: ω is a straight path of direction d.

Namely, the wind path ω from i to j indicates that a wind of direction d goes from i to j. For any $i \in [wh]$, we use $D_i^{\mathrm{wind}} \subseteq D$ to denote the directions of winds blowing on i; namely, $d \in D_i^{\mathrm{wind}}$ indicates that there exist a black cell $(j,d) \in B$ and a wind path ω of direction d from j to i. We here introduce new colors, *gray* and *colorless* to make the explanation easier: for any $i \in [wh]$ such that $\forall d \in D, (i,d) \notin W \cup B$, i is *gray* if $D_i^{\mathrm{wind}} \neq \emptyset$ and *colorless* if $D_i^{\mathrm{wind}} = \emptyset$; namely, a cell with no arrow is gray if it is blown, and colorless if otherwise. For example, the grid board (c) of Figure 1, obtained by adding gray and colorless to the grid board (a), consists of three black cells, one white cell, six gray cells, and six colorless cells.

2.2 Formulating Nagareru as a constrained cycle finding problem

We formulate the problem of finding a solution of a Nagareru instance as a constrained cycle finding problem on a graph representing the instance. Let $C \equiv \{\mathsf{White}, \mathsf{Black}, \mathsf{Gray}, \mathsf{No}\}$ denote a set of colors, where No indicates colorless. For any set V and $n \in \mathbb{Z}_+$, let $\binom{V}{n} \equiv \{S \subseteq V \mid |S| = n\}$. For any $i \in [wh]$ and $d \in D$, let $E_{i,d}^{\perp} \equiv \{\{i,j\} \mid j \in \mathrm{adj}(i)\,, \mathrm{rel}(i,j) \notin \{d, d^{-1}\}\}$ be edges of i that are orthogonal to direction d.

▶ **Definition 3** (A graph representation of a Nagareru instance). *Let $\mathrm{G} \equiv \langle V, E, \mathrm{col}, \mathrm{dir}\rangle$ where $V \subseteq [wh]$ is a vertex set, $E \subseteq \binom{V}{2}$ is an edge set, $\mathrm{col} : V \to C$ defines the color of each vertex, and $\mathrm{dir} : V \to 2^D$ defines the direction set of each vertex. G represents a Nagareru instance P if G satisfies following conditions:*

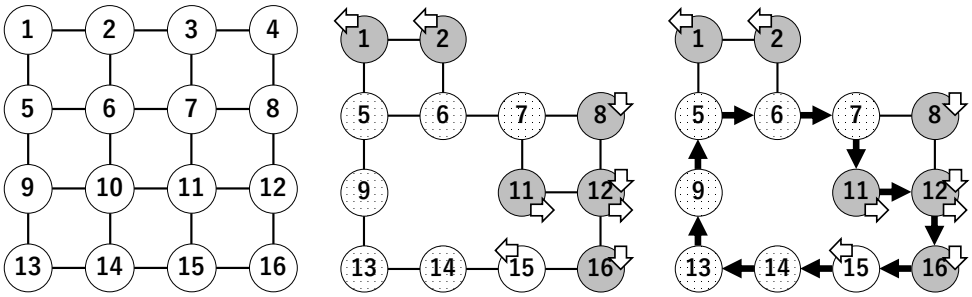
$$V \equiv [wh] \setminus \bigcup_{(i,d)\in B} \{i\}, \tag{1}$$

$$E \equiv \left\{\{i,j\} \in \binom{V}{2} \mid j \in \mathrm{adj}(i)\right\} \setminus \bigcup_{(i,d)\in W} E_{i,d}^{\perp}, \tag{2}$$

$$\mathrm{col}(i) \equiv \begin{cases} \mathsf{White} & \exists d \in D, (i,d) \in W \\ \mathsf{Gray} & \forall d \in D, (i,d) \notin W, D_i^{\mathrm{wind}} \neq \emptyset \\ \mathsf{No} & \forall d \in D, (i,d) \notin W, D_i^{\mathrm{wind}} = \emptyset \end{cases}, \tag{3}$$

$$\mathrm{dir}(i) \equiv \begin{cases} \{d\} & (i,d) \in W \\ D_i^{\mathrm{wind}} & \textit{otherwise} \end{cases}. \tag{4}$$

For any color $c \in C$, let $V_c \equiv \{i \in V \mid \mathrm{col}(i) = c\}$. Equation (1) leads to $V_{\mathsf{Black}} = \emptyset$. Equation (2) indicates that E has no edge inconsistent with (orthogonal to) an arrow of a white vertex. Figure 2(a) indicates a 4x4 grid graph, and Figure 2(b) is the graph representation of the instance shown in Figure 1.



(a) 4x4 grid graph. (b) Graph representation. (c) Solution.

Figure 2 The graph (a) indicates a 4x4 grid graph with 16 vertices and 24 edges. The graph (b) is the graph representation of the instance shown in Figure 1, where white, gray, and dotted circles represent vertices colored by white, gray, and colorless, respectively, and white arrows attached to each vertex i indicate $\mathrm{dir}(i)$. A directed cycle of bold black directed edges in the graph (c) forms a solution of Nagareru.

For any $F \subseteq E$, let $V[F] \equiv \bigcup_{e \in F} e$ denote a set of all endpoints of F, $\mathrm{G}[F] \equiv \langle V[F], F, \mathrm{col}, \mathrm{dir} \rangle$ denote an edge-induced subgraph of G, $\mathrm{nei}_F(i) \equiv \{j \in V[F] \mid \{i,j\} \in F\}$ denote the neighbors of i on $\mathrm{G}[F]$, and $\deg_F(i) \equiv |\mathrm{nei}_F(i)|$ denote the degree of i on $\mathrm{G}[F]$. Let n, m, n_F, and m_F denote $|V|$, $|E|$, $|V[F]|$, and $|F|$, respectively.

▶ **Definition 4** (A solution of a Nagareru instance). *An edge set $F \subseteq E$ is a solution of a Nagareru instance P if there exits a permutation of $V[F]$, denoted by $p \equiv (p_1, \ldots, p_{n_F})$, such that the following conditions are satisfied, where let $p_0 \equiv p_{n_F}$, $p_{n_F+1} \equiv p_1$, $e_l \equiv \{p_l, p_{l+1}\}$, and $r_l \equiv \mathrm{rel}(p_l, p_{l+1})$ for any $l \in [n_F]$.*

$$\forall i \in V_{\mathsf{White}}, \exists l \in [n_F], p_l = i, \tag{5}$$
$$\forall l \in [n_F], \deg_F(p_l) = 2 \wedge e_l \in F, \tag{6}$$
$$\forall l \in [n_F], \mathrm{col}(p_l) = \mathsf{White} \implies \forall d \in \mathrm{dir}(p_l), r_{l-1} \neq d^{-1}, r_l \neq d^{-1}, \tag{7}$$
$$\forall l \in [n_F], \mathrm{col}(p_l) = \mathsf{Gray} \implies \forall d \in \mathrm{dir}(p_l), r_{l-1} \neq d^{-1}, r_l \neq d^{-1}, \tag{8}$$
$$\forall l \in [n_F], \mathrm{col}(p_l) = \mathsf{Gray} \implies \forall d \in \mathrm{dir}(p_l), \{e_{l-1}, e_l\} \neq E^{\perp}_{p_l,d}. \tag{9}$$

Equation (5) guarantees that every white vertex is contained in $\mathrm{G}[F]$. Equation (6) restricts $\mathrm{G}[F]$ to be a cycle. Equation (7) (resp. (8)) prohibits $\mathrm{G}[F]$ from advancing against any wind of a white (resp. gray) vertex. Equation (9) prohibits $\mathrm{G}[F]$ from orthogonal to (crossing) any wind of a gray vertex. Consequently, $\mathrm{G}[F]$ forms a solution of P. Let $\mathcal{F}_P \subseteq 2^E$ be the set of all the solutions of P. P is said to be *invalid*, *valid*, and *good* if it has no solution ($|\mathcal{F}|_P = 0$), at least one solution ($|\mathcal{F}_P| \geq 1$), exactly one solution ($|\mathcal{F}_P| = 1$), respectively. For example, let P be a Nagareru instance shown in Figure 1, G be its graph representation shown in Figure 2(b), and F be the set of bold black arrows (edges) in Figure 2(c). Then, $\mathrm{G}[F]$ forms a solution of P, and P admits no other solution; namely, P is a good instance.

3 A CSP-based Nagareru solver

We propose a baseline method for finding a solution of a Nagareru instance. The method consists of three steps: (1) translating a constrained subgraph finding problem on a Nagareru instance as a CSP instance, (2) solving the CSP instance by a CSP solver, and (3) converting the obtained CSP solution to the solution of Nagareru.

A CSP instance is denoted by a triplet of variables, variable domains, and constraints. We first introduce variables and their domains. Because $\mathrm{G}[F]$ for any $F \in \mathcal{F}_P$ forms a directed cycle, we introduce the same idea of a CSP formulation of Slitherlink [19] to represent a cycle constraint; introducing auxiliary variables to represent a visiting order of variables that forms a cycle. Let $\mathbf{D}_E \equiv \{(i,j) \mid i < j, \{i,j\} \in F\}$. Then, our CSP formulation contains following four types of variables and domains: $u_{i,j} \in \{-1, 0, +1\}$, $d_i \in \{0, 2\}$, $q_i \in \{0, 1, \ldots, m\}$, and $s_i \in \{0, 1\}$ for any $(i,j) \in \mathbf{D}_E$. $u_{i,j}$ denotes the use of edge $\{i,j\}$ $(i < j)$ with direction: $u_{i,j} = 0$ indicates $\{i,j\} \notin F$ and $u_{i,j} = +1$ (resp. -1) indicates $\{i,j\} \in F$ with the forward direction $\mathrm{rel}(i,j)$ (resp. backward direction $\mathrm{rel}(j,i)$). d_i denotes $\deg_F(i)$. A set of q_i denotes a permutation of $V[F]$: $q_i = 0$ indicates $i \notin V[F]$ and the rest q_i define a permutation $p = (p_1, \ldots, p_{n_F})$ such as $p_{q_i} = i$. s_i denotes the starting vertex of the permutation p: $q_i = 1$ is represented by $s_i = 1$ and $s_{i'} = 0$ for $i' \in [wh] \setminus \{i\}$.

We then introduce constraints such that an assignment of the above variables satisfying the conditions if-and-only-if P is valid; in other words, we describe Equation (5), (6), (7), (8), and (9) of Definition 4 as formulas of the above variables. We first introduce the following sets of doublets and triplets of indices:

$$
\begin{aligned}
\mathbf{D}_i &\equiv \{(j,k) \in \mathbf{D}_E \mid i \in \{j,k\}\},\\
\mathbf{F}_c &\equiv \{(i,j) \in \mathbf{D}_E \mid \exists k \in \{i,j\}, \mathrm{col}(k) = c, \mathrm{rel}(i,j) \in \mathrm{dir}(k)\},\\
\mathbf{B}_c &\equiv \{(i,j) \in \mathbf{D}_E \mid \exists k \in \{i,j\}, \mathrm{col}(k) = c, \mathrm{rel}(j,i) \in \mathrm{dir}(k)\},\\
\mathbf{T}_\perp &\equiv \{(i,j,k) \mid (i,j),(j,k) \in \mathbf{D}_E, \mathrm{col}(j) = \mathsf{Gray}, \exists d \in \mathrm{dir}(j), \{\{i,j\},\{j,k\}\} = E^{\perp}_{j,d}\}.
\end{aligned}
$$

Then, the constraints of our CSP formulation are follows:

$$
\begin{aligned}
&\textstyle\bigwedge_{i \in V_{\mathsf{White}}} (q_i > 0)\,, && (10)\\
&\left(\textstyle\bigwedge_{i \in V} \left(d_i = \sum_{(j,k) \in \mathbf{D}_i} |u_{j,k}|\right)\right) \wedge \left(\textstyle\bigwedge_{i \in V} \left(\sum_{j:(j,i) \in \mathbf{D}_i} u_{j,i} = \sum_{j:(i,j) \in \mathbf{D}_i} u_{i,j}\right)\right), && (11)\\
&\textstyle\bigwedge_{i \in V} (q_i > 0 \implies d_i > 0) \wedge (q_i = 1 \iff s_i = 1)\,, && (12)\\
&\textstyle\bigwedge_{i \in V} (u_{i,j} = +1 \Rightarrow (q_i + 1 = q_j \vee q_j = 1))\,, && (13)\\
&\textstyle\bigwedge_{i \in V} (u_{i,j} = -1 \Rightarrow (q_i = q_j + 1 \vee q_i = 1))\,, && (14)\\
&\textstyle\sum_{i \in V} s_i = 1, && (15)\\
&\left(\textstyle\bigwedge_{(i,j) \in \mathbf{F}_{\mathsf{White}}} (u_{i,j} \neq -1)\right) \wedge \left(\textstyle\bigwedge_{(i,j) \in \mathbf{B}_{\mathsf{White}}} (u_{i,j} \neq +1)\right), && (16)\\
&\left(\textstyle\bigwedge_{(i,j) \in \mathbf{F}_{\mathsf{Gray}}} (u_{i,j} \neq -1)\right) \wedge \left(\textstyle\bigwedge_{(i,j) \in \mathbf{B}_{\mathsf{Gray}}} (u_{i,j} \neq +1)\right), && (17)\\
&\textstyle\bigwedge_{(i,j,k) \in \mathbf{T}_\perp} (u_{i,j} = 0 \vee u_{j,k} = 0)\,, && (18)
\end{aligned}
$$

Equation (10) corresponds to Equation (5). Equation (11) defines $d_i = \deg_F(i)$. Equation (12), (13), (14), and (15) jointly represent Equation (6). Equation (16), (17), and (18) correspond to Equation (7), (8), and (9), respectively.

▶ **Proposition 5** (A CSP formulation of Nagareru)**.** *The triplet of the above variables, domains, and constraints is a CSP formulation for finding a solution $F \in \mathcal{F}_P$ of a Nagareru instance P, and F is constructed from a CPS solution as $F = \{\{i,j\} \mid (i,j) \in \mathbf{D}_E, u_{i,j} \neq 0\}$.*

4 A ZDD-based Nagareru Solver

We here propose a ZDD-based Nagareru solver that constructs a ZDD representing $\mathcal{F}_P$. We first review a ZDD, a compact graph expression of a set family, and FBS, a meta-algorithm

to construct a ZDD for constrained subgraphs. Then, we propose a new FBS for a ZDD of $\mathcal{F}_P$.

4.1 ZDDs for subgraphs

A ZDD Z is a compact graph representation of a set family over a universe set E and let $\mathcal{F}_{\mathrm{Z}} \subseteq 2^E$ be the set family represented by Z. When the universe set E is an edge set of a graph $\mathrm{G} \equiv \langle V, E \rangle$, Z can be regarded as a set of edge-induced subgraphs $\mathrm{G}[F]$ for each $F \in \mathcal{F}_{\mathrm{Z}}$. To avoid confusing two graphs Z and G, we use terms *nodes* and *arcs* for describing Z. A ZDD requires a total order on E denoted by $\succ$ and let e_l denote the lth smallest element of E. Then, a ZDD Z and its set family $\mathcal{F}_{\mathrm{Z}}$ are defined as follows.

▶ **Definition 6** (A ZDD). *Let* $\mathrm{Z} \equiv \langle N, A_0, A_1, \ell \rangle$ *where* N *is a set of nodes,* $A_b \subseteq N \times N$ *is a set of* b*-arcs for any* $b \in \{0,1\}$*, and* $\ell : N \to E \cup \{\mathsf{Null}\}$ *defines the label of each node.* Z *is a ZDD if it satisfies the following conditions:*

- N *has exactly one root node denoted by* ρ *and exactly two terminal nodes denoted by* τ_0 *and* τ_1 *such as* $\ell(\tau_0) = \ell(\tau_1) = \mathsf{Null}$.
- *Each non-terminal node* $\nu \in N \setminus \{\tau_0, \tau_1\}$ *has exactly one outgoing* b*-arc for each* $b \in \{0,1\}$*, and is labeled by some element of* E*:* $\ell(\nu) \in E$.
- *For any arc* $(\nu, \nu') \in A_0 \cup A_1$*,* $\ell(\nu) \succ \ell(\nu')$ *holds where let* $e \succ \mathsf{Null}$ *for any* $e \in E$.

▶ **Definition 7** (A set family represented by a ZDD). *For any non-terminal node* $\nu \in N \setminus \{\tau_0, \tau_1\}$ *and* $b \in \{0,1\}$*, let* ν_b *be the node pointed by the* b*-arc of* ν*, referred to as the* b*-child of* ν*. For any* $\nu \in N$*, let* $\mathcal{F}_\nu \subseteq 2^E$ *be a set family recursively-defined as*

$$\mathcal{F}_{\tau_0} \equiv \emptyset, \qquad \mathcal{F}_{\tau_1} \equiv \{\emptyset\}, \qquad \mathcal{F}_\nu \equiv \mathcal{F}_{\nu_0} \cup \{F \cup \{\ell(\nu)\} \mid F \in \mathcal{F}_{\nu_1}\}.$$

Then, Z *is said to represent* $\mathcal{F}_\rho$ *denoted by* $\mathcal{F}_{\mathrm{Z}}$.

Definition 6 restricts Z to a rooted directed cyclic graph (DAG). Definition 7 defines $\mathcal{F}_\nu$ in a bottom-up manner; however, it has another intuitive definition as follows. Let $\pi \equiv (\pi_1, \ldots, \pi_L) \in N^L$ be a directed path from π_1 to π_L on Z of L-length, and also let $F_\pi \equiv \{\ell(\pi_l) \mid l \in [L-1], (\pi_l, \pi_{l+1}) \in A_1\}$; namely, F_π contains $\ell(\nu)$ if π contains the 1-arc of ν. Let $\Pi_{\nu \to \nu'}$ be a set of all directed path from ν to ν' on Z, and also let $\mathcal{F}_{\nu \to \nu'} \equiv \{F_\pi \mid \pi \in \Pi_{\nu \to \nu'}\}$. Then, for any $\nu \in N$, $\mathcal{F}_\nu = \mathcal{F}_{\nu \to \tau_1}$ holds [14]. Consequently, checking $|\mathcal{F}_{\mathrm{Z}}| > 0$ corresponds to finding a directed path from ρ to τ_1 on Z, and counting $|\mathcal{F}_{\mathrm{Z}}|$ is equivalent to counting such paths. Since Z is a rooted DAG, dynamic programming solves both tasks in $O(|N|)$ time.

4.2 FBS for constrained subgraphs

A ZDD Z is said to represent constraint subgraphs of G if $\mathrm{G}[F]$ for all $F \in \mathcal{F}_{\mathrm{Z}}$ satisfies the target constraint but not for any $F \notin \mathcal{F}_{\mathrm{Z}}$. FBS is a meta-algorithm to construct such Z. The basic idea of FBS is layer-wise top-down construction. For any $l \in [m]$, let N_l be non-terminal nodes labeled by e_l, and also let l be referred to as *layer*. FBS initializes $N_m \equiv \{\rho\}$ and repeats the generation of N_{l-1} from N_l in order from the top layer m to the bottom layer 1, and the resulting $N = (\cup_{l \in [m]} N_l) \cup \{\tau_0, \tau_1\}$ forms a ZDD. The essential idea of FBS is introducing a *state* to each node ν, denoted by S_ν. Each S_ν is a set of some variables, and its specific definition depends on the constraint of interest. We use $\mathrm{S}_\nu.x$ to denote the value of the variable x in S_ν. Given $\nu \in N_l$ and its state S_ν, its b-child ν_b and its state $\mathrm{S}.\nu_b$ is generated by *only* using the information of S_ν and G. In other words, all the necessary information to create the children of ν should be concentrated in S_ν. In addition,

Algorithm 1 ConstructZDD.

Let ρ be a new node and $\ell(\rho) \leftarrow m$. ▷ *Initialize the root*
$N_m \leftarrow \{\rho\}$, $N_l \leftarrow \emptyset$ for $l \in [m-1]$ ▷ *Initialize the node set N*
$A_b \leftarrow \emptyset$ for $b \in \{0,1\}$ ▷ *Initialize the arc sets A_0 and A_1*
for $l = m, \ldots, 1$ **do** ▷ *The layer-wise top-down construction*
  for $\nu \in N_l$ **do**
    for $b \in \{0,1\}$ **do**
      $\nu_b \leftarrow \mathrm{getChild}(\mathrm{S}_\nu, l, b)$ ▷ *Create ν_b, the b-child of ν*
      if $\nu_b \in \{\tau_0, \tau_1\}$ **then** ▷ *Branching*
        ▷ *do nothing* ◁
      else if $\exists \nu' \in N_{l-1} : \mathrm{S}_{\nu_b} = \mathrm{S}_{\nu'}$ **then** ▷ *ν_b has an identical existing node ν'*
        $\nu_b \leftarrow \nu'$ ▷ *Merge ν' and ν_b*
      else ▷ *ν_b has no identical existing node*
        $\ell(\nu_b) \leftarrow l - 1$
        $N_{l-1} \leftarrow N_{l-1} \cup \{\nu_b\}$ ▷ *Add ν_b to N_{l-1} as a new node*
      $A_b \leftarrow A_b \cup \{(\nu, \nu_b)\}$ ▷ *Add (ν, ν_b), the b-arc of ν, to A_b as a new arc*
$N \leftarrow \left(\bigcup_{l \in [m]} N_l\right) \cup \{\tau_0, \tau_1\}$ ▷ *Unite all layers*
return $\langle N, A_0, A_1, \ell \rangle$

nodes with the same state must have the same children. Hence, such identical nodes in the same layer can be merged into a single node. Algorithm 1 is the pseudo-code of FBS, where a subroutine $\mathrm{getChild}(\mathrm{S}_\nu, l, b)$, which returns the b-child of ν with its state, is defined depend on the target constraint.

▶ **Theorem 8** (The complexity of FBS [11]). *For any $l \in [m]$, let κ_l be the number of different realizations of the state at the lth layer; namely, $|N_l| = \kappa_l$ holds. Let $\kappa \equiv \max_{l \in [m]} \kappa_l$. Then, the space and time complexity of FBS is $O(m\kappa)$ under the assumption that the identical state can be found in $O(1)$ time.*

4.2.1 Example: the size constraint

Let us consider the FBS for the size constraint $|F| \leq K$. For any non-terminal node ν, let S_ν consist only of an integer variable x_{used} that indicates the number of passed 1-arcs from the root ρ to ν; namely, $|F_\pi| = \mathrm{S}_\nu.x_{\mathrm{used}}$ holds for any $\pi \in \Pi_{\rho \to \nu}$ of Z under construction. $\mathrm{S}_\nu.x_{\mathrm{used}} = K$ indicates that F_π cannot adopt any more edges. Two nodes ν and ν' are equivalent if $\mathrm{S}_\nu = \mathrm{S}_{\nu'}$ because the paths from ρ to them passed the same number of 1-arcs.

In summary, Algorithm 2 describes $\mathrm{getChild}(\mathrm{S}, l, b)$ of the size constraint. Since $\kappa_l \leq K$ holds for any $l \in [m]$, the resulting ZDD size is $O(mK)$ by Theorem 8.

Algorithm 2 $\mathrm{getChild}(\mathrm{S}, l, b)$ for a size constraint $|F| \leq K$.

Let S' be a new state.
$\mathrm{S}'.x_{\mathrm{used}} \leftarrow (l = m)\ ?\ 0 : \mathrm{S}.x_{\mathrm{used}}$ ▷ *Initialize & Copy*
if $b = 1$ **then** ▷ *Adopt e_l*
  $\mathrm{S}'.x_{\mathrm{used}} \leftarrow \mathrm{S}'.x_{\mathrm{size}} + 1$ ▷ *Update S'*
  return τ_0 **if** $\mathrm{S}'.x_{\mathrm{size}} > K$ ▷ *Pruning: detect $|F| > K$*
return τ_1 **if** $l = 1$ ▷ *Termination: reach the end without the violation*
return a new node ν with $\mathrm{S}_\nu = \mathrm{S}'$

Algorithm 3 getChild(S, l, b) for a cycle constraint.

Let $e_l = \{i, j\}$, S' be a new state.
$\mathrm{S}'.m_k \leftarrow \mathrm{S}.m_k$ **for each** $k \in V_l \cap V_{l+1}$ ▷ *Copy*
$\mathrm{S}'.m_k \leftarrow k$ **for each** $k \in V_l \setminus V_{l+1}$ ▷ *Initialize*
$m_k \leftarrow \mathrm{S}'.m_k$ **for each** $k \in V_l$ ▷ *Abbreviate*
if $b = 1$ **then** ▷ *Adopte* e_l
  return τ_0 **if** $m_i = \mathsf{Null} \vee m_j = \mathsf{Null}$ ▷ *Pruning: detect a branching*
  if $m_i = j \wedge m_j = i$ **then** ▷ *Detect a cycle*
    return τ_0 **if** $\exists k \in V_l \setminus e_l, m_k \in V_k \setminus \{k\}$ ▷ *Pruning: detect a redundant endpoint*
    return τ_1 ▷ *Termination: complete a single cycle*
  ▷ *Update* S' ◁
  $\mathrm{S}'.m_{m_i} \leftarrow m_j$
  $\mathrm{S}'.m_{m_j} \leftarrow m_i$
  $\mathrm{S}'.m_i \leftarrow \mathsf{Null}$ **if** $\mathrm{S}'.m_j \neq i$
  $\mathrm{S}'.m_j \leftarrow \mathsf{Null}$ **if** $\mathrm{S}'.m_i \neq j$
return τ_0 **if** $\exists k \in V_l \setminus V_{l-1}, \mathrm{S}'.m_k \in V_{l-1}$ ▷ *Pruning: detect a leaving endpoint*
return τ_0 **if** $l = 1$ ▷ *Pruning: reach the end without completing a cycle*
return a new node ν with $\mathrm{S}_\nu = \mathrm{S}'$

4.2.2 Example: the cycle constraint

Let us consider the FBS for the cycle constraint. For each layer $l \in [m]$, let $E_{\leq l} \equiv \{e_{l'} \in E \mid l' \leq l\}$, $E_{\geq l} \equiv \{e_{l'} \in E \mid l' \geq l\}$, $V_l \equiv \{i \in V \mid \exists e \in E_{\leq l}, \exists e' \in E_{\geq l}, i \in e \cap e'\}$, and $\lambda \equiv \max_{l \in [m]} |V_l|$, where V_l and λ are referred to as the *frontier* of the lth layer and the maximum frontier size, respectively.

For each non-terminal node $\nu \in N_l$, let $\mathrm{S}_\nu \equiv \{m_k \mid k \in V_l\}$ where $m_k \in V_l \cup \{\mathsf{Null}\}$ is referred to as the *mate* of k. The mate m_k indicates the connectivity of k in $\mathrm{G}[F_\pi]$ for any $\pi \in \Pi_{\rho \to \nu}$: (1) $m_k = k$ indicates that k is an isolated fragment (vertex), (2) $m_k \in V_l \setminus \{k\}$ indicates that k and m_k are the endpoints of a path fragment, and (3) $m_k = \mathsf{Null}$ indicates that k is an intermediate vertex of a path fragment.

We next consider what will happen if an edge $e_l = \{i, j\}$ is adopted to F_π. If $m_i = \mathsf{Null} \vee m_j = \mathsf{Null}$ (i.e., i and/or j is an intermediate vertex of a path fragment), adopting e_l violates the cycle condition because it causes a branching. Otherwise, it connects two fragments to which i belongs and j belongs. More specifically, if $m_i \neq j \wedge m_j \neq i$ (i.e., i and j belong to different fragments), it constructs a new path fragment whose endpoints are m_i and m_j. If $m_i = j \wedge m_j = i$ (i.e., i and j are the endpoints of the same path fragment), it completes a cycle; in addition, if $\mathrm{G}[F_\pi]$ has no redundant path fragment (i.e., $\forall k \in V_l \setminus e_l, m_k \notin V_l \setminus \{k\}$), $\mathrm{G}[F_\pi]$ forms a single cycle. Similarly, leaving an endpoint k (i.e., $m_k \in V_l \setminus \{k\}$) from the frontier violates the cycle constraint because k has no chance to join a cycle anymore; namely, k is fixed as an endpoint of a redundant path fragment.

In summary, Algorithm 3 describes getChild(S, l, b) of the cycle constraint. Since S_ν corresponds to a matching in the complete graph $(V_l, \binom{V_l}{2})$, $\kappa_l \leq 2^{|V_l|^2}$ holds and the resulting ZDD size is $O(m2^{\lambda^2})$ by Theorem 8. In practice, however, S_ν does not take as many realizations as 2^{λ^2}, the actual ZDD size is empirically much smaller.

4.3 The FBS for the Nagareru constraints

We here propose the FBS for the Nagareru constraints shown in Definition 4. For any non-terminal node ν, let S_ν consist of three types of variables: $m_k \in V_l \cup \{\mathsf{Null}\}$, $u_k \in \{0,1\}$, and $d_k \in D \cup \{\mathsf{Null}\}$ for any $k \in V_l$. m_k is the exactly same as the mate of Example 2 and indicates the connectivity of k on $\mathrm{G}[F_\pi]$. u_k indicates the upper stream of path fragments on $\mathrm{G}[F_\pi]$. d_k indicates the relative direction from the neighbor of k to k on $\mathrm{G}[F_\pi]$. More specifically, u_k and d_k are defined as follows: If $m_k \notin V_l \setminus \{k\}$ (i.e., k is an isolated vertex or an intermediate vertex of a path fragment), let $u_k = 0$ and $d_k = \mathsf{Null}$. If $m_k \in V_l \setminus \{k\}$ (i.e., k is an endpoint of a path fragment on $\mathrm{G}[F_\pi]$), let $u_k = 1$ indicate that k must be upper stream of a path fragment whose endpoints are k and m_k, and also let $d_k \equiv \mathrm{rel}(k', k)$ where $\{k', k\} \in F_\pi$. By regarding u_k as a Boolean variable, $u_k \wedge u_{m_k}$ must always be false and $\neg u_k \wedge \neg u_{m_k}$ indicates that the upper stream of the path fragment is not yet decided.

Let us consider what will happen if $e_l = \{i, j\}$ is adapted to F_π. As Example 2, if $m_i \neq j \wedge m_j \neq i$, adapting e_l connects two different path fragments on $\mathrm{G}[F_\pi]$ and constructs a new path fragment whose endpoints are m_i and m_j. If $u_i \wedge u_j$ (resp. $u_{m_i} \wedge u_{m_j}$), it corresponds to connecting up streams (resp. down streams) of the two path fragments on $\mathrm{G}[F_\pi]$; namely, $\mathrm{G}[F_\pi]$ has no consistent direction. If $u_i \vee u_{m_j}$ (resp. $u_j \vee u_{m_i}$), the up stream of the resulting path fragment should be m_j (resp. m_i). In addition, the direction of e_l decided by its colored endpoints must be consistent with this direction. If $\mathrm{col}(i) = \mathsf{White} \vee \mathrm{col}(j) = \mathsf{White}$, e_l must be adapted to satisfy Equation (5) of Definition 4. Let $l_{\mathsf{White}} \equiv \min\{l \in [m] \mid \exists k \in e_l, \mathrm{col}(k) = \mathsf{White}\}$. Then, completing a cycle before reaching l_{White}th layer deduces that at least one edge with a white endpoint is unused; namely, Equation (5) is violated. In summary, Algorithm 4 describes $\mathrm{getChild}(\mathrm{S}, l, b)$ of the Nagareru constraints shown in Definition 4.

The complexity of the proposed FBS is following: Because κ_l is less than $2^{|V_l|^2} \times 2^{|V_l|} \times |D|^{|V_l|}$, the product of the domain size of each variable in the state, $\kappa_l = O(2^{|V_l|^2})$ holds. Hence, its complexity is $O(m2^{\lambda^2})$ that is the same as Example 2, where λ depends on G and the total order $\succ$ on E. For instance, when $\succ$ is defined as $e \succ e' \Leftrightarrow (\min e < \min e') \vee (\min e = \min e' \wedge \max e < \max e')$, λ of an $n \times n$ grid graph is n.

▶ **Proposition 9** (The FBS for the Nagareru constraints). *Given a Nagareru instance P, the FBS shown in Algorithm 1 with* $\mathrm{getChild}(\mathrm{S}, l, b)$ *shown in Algorithm 4 constructs a ZDD representing $\mathcal{F}_P$, a set of all the solutions of P defined by Definition 4. The complexity of the proposed FBS and the resulting ZDD size is $O(m2^{\lambda^2})$.*

5 An efficient Nagareru instance generator

In this section, we first define the "interesting" instance of Nagareru and propose an efficient Nagareru instance generator that generates interesting instances using our ZDD-based Nagareru solver as its building blocks.

5.1 An interesting instance of Nagareru

Let us begin by defining an *interesting* instance. Given an instance P and its graph G of Definition 3, we introduce *infeasible*, *ineffective*, and *redundant* cells as follows: A white cell $(i, d) \in W$ is infeasible if $D_i^{\mathrm{wind}} \not\subseteq \mathrm{dir}(i) \vee \deg_E(i) < 2$; namely, there exists a wind inconsistent to its arrow or there is not enough number of neighbors to follow its arrow. Consequently, P with an infeasible cell has no solution. A black cell $(i, d) \in B$ is ineffective if

Algorithm 4 getChild(S, l, b) for the Nagareru constraints.

▷ *Initialize & Copy* ◁
Let S' be a new state.
$\mathrm{S}'.m_k \leftarrow k$, $\mathrm{S}'.u_k \leftarrow 0$, $\mathrm{S}'.d_k \leftarrow \mathsf{Null}$ **for each** $k \in V_l \setminus V_{l+1}$
$\mathrm{S}'.m_k \leftarrow \mathrm{S}.m_k$, $\mathrm{S}'.u_k \leftarrow \mathrm{S}.u_k$, $\mathrm{S}'.d_k \leftarrow \mathrm{S}.d_k$ **for each** $k \in V_l \cap V_{l+1}$,
▷ *Abbreviate* ◁
Let $e_l = \{i, j\}$.
$m_k \leftarrow \mathrm{S}'.m_k$, $u_k \leftarrow \mathrm{S}'.u_k$, $d_k \leftarrow \mathrm{S}'.d_k$ **for each** $k \in V_l$
▷ *Direction that e_l must follow* ◁
$d_l \leftarrow \mathsf{Null}$
$d_l \leftarrow \mathrm{rel}(i, j)$ **if** $u_{m_i} \vee u_j$
$d_l \leftarrow \mathrm{rel}(j, i)$ **if** $u_{m_j} \vee u_i$
if $b = 1$ **then** ▷ *Adopt e_l*
  ▷ *Pruning: check Equation (6) of Definition 4* ◁
  return τ_0 **if** $m_i = \mathsf{Null} \vee m_j = \mathsf{Null}$
  return τ_0 **if** $(u_i \wedge u_j) \vee (u_{m_i} \wedge u_{m_j})$
  ▷ *Pruning: check Equation (7) of Definition 4* ◁
  return τ_0 **if** $\mathrm{col}(i) = \mathsf{White} \wedge d_l^{-1} \in \mathrm{dir}(i)$
  return τ_0 **if** $\mathrm{col}(j) = \mathsf{White} \wedge d_l^{-1} \in \mathrm{dir}(j)$
  ▷ *Pruning: check Equation (8) of Definition 4* ◁
  return τ_0 **if** $\mathrm{col}(i) = \mathsf{Gray} \wedge d_l^{-1} \in \mathrm{dir}(i)$
  return τ_0 **if** $\mathrm{col}(j) = \mathsf{Gray} \wedge d_l^{-1} \in \mathrm{dir}(j)$
  ▷ *Pruning: check Equation (9) of Definition 4* ◁
  return τ_0 **if** $\mathrm{col}(i) = \mathsf{Gray} \wedge d_i = \mathrm{rel}(i, j) \wedge (\exists d \in \mathrm{dir}(i), d_i \notin \{d, d^{-1}\})$
  return τ_0 **if** $\mathrm{col}(j) = \mathsf{Gray} \wedge d_j = \mathrm{rel}(j, i) \wedge (\exists d \in \mathrm{dir}(j), d_j \notin \{d, d^{-1}\})$
  if $m_i = j \wedge m_j = i$ **then** ▷ *Detect a cycle*
    return τ_0 **if** $l > l_{\mathsf{White}}$ ▷ *Pruning: detect a unused white vertex*
    return τ_0 **if** $\exists k \in V_l \setminus e_l, m_k \in V_l \setminus \{k\}$ ▷ *Pruning: detect a redundant endpoint*
    return τ_1 ▷ *Termination: complete a solution*
  ▷ *Update S'* ◁
  $\mathrm{S}'.m_{m_i} \leftarrow m_j$, $\mathrm{S}'.m_{m_j} \leftarrow m_i$,
  $\mathrm{S}'.m_i \leftarrow \mathsf{Null}$ **if** $m_i \neq i$
  $\mathrm{S}'.m_j \leftarrow \mathsf{Null}$ **if** $m_j \neq j$
  $\mathrm{S}'.d_i \leftarrow (m_i = i)\ ?\ j : \mathsf{Null}$
  $\mathrm{S}'.d_j \leftarrow (m_j = j)\ ?\ i : \mathsf{Null}$
  $\mathrm{S}'.u_{m_i} \leftarrow u_j$, $\mathrm{S}'.u_{m_j} = u_i$
  $\mathrm{S}'.u_{m_i} = 1$ **if** $\exists k \in e_l, (\mathrm{col}(k) \in \{\mathsf{White}, \mathsf{Gray}\}) \wedge (\mathrm{rel}(i, j) \in \mathrm{dir}(k))$
  $\mathrm{S}'.u_{m_j} = 1$ **if** $\exists k \in e_l, (\mathrm{col}(k) \in \{\mathsf{White}, \mathsf{Gray}\}) \wedge (\mathrm{rel}(j, i) \in \mathrm{dir}(k))$
else ▷ *Does not adopt e_l*
  ▷ *Pruning: check Equation (5) of Definition 4* ◁
  return τ_0 **if** $\mathrm{col}(i) = \mathsf{White} \vee \mathrm{col}(j) = \mathsf{White}$
return τ_0 **if** $\exists k \in V_l \setminus V_{l-1}, m_k \in V_{l-1}$ ▷ *Pruning: detect a leaving endpoint*
return τ_0 **if** $l = 1$ ▷ *Pruning: reach the end without completing a cycle*
return a new node ν with $\mathrm{S}_\nu = \mathrm{S}'$

$i_d = \mathsf{Null}$; namely, there is no cell affected by its wind. A white or black cell $(i, d) \in W \cup D$ is redundant if $|\mathcal{F}_P| = |\mathcal{F}_{P'}|$ where P' is obtained by removing (i, d) from P; namely, removing it does not change the number of solutions. We define that P is *interesting* if P is *good* (i.e., $|\mathcal{F}_P| = 1$) and contains neither infeasible, ineffective, nor redundant cell.

5.2 The proposed Nagareru instance generator

We propose a new efficient method to generate an interesting Nagareru instance P as follows:

1. Let $W = B = \emptyset$ and $P \equiv (w, h, W, B)$
2. Enumerate $A_W \subseteq [wh] \times D$ that is a set of non-infeasible white cells, and $A_B \subseteq [wh] \times D$ that is a set of non-ineffective black cells.
3. If $A_W = A_B = \emptyset$, restart this algorithm. Otherwise uniformly sample (i, d) from A_W (or A_B) without replacement, and add (i, d) to W (or B.)
4. If P is good, go to 5. If P is not good but valid, repeat 2-3. If P is invalid, delete (i, d) from P and go to 3.
5. Delete all redundant cells on P and output P.

The Algorithm employs our ZDD-based solver as its building block. In Step 4, the Algorithm constructs the ZDD Z of $\mathcal{F}_P$ and checks whether P is good, valid, or not by computing $|\mathcal{F}_P|$ on Z. In Step 5, for each white and black cell on P, the Algorithm constructs the ZDD of $\mathcal{F}_{P'}$ where P' is obtained by removing the cell from P and checks $|\mathcal{F}_P| = |\mathcal{F}_{P'}|$ or not. If every cell on P is non-redundant, the Algorithm outputs P as an interesting instance.

6 Experiments

We conducted several experiments and confirmed the following two facts:

1. The ZDD-based Nagareru solver works more efficiently than the CSP-based solver,
2. The ZDD-based Nagareru generator creates interesting instances with realistic board sizes in a reasonable time.

We have uploaded our code and datasets used in the experiments to the following GitHub repository: `https://github.com/masakazu-ishihata/Nagareru`.

6.1 Experimental Setting

Our CSP-based Nagareru solver was implemented using Sugar [20], a state-of-the-art CSP solver. Our ZDD-based Nagareru solver was implemented in C++ using TdZdd[1] that is a C++ library for FBS. Our Nagareru instance generator was also implemented in C++ using the ZDD-based solver as a building block. The ZDD-based solver and generator were compiled by g++ 11.0.3 with the -O3 option. All experiments were conducted on a 64-bit mac OS Big Sur 11.2.3 with six Intel Core i7 3.2 GHz CPU and 64 GB RAM; however, we ran all programs on a single core. The timeout for solving each instance is 100 seconds throughout the experiment.

The whole dataset for evaluation consisted of 10 synthetic datasets and one handcrafted dataset. Each synthetic dataset consisted of 100 interesting instances generated by our generator with the different gird size $(w, h) = (5, 5), (6, 6), \ldots, (14, 14)$. The handcrafted dataset consisted of 97 interesting instances on $(10, 10)$ grid board obtained by crawling some puzzle creators' blogs and collecting instances in PUZ-PRE format, where PUZ-PRE[2] is a web application for editing and playing paper-and-pencil puzzles.

[1] `https://github.com/kunisura/TdZdd`
[2] `http://pzv.jp/`

Table 1 The averages (Ave.), variance (Var.), and median (Med.) of the computation time (sec) and the numbers of solved instances (Sol.) of the CSP-based and the ZDD-based solver, respectively. Timeout instances were excluded when calculating the averages, variances, and medians.

Datataset	(w, h)	The CSP-based solver				The ZDD-based solver			
		Ave.	Var.	Med.	Sol.	Ave.	Var.	Med.	Sol.
Synthetic	$(5, 5)$	0.521	0.001	0.523	100	0.007	0.000	0.007	100
	$(6, 6)$	0.571	0.001	0.556	100	0.007	0.000	0.007	100
	$(7, 7)$	0.673	0.060	0.611	100	0.007	0.000	0.007	100
	$(8, 8)$	2.672	73.400	0.909	100	0.008	0.000	0.008	100
	$(9, 9)$	1.402	11.518	0.948	99	0.009	0.000	0.008	100
	$(10, 10)$	1.088	1.344	0.855	100	0.009	0.000	0.008	100
	$(11, 11)$	1.306	0.682	1.186	100	0.010	0.000	0.010	100
	$(12, 12)$	2.185	24.127	1.309	100	0.015	0.000	0.011	100
	$(13, 13)$	4.169	105.999	1.565	100	0.022	0.001	0.013	100
	$(14, 14)$	2.791	18.174	1.851	98	0.022	0.001	0.016	100
Handcrafted	$(10, 10)$	1.089	0.008	1.084	97	0.008	0.000	0.008	97

6.2 Experimental Results

Table 1 shows the computation times and numbers of solved instances of the CSP- and ZDD-based solver, respectively. It indicates that the ZDD-based solver is drastically faster than the CSP-based solver for each dataset; even though the former implicitly enumerates all the solutions, the latter finds only one solution. It also shows that the variance of the ZDD-based solver is significantly smaller than that of the CSP-based solver. Similar results have been reported for solving Slitherlink [24].

Table 2 indicates the statistics of each dataset; it shows that the average generation time increases exponentially with the grid size, whereas the average number of calls of the ZDD-based solver rises almost linearly. This result implies that the computation time of the ZDD-based solver increases exponentially with the grid size, which is consistent with its computational complexity shown in Proposition 9. It also shows that synthetic instances slightly tend to have more white cells, fewer black cells, and smaller solutions than handcrafted instances; however, it is unknown whether the proportion of white and black cells directly contributes to the fun of instances that humans feel. The interesting instances of $(14, 14)$ grid with the smallest/largest $|W|$, $|B|$, and $|F|$ are shown in Appendix; the one with the smallest $|F|$ seems too easy for humans to solve; however, the others seem complicated enough to enjoy solving. Note that our generator allows adjusting the ratio of white and black cells by changing the sampling distribution of its Step 3 and adjusting the size of the solution by adding a size constraint to Nagareru constraints.

7 Conclusion

We proposed an efficient solver and generator for Nagareru. Our solver constructs a ZDD representing all the solutions of a Nagareru instance by the FBS designed for this problem. Our generator employs our solver to guarantee that a generated instance is interesting; namely, it admits precisely one solution and has no redundant cell. We conducted some experiments and confirmed that our ZDD-based solver was drastically faster than a CSP-based one and our generator created interesting instances in a reasonable time.

Table 2 The first and second columns indicate the database type and grid size, respectively. The third and fourth ones indicate the averages of the construction time (sec) and the number of calls of the ZDD-based solver of each instance generation, respectively. The fifth and sixth ones indicate the average number of white and black cells, respectively. The seventh one indicates the average of the solution size $|F|$.

Dataset	(w, h)	Ave. Time	Ave. # calls	Ave. $\|W\|$	Ave. $\|B\|$	Ave. $\|F\|$
Synthetic	$(5, 5)$	0.020	26.280	2.050	2.850	11.500
	$(6, 6)$	0.040	39.930	2.920	3.430	16.360
	$(7, 7)$	0.085	59.880	3.900	4.810	22.680
	$(8, 8)$	0.212	84.460	5.200	6.470	30.580
	$(9, 9)$	0.677	110.640	6.800	8.530	42.940
	$(10, 10)$	2.638	148.380	8.800	10.720	53.980
	$(11, 11)$	11.660	197.880	10.780	12.710	65.620
	$(12, 12)$	58.152	240.010	12.590	15.780	76.680
	$(13, 13)$	317.790	281.710	14.630	18.510	88.840
	$(14, 14)$	1525.987	337.740	17.450	21.360	105.620
Handcrafted	$(10, 10)$	-	-	6.814	16.278	66.020

References

1 Aviv Adler, Jeffrey Bosboom, Erik D. Demaine, Martin L. Demaine, Quanquan C. Liu, and Jayson Lynch. Tatamibari is np-complete. In Martin Farach-Colton, Giuseppe Prencipe, and Ryuhei Uehara, editors, *10th International Conference on Fun with Algorithms, FUN 2021, May 30 to June 1, 2021, Favignana Island, Sicily, Italy*, volume 157 of *LIPIcs*, pages 1:1–1:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.FUN.2021.1.

2 Daniel Andersson. HIROIMONO is np-complete. In Pierluigi Crescenzi, Giuseppe Prencipe, and Geppino Pucci, editors, *Fun with Algorithms, 4th International Conference, FUN 2007, Castiglioncello, Italy, June 3-5, 2007, Proceedings*, volume 4475 of *Lecture Notes in Computer Science*, pages 30–39. Springer, 2007. doi:10.1007/978-3-540-72914-3_5.

3 Erich Friedman. Corral puzzles are np-complete. *Technical Report*, 2002. URL: https://erich-friedman.github.io/papers/corral.pdf.

4 Markus Holzer, Andreas Klein, Martin Kutrib, and Oliver Ruepp. Computational complexity of NURIKABE. *Fundam. Informaticae*, 110(1-4):159–174, 2011. doi:10.3233/FI-2011-534.

5 Ayaka Ishibashi, Yuichi Sato, and Shigeki Iwata. Np-completeness of two pencil puzzles: Yajilin and country road. *UTILITAS MATHEMATICA*, 88:237–246, July 2012.

6 Chuzo Iwamoto. Yosenabe is np-complete. *J. Inf. Process.*, 22(1):40–43, 2014. doi:10.2197/ipsjjip.22.40.

7 Chuzo Iwamoto, Masato Haruishi, and Tatsuaki Ibusuki. Herugolf and makaro are np-complete. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *9th International Conference on Fun with Algorithms, FUN 2018, June 13-15, 2018, La Maddalena, Italy*, volume 100 of *LIPIcs*, pages 24:1–24:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.FUN.2018.24.

8 Chuzo Iwamoto and Tatsuaki Ibusuki. Dosun-fuwari is np-complete. *J. Inf. Process.*, 26:358–361, 2018. doi:10.2197/ipsjjip.26.358.

9 Chuzo Iwamoto and Tatsuya Ide. Moon-or-sun, nagareru, and nurimeizu are np-complete (in japanese). In *Winter LA Symposium 2019*, 2019.

10 Chuzo Iwamoto and Tatsuya Ide. Nurimisaki and sashigane are np-complete. In Zachary Friggstad and Jean-Lou De Carufel, editors, *Proceedings of the 31st Canadian Conference on Computational Geometry, CCCG 2019, August 8-10, 2019, University of Alberta, Edmonton, Alberta, Canada*, pages 184–194, 2019.

11 Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato. Frontier-based search for enumerating all constrained subgraphs with compressed representation. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 100-A(9):1773–1784, 2017. doi:10.1587/transfun.E100.A.1773.

12 Jun Kawahara, Toshiki Saitoh, Hirofumi Suzuki, and Ryo Yoshinaka. Colorful frontier-based search: Implicit enumeration of chordal and interval subgraphs. In Ilias S. Kotsireas, Panos M. Pardalos, Konstantinos E. Parsopoulos, Dimitris Souravlias, and Arsenis Tsokas, editors, *Analysis of Experimental Algorithms - Special Event, SEA^2 2019, Kalamata, Greece, June 24-29, 2019, Revised Selected Papers*, volume 11544 of *Lecture Notes in Computer Science*, pages 125–141. Springer, 2019. doi:10.1007/978-3-030-34029-2_9.

13 Graham Kendall, Andrew J. Parkes, and Kristian Spoerer. A survey of np-complete puzzles. *J. Int. Comput. Games Assoc.*, 31(1):13–34, 2008.

14 Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In Alfred E. Dunlop, editor, *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993*, pages 272–277. ACM Press, 1993. doi:10.1145/157485.164890.

15 Yu Nakahata, Jun Kawahara, Takashi Horiyama, and Shin-ichi Minato. Implicit enumeration of topological-minor-embeddings and its application to planar subgraph enumeration. In M. Sohel Rahman, Kunihiko Sadakane, and Wing-Kin Sung, editors, *WALCOM: Algorithms and Computation - 14th International Conference, WALCOM 2020, Singapore, March 31 - April 2, 2020, Proceedings*, volume 12049 of *Lecture Notes in Computer Science*, pages 211–222. Springer, 2020. doi:10.1007/978-3-030-39881-1_18.

16 Nikoli Co., Ltd. Puzzles: Nagareru [Nikoli]. *Available online*, 2021. https://www.nikoli.co.jp/en/puzzles/nagareru.html, (accessed on 20th April 2021).

17 Hirofumi Suzuki, Sun Hao, and Shin-ichi Minato. Generating all solutions of minesweeper problem using degree constrained subgraph model. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 356. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2016.

18 Naoyuki Tamura. Solving Puzzles with Sugar Constraint Solver (in Japanese). *Available online*, 2013. https://cspsat.gitlab.io/sugar-puzzles/, (accessed on 20th April 2021).

19 Naoyuki Tamura. Solving Slither Link Puzzles with Sugar Constraint Solver (in Japanese). *Available online*, 2013. https://cspsat.gitlab.io/sugar-puzzles/slitherlink.html, (accessed on 20th April 2021).

20 Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints An Int. J.*, 14(2):254–272, 2009. doi:10.1007/s10601-008-9061-0.

21 Nobuhisa Ueda and Tadaaki Nagao. Np-completeness results for nonogram via parsimonious reductions. Technical report, Technical Report, TR96-0008, 1996.

22 Gerhard van der Knijff, H Zantema, and JH Geuvers. Solving and generating puzzles with a connectivity constraint. *Bachelor thesis of Radboud University*, 2021.

23 Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 86-A(5):1052–1060, 2003. URL: http://search.ieice.org/bin/summary.php?id=e86-a_5_1052.

24 Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by zdds. *Algorithms*, 5(2):176–213, 2012. doi:10.3390/a5020176.

A Solving blank instances

A Nagareru instance with a small number of white and black cells has a small number of constraints; that is, it has many solutions. For instance, the number of solutions of the $n \times n$ blank instance, which is a grid with no colored cell, is the same as the number of cycles on

the grid that is exponential in n. Whereas it is trivial for humans to find a cycle in the blank instance, it is not trivial for the CSP-based and ZDD-based solvers because, in most cases of CSP, constraints help to reduce the search space. The CPS-based solver has to find a cycle with no additional directional constraint, and the ZDD-based solver has to enumerate all cycles in the grid. Table 3 indicates the computation time of solving the $n \times n$ blank instance ($n = 5, 6, \ldots, 14$) of the CSP-based and the ZDD-based solvers. Compared to Table 1 in our main manuscript, the computation time of the blank instance is relatively more significant than that of an interesting instance. Our generator constructs the ZDD of the blank instance in the first step, which accounts for a large part of the total generation time. Thus, we can quickly scale up our generator by initializing the grid with many colored cells.

Table 3 The computation time of solving the $n \times n$ blank instance of the CSP-based and the ZDD-based solver. T.O. indicates timeout, meaning that it takes more than 100 seconds.

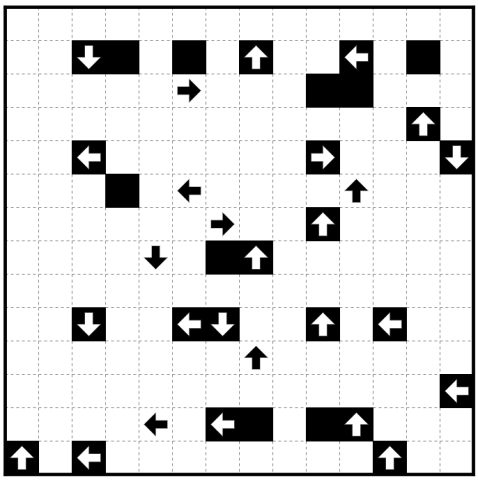
n	CSP	ZDD
5	0.586906	0.005276
6	0.722846	0.006490
7	0.867803	0.009694
8	1.106015	0.020341
9	1.102662	0.056783
10	1.430713	0.172517
11	4.939170	0.544814
12	4.022487	1.838747
13	T.O.	6.246969
14	T.O.	22.105007

B Various interesting instances generated by our Nagareru generator

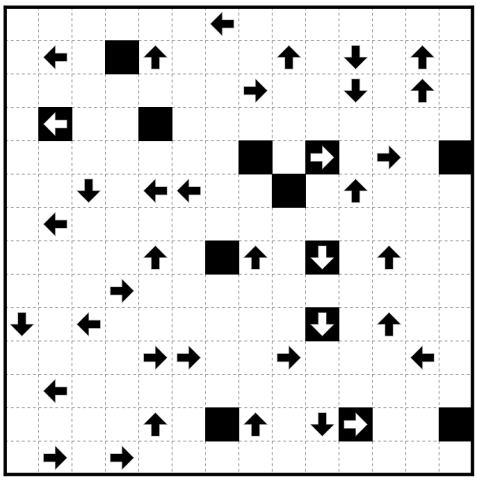
Figure 3 shows a part of interesting instances generated by our generator. Figure 3(a) and (b) have the smallest and the largest number of $|W|$, respectively. Figure 3(c) and (d) have the smallest and the largest number of $|B|$, respectively. Figure 3(e) and (f) have the largest number of $|F|$ and $|B \cup W|$, respectively. Figure 3(c) has also the smallest $|F|$ and $|B \cup W|$.

Figure 3(c) looks easy to solve for humans; however, the CSP-based solver could not solve it in 100 seconds. In this instance, the variables corresponding to the bottom four blank rows are not constrained. We consider that the CSP-based solver wasted much time determining the values of such non-constraint variables.

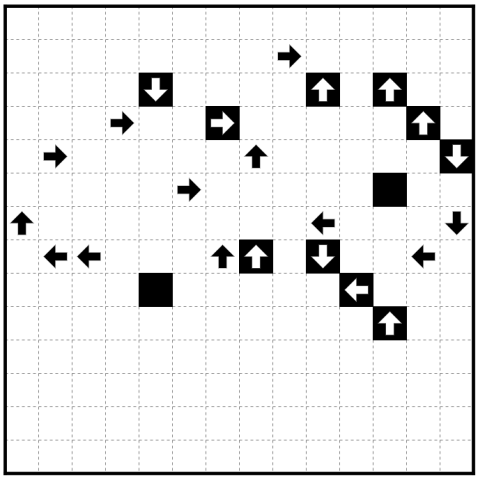
In contrast to Figure 3(c), the other instances seem complex enough for humans to enjoy solving. The first step of solving a Nagareru instance is extending each arrow in each direction of its head and tail by one cell length and creating some line fragments. For example, Figure 3(b) has the largest number of white cells; therefore, the first step can create many long line fragments. On the other hand, Figure 3(a) and (d) have a few white cells, and the first step creates a few short line fragments. However, an instance with many line fragments is not always easy to solve because it is not obvious how to connect them to construct a single consistent cycle. In fact, in Figure 3(b), one has to connect those line fragments carefully to form a single consistent cycle.



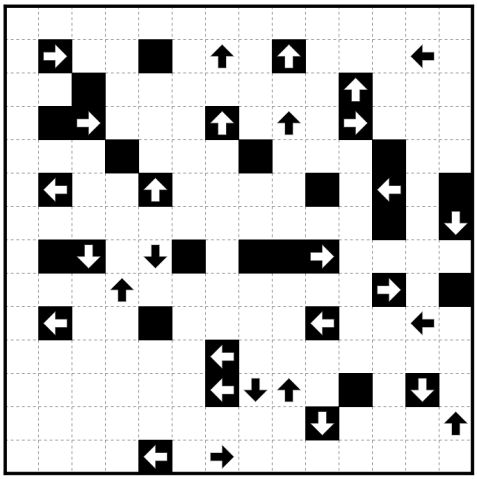
(a) Smallest $|W|$.



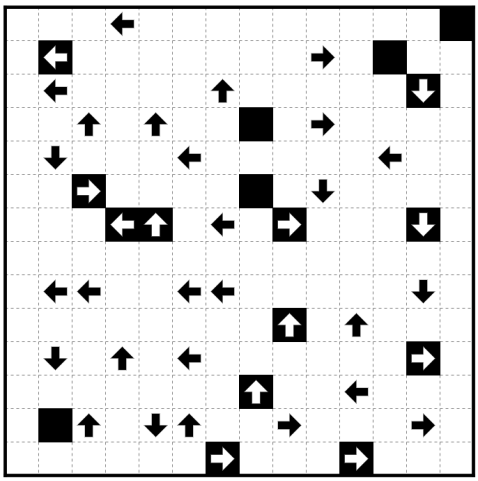
(b) Largest $|W|$.



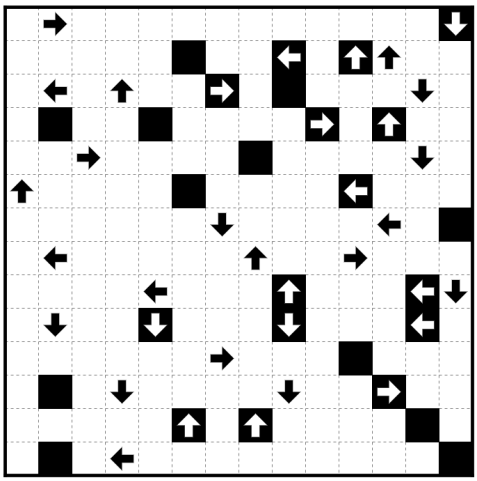
(c) Smallest $|B|$.



(d) Largest $|B|$.



(e) Largest $|F|$.



(f) Largest $|B \cup W|$.

Figure 3 Some examples of 14x14 Nagareru instances generated by our method.

Fast Computation of Shortest Smooth Paths and Uniformly Bounded Stretch with Lazy RPHAST

Tim Zeitz
Karlsruhe Institute of Technology, Germany

Abstract

We study the shortest smooth path problem (SSPP), which is motivated by traffic-aware routing in road networks. The goal is to compute the fastest route according to the current traffic situation while avoiding undesired detours, such as briefly using a parking area to bypass a jammed highway. Detours are prevented by limiting the uniformly bounded stretch (UBS) with respect to a second weight function which disregards the traffic situation. The UBS is a path quality metric which measures the maximum relative length of detours on a path. In this paper, we settle the complexity of the SSPP and show that it is strongly NP-complete. We then present practical algorithms to solve the problem on continental-sized road networks both heuristically and exactly. A crucial building block of these algorithms is the UBS evaluation. We propose a novel algorithm to compute the UBS with only a few shortest path computations on typical paths. All our algorithms utilize Lazy RPHAST, a recently proposed technique to incrementally compute distances from many vertices towards a common target. An extensive evaluation shows that our algorithms outperform competing SSPP algorithms by up to two orders of magnitude and that our new UBS algorithm is the first to consistently compute exact UBS values in a matter of milliseconds.

2012 ACM Subject Classification Theory of computation → Shortest paths; Mathematics of computing → Graph algorithms; Applied computing → Transportation

Keywords and phrases realistic road networks, route planning, shortest paths, traffic-aware routing, live traffic, uniformly bounded stretch

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.3

Supplementary Material *Software (Source Code)*: `https://github.com/kit-algo/traffic_aware` archived at `swh:1:dir:656ed74654a81353c3de75f0a7a79bbc65ec5818`

Acknowledgements I want to thank my colleagues for from the scalable algorithms group Christopher Weyand, Marcus Wilhelm and Thomas Bläsius for pointing out a crucial fact on subpath-optimality at our group workshop. Further, I want to thank my colleague Jonas Sauer for many helpful discussions on algorithmic ideas and proofreading of early drafts of this paper. I also want to thank the anonymous reviewers for their helpful comments. Finally, I would like to thank Jakob Bussas who did a proof-of-concept implementation of the ideas presented here for his bachelor's thesis.

1 Introduction

Over the past years, mobile navigation applications have become ubiquitous. A core feature of these applications is to compute routes between locations in road networks. These routes can be obtained by computing shortest paths on a weighted graph representing the road network with travel times as weights. To present users with *good* routes, it is crucial to take the current traffic situation into account. However, integrating the current traffic situation comes with its own challenges. As traffic feeds are derived from live data, they are inherently noisy and incomplete. Simply exchanging free flow for live traffic travel times and then solving the classical shortest path problem may lead to problematic routes. For example, such routes may include undesired detours such as briefly using a parking area to bypass a jammed highway.

We therefore study an extended problem model, the *shortest smooth path problem* (SSPP) [8]. To avoid undesired detours, a second weight function is taken into account. The first *volatile* weight function models the current traffic situation. The second *smooth* weight function models the free flow travel times and may include additional penalties, for example to avoid residential areas. The goal is to find the shortest path with respect to the volatile weights without too severe detours with respect to the smooth weights.

Related Work. The classical shortest path problem on weighted graphs can be solved with Dijkstra's algorithm [11]. To this day, no asymptotically faster algorithm is known. However, for many practical applications on continental-sized road networks, it is too slow. During the past decade, this has motivated a lot of research effort on engineering faster shortest path algorithms on road networks. Results from this research have played an important role in enabling modern routing applications. By introducing an offline preprocessing phase where auxiliary data is precomputed, queries can be accelerated by more than three orders of magnitude over Dijkstra's algorithm. For an extensive overview on these speed-up techniques, we refer to [2]. One particularly popular technique which we also utilize in this work is *Contraction Hierarchies* (CH) [14]. During preprocessing, additional shortcut arcs are inserted into the graph, which skip over unimportant vertices. On continental-sized road networks with tens of millions of vertices and arcs, CH preprocessing typically takes a few minutes. Shortest path queries can be answered in less than a millisecond. Applying CH to extended problem models is not always trivially possible. We therefore utilize CH indirectly as an A^* [15] heuristic. This approach is called *CH-Potentials* [18]. CH-Potentials is built on *Lazy RPHAST* [18], a CH-based many-to-one algorithm to incrementally compute exact distances from many vertices towards a common target.

So far, research on route planning algorithms and traffic has mostly focused on the interactions between traffic and the preprocessing. For live traffic, speed-up techniques which have two preprocessing phases have been proposed. The first one may be slow but must be independent of any weight function. The second one, called *customization*, typically takes few seconds or less and allows regular traffic updates. *Multi-Level Dijkstra*, commonly referred to as *CRP*, was the first such three-phase technique and has since been extended into a comprehensive framework of routing algorithms [2]. With *Customizable Contraction Hierarchies* (CCH), CH has also been extended to support a three-phase setup [10]. Another line of research studies the integration of predicted traffic [7, 3, 4, 17]. Here, edge weights are functions of the daytime instead of scalar values.

The SSPP was initially introduced by Delling et al. in [8]. The authors discuss the complexity of the problem and show some relations between SSPP and Knapsack but no definitive conclusions could be drawn in their work. The paper also includes two CRP-based algorithms for the SSPP. *Iterative Path Blocking* (IPB) is presented as an exact algorithm for the SSPP. However, it has two issues: First, it takes several seconds even on short-range queries. This makes it unsuitable for practical applications. Second, as we show in this work, it is, in fact, not exact. The authors also present a heuristic algorithm based on the via-node paradigm, i.e. it finds solutions which are concatenations of two shortest paths. It is much faster but may miss promising paths because only via-paths are considered and the UBS is checked heuristically. We are not aware of any other works studying the SSPP.

In the SSPP, limiting the relative length of detours is formalized with the *uniformly bounded stretch* (UBS). The UBS is a path quality measure and quantifies how much longer detours on a path are than their respective fastest alternative. So far, it has been primarily studied in the context of alternative routes [1]. While quite useful, it is expensive to compute

and requires evaluating all subpaths of a path. The authors of [1] state that it would be ideal to check the UBS in time proportional to the length of the path and a few shortest path queries, though they are not aware of any way to do that. To the best of our knowledge, this goal has not been achieved to this day.

Contribution. In Section 3, we settle the complexity of the SSPP by proving that it is strongly NP-complete. Section 4 contains algorithmic results. First, we show that IPB as described in [8] may not find optimal results. Second, we describe necessary adjustments to make it exact. Third, we present an alternative realization based on A* and CH-Potentials [18]. Fourth, we present an efficient algorithm to compute exact UBS values, typically with only a few shortest path queries and in time proportional to the path length as the authors of [1] had hoped for. Fifth, we present Iterative Path Fixing, a new SSPP heuristic. All our algorithms utilize Lazy RPHAST as a crucial ingredient to achieve fast running times. Section 5 contains a thorough evaluation of our algorithms. It clearly shows the effectiveness of our UBS algorithm and our CH-Potentials-based IPB realization, outperforming the state of the art by up to two orders of magnitude.

2 Preliminaries

Let $G = (V, A)$ be a directed graph with $n = |V|$ vertices and $m = |A|$ arcs. We use uv as a short notation for arcs. A *weight function* $w : A \to \mathbb{N}$ maps arcs to positive integers. The *reversed* graph $\overleftarrow{G} := (V, \{vu \mid uv \in A\})$ contains all arcs in reverse direction. The corresponding reversed weight function is $\overleftarrow{w}(vu) := w(uv)$. A sequence of vertices $P = (v_1, \ldots, v_k)$ where $v_i v_{i+1} \in A$ is called a *path*. We denote by $P_{i,j} = (v_i, \ldots, v_j), 1 \leq i < j \leq k$ a *subpath* of P. The length of a path with respect to a weight function w is denoted by $w(P) = \sum w(v_i v_{i+1})$. We refer to a shortest path between two vertices s and t by $\mathrm{OPT}_w(s, t)$ and call its length the *distance* $\mathcal{D}_w(s, t)$ between s and t.

Dijkstra's algorithm [11] computes $\mathcal{D}_w(s, t)$ by traversing vertices by increasing distance from s until t is reached. Vertices are inserted into a priority queue when they are discovered. In each iteration the closest vertex u is popped from the queue and *settled*. Its distance is now final. Outgoing arcs uv are *relaxed*, i.e. the algorithm checks if the path from s to v via u is shorter than the previously known distance from s to v. If this is the case, v will be inserted into the priority queue. To keep track of the best-known distances, the algorithm maintains for each vertex v a tentative distance $\mathtt{D}[v]$. By storing the predecessor vertex on the shortest path from s to v in a parent array $\mathtt{P}[v]$, shortest paths can be efficiently reconstructed. By construction, Dijkstra's algorithm visits all vertices closer to s than the target. The visited vertices are sometimes called the *search space*. It can be reduced with the *A* algorithm* [15] by guiding the search towards the target. Here, the queue is ordered by $\mathtt{D}[v] + h_t(v)$ where h_t is a *heuristic* which estimates $\mathcal{D}(v, t)$.

2.1 (C)CH-Potentials

Contraction Hierarchies (CH) is a two-phase speed-up technique to accelerate shortest path computations on road networks through precomputation. For a detailed discussion we refer to [14]. Here, we only briefly introduce necessary notation and algorithms used in this paper. In a preprocessing phase, vertices are ordered totally by "importance" where more important vertices should lie on more shortest paths. Intuitively, vertices on highways are more important than vertices on some rural street. For CH, such an ordering is obtained heuristically. Then, all vertices are contracted successively by ascending importance. To

contract a vertex means to temporarily remove it from the graph while inserting *shortcut arcs* between more important neighbors to preserve shortest distances among them. The result is an *augmented graph* G^+ with original arcs and shortcuts. G^+ can be split into $G^\uparrow$ and $G^\downarrow$ where $G^\uparrow$ only contains arcs uv where u is less important than v and $G^\downarrow$ vice versa. The augmented graph has the property that for any two vertices s and t, there always exists an *up-down st*-path of shortest distance which first uses only arcs from $G^\uparrow$ and then only arcs from $G^\downarrow$. Such a path can be found by running Dijkstra's algorithm from s on $G^\uparrow$ and from t on the reversed downward graph $\overleftarrow{G^\downarrow}$ graph. Reconstructing the full path without any shortcuts is possible by recursively unpacking shortcuts. For this, one can store for each shortcut the vertex which was contracted when the shortcut was inserted. The set of vertices reachable in $G^\uparrow$ and $\overleftarrow{G^\downarrow}$ is called the *CH search space* of a vertex.

Algorithm 1 Computing the distance from a single vertex u to t with Lazy RPHAST.

Data: $\mathtt{D}^\downarrow[u]$: tentative distance from u to t computed by Dijkstra's algorithm on $\overleftarrow{G^\downarrow}$
Data: $\mathtt{D}[u]$: memoized final distance from u to t, initially $\perp$
Function `ComputeAndMemoizeDist`(u)**:**
  if $\mathtt{D}[u] = \perp$ **then**
    $\mathtt{D}[u] \leftarrow \mathtt{D}^\downarrow[u]$;
    for *all arcs* uv *in* $G^\uparrow$ **do**
      $d \leftarrow$ `ComputeAndMemoizeDist`(v);
      if $d + w(uv) < \mathtt{D}[u]$ **then**
        $\mathtt{D}[u] \leftarrow d + w(uv)$;
  return $\mathtt{D}[u]$;

Lazy RPHAST [18] is a CH-based algorithm to quickly compute distances from many sources to a single target. Lazy RPHAST starts by running Dijkstra's algorithm from t on $\overleftarrow{G^\downarrow}$, similar to a standard CH query. The forward search space, however, is explored through a recursive DFS-like search while memoizing distances to t as depicted in Algorithm 1. This allows reusing the already computed distances for following sources. Lazy RPHAST can be used analogously to compute distances from one vertex to many targets by swapping $G^\uparrow$ and $\overleftarrow{G^\downarrow}$. Using Lazy RPHAST as an A* heuristic is called *CH-Potentials* [18].

Customizable Contraction Hierarchies (CCH) [10] is a three-phase variant of CH. It allows fast updates to the preprocessing, for example to integrate information on the current traffic situation. However, this only affects the preprocessing. The result of the CCH preprocessing is also an augmented graph, only with some additional properties. The CH query algorithms and Lazy RPHAST can be applied without any modification. For the algorithms we discuss in this paper, there is no practical difference between CH and CCH, and we describe our algorithms based on augmented graphs. Our implementation is built on CCH-Potentials to support quick updates to live traffic weights. For a detailed discussion of the differences between CH and CCH and the changes to the preprocessing see [10].

2.2 Smooth Paths

The *stretch* of a path is defined as $S_w(P) = \frac{w(P)}{\mathcal{D}_w(v_1, v_k)}$, i.e. the ratio between the path length and the shortest distance between its endpoints. The *uniformly bounded stretch* $\mathrm{UBS}_w(P) = \max_{0 \le i < j \le k} S_w(P_{i,j})$ indicates the maximum stretch over all subpaths. We observe the following useful property of UBS:

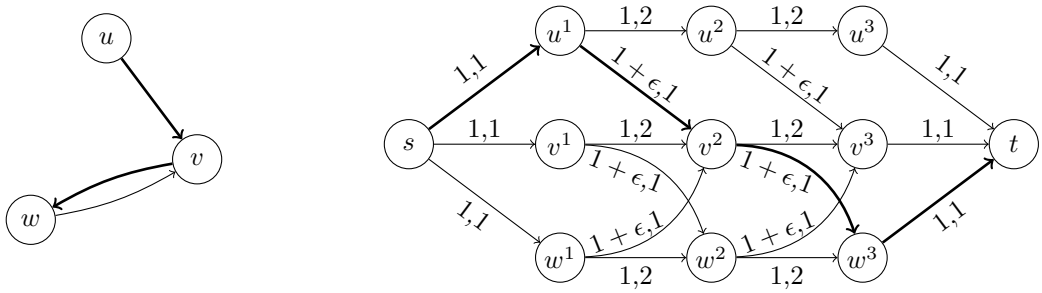


Figure 1 Illustration of our transformation from HamiltonPath to ShortestSmoothPath. The first arc weight is the smooth weight, the second the volatile weight. The thick arcs indicate a Hamiltonian path and the corresponding shortest ϵ-smooth path.

▶ **Observation 1.** *The UBS of a path* $P = (v_1, \ldots, v_i, \ldots, v_j \ldots, v_k)$ *where* $P_{1,i} = \mathrm{OPT}(v_1, v_i)$ *and* $P_{j,k} = \mathrm{OPT}(v_j, v_k)$ *is equal to* $\mathrm{UBS}(P_{i,j})$.

This is because the stretch of any subpath only decreases when appending optimal segments to the beginning or end.

In [8], Delling et al. introduce the shortest smooth path problem (SSPP). A path P is ϵ-smooth with respect to a weight function w when $\mathrm{UBS}_w(P) < 1 + \epsilon$. Given a graph G, vertices s and t, a smooth weight function w and a volatile weight function w^* and a parameter $\epsilon > 0$, the shortest smooth path problem is to find the shortest path with respect to w^* that is ϵ-smooth with respect to w.

3 Complexity

In this section, we prove that SSPP is strongly NP-complete for any $\epsilon \in \mathbb{Q}^{>0}$. We define the decision variant of the problem as follows: An instance (G, w, w^*, s, t, k) of the of ϵ-ShortestSmoothPath-Dec problem admits a feasible solution if and only if there exists a path $P = (s, \ldots, t)$ in G with $w^*(P) \leq k$ and $\mathrm{UBS}_w(P) < 1 + \epsilon$.

▶ **Theorem 2.** *ϵ-ShortestSmoothPath-Dec is strongly NP-complete for any $\epsilon \in \mathbb{Q}^{>0}$.*

Proof. A solution can be verified in polynomial time. Determining the path weight in w^* takes running time linear in $\mathcal{O}(|P|)$. To check the UBS, shortest distances have to be computed for all $\mathcal{O}(|P|^2)$ subpaths. This shows that ShortestSmoothPath-Dec $\in$ NP.

To prove the hardness, we give a reduction from the strongly NP-complete HamiltonPath problem [13]. The goal is to find a *Hamiltonian* path, i.e. a simple path which traverses every vertex exactly once. Let $G = (V, A)$ be the HamiltonPath instance. To distinguish them from the vertices in the SSPP instance, we will denote the vertices in the HamiltonPath instance as *nodes*. We construct the vertices of our SSPP instance by copying each node $|V|$ times (forming $|V|$ layers) and creating two additional vertices s and t. Arcs only connect successive layers. There are arcs between vertices corresponding to the same node and arcs corresponding to arcs from the HamiltonPath instance. Any $(s, \ldots, t)$ path has exactly $|V| + 1$ arcs and has to traverse all layers. We will choose the arc weights in such a way that the shortest ϵ-smooth path between s and t has to use a different node in each layer. Paths using the same node in different layers will always be non-ϵ-smooth in w or too long in w^*.

Formally, we construct the graph $G' = (V', A')$ for our SSPP instance as follows: We set the vertices $V' = \{v^i \mid v \in V, i \in [1, n]\} \cup \{s, t\}$. The arc set A' is the union of three groups of arcs A_{orig}, A_{self} and A_{terminal} where $A_{\text{orig}} = \{(u^i, v^{i+1}) \mid uv \in A, 1 \leq i < n\}$

are the arcs between the layers corresponding to arcs in the HamiltonPath instance, $A_{\text{self}} = \{(v^i, v^{i+1}) \mid v \in V, 1 \leq i < n\}$ are the additional arcs between the same nodes in successive layers and $A_{\text{terminal}} = \{(s, v^1) \mid v \in V\} \cup \{(v^n, t) \mid v \in V\}$ are the arcs connecting the terminals with the first and last layer. In both weight functions, all arcs $a_{\text{term}} \in A_{\text{terminal}}$ get the weight $w(a_{\text{term}}) = w^*(a_{\text{term}}) = 1$. The arcs in $a_{\text{self}} \in A_{\text{self}}$ get a smooth weight $w(a_{\text{self}}) = 1$ and a volatile weight $w^*(a_{\text{self}}) = 2$. The arcs in $a_{\text{orig}} \in A_{\text{orig}}$ get a smooth weight $w(a_{\text{orig}}) = 1 + \epsilon$ and a volatile weight $w^*(a_{\text{orig}}) = 1$. Setting $k = |V| + 1$, this forms our ShortestSmoothPath-Dec instance. This transformation has a running time in $\mathcal{O}(n \cdot (n + m))$. See Figure 1 for an illustrated example of the construction. For the sake of readability, we use non-integer weights of $1 + \epsilon$ in this proof. The weights can be turned into integers by multiplying them with the denominator of ϵ.

Now, assume that the HamiltonPath instance admits a Hamiltonian path $P = (v_1, \ldots, v_n)$. Then, $P' = (s, v_1^1, \ldots, v_n^n, t)$ is a solution to the SSPP problem. The path uses two arcs from A_{terminal} and $|V| - 1$ arcs from A_{orig}. Thus, $w^*(P') = |V| + 1 = k$. Also, its $\text{UBS}_w(P')$ must be smaller than $1 + \epsilon$. Due to Observation 1, it is sufficient to show that the UBS is small enough for $P'' = (v_1^1, \ldots, v_n^n)$. For any subpath $P''_{i,j} = (u^i, \ldots v^j)$ for $i < j$, u must not be equal to v because P is a Hamiltonian path. As all arcs are from A_{orig}, $w(P''_{i,j}) = (j - i) \cdot (1 + \epsilon)$. The shortest path (with respect to w) between u^i and v^j has to use at least one A_{orig} arc because $u \neq v$. Thus, $\mathcal{D}_w(P''_{i,j}) \geq (j - i - 1) + (1 + \epsilon)$. This yields

$$\text{UBS}_w(P''_{i,j}) = \frac{w(P''_{i,j})}{\mathcal{D}_w(P''_{i,j})} \leq \frac{(j-i)\cdot(1+\epsilon)}{j-i+\epsilon} < \frac{(j-i)\cdot(1+\epsilon)}{j-i} = 1 + \epsilon$$

which proves that P' is a valid solution for the SSPP instance.

Conversely, suppose that our SSPP instance has an ϵ-smooth path $P' = (s, v_1^1, \ldots, v_n^n, t)$ of weight $w^*(P') = |V| + 1$. Such a path cannot contain any arcs from A_{self} because their volatile weight is 2. We now show that no two vertices in the path can correspond to the same node and thus that $P = (v_1, \ldots, v_n)$ is indeed a Hamiltonian path in G. Suppose for contradiction that $P'_{i,j} = (v^i, \ldots, v^j)$ was a subpath of P'. The length $w(P'_{i,j})$ is $(i-j)\cdot(1+\epsilon)$. Since start and end vertex correspond to the same node, the shortest path with respect to w between these vertices is made up of arcs from A_{self} and has distance $\mathcal{D}_w(v^i, v^j) = i - j$. Thus, $\text{UBS}_w(P'_{i,j}) = (1 + \epsilon)$ which means that this subpath must not be part of a solution for the SSPP instance. This is a contradiction. Thus, the SSPP solution induces a valid solution for the HamiltonPath instance. ◀

4 Algorithms

In [8], the *Iterative Path Blocking* (IPB) algorithm is proposed to solve the SSPP optimally. The algorithm repeats two steps until a valid path is found. It maintains a set of blocked paths, which is initially empty. In the first step, a shortest path with respect to w^* is computed while avoiding any blocked paths. In the second step, the obtained path is checked for subpaths violating the UBS constraint. Any violating subpaths are added to the list of blocked paths and the algorithm continues with the next iteration. If no violating subpath is found, the final path is returned.

This framework can be implemented with different concrete algorithms for both steps. The implementation described in [8] is based on CRP [5]. In this paper, we propose optimized implementations for both steps based on Lazy RPHAST and (C)CH-Potentials.

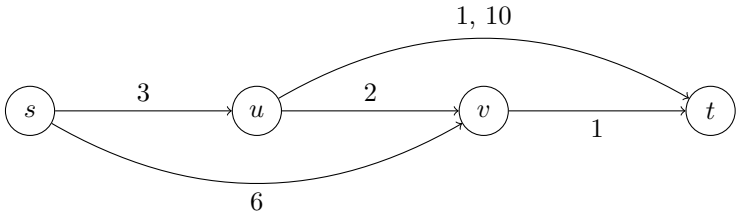


Figure 2 Example graph where for $\epsilon = 1$ the shortest ϵ-smooth path (s, v, t) is not prefix-optimal. For all arcs except ut, the smooth and the volatile weight function are equal. For ut, the smooth weight is 1 and the volatile weight 10.

4.1 Avoiding Blocked Paths

The authors of [8] describe their approach to the first phase as a variant of Dijkstra's algorithm. When relaxing an arc uv where v is the endpoint of a blocked path, they backtrack the parent pointers of v, comparing the reconstructed path to the blocked path. Should the paths match, the search is pruned at v. This algorithm correctly avoids blocked paths. However, it also avoids some additional paths because Dijkstra's algorithm by construction only finds prefix-optimal paths. But optimal shortest smooth paths may not be prefix-optimal with respect to the volatile weight function w^*. See Figure 2 for an example. To the best of our understanding, IPB as described in [8] will not find the shortest smooth path in this example. The algorithm will find the path (s, u, v, t) in the first iteration. This path is not 1-smooth because (u, v, t) has stretch 3 and (u, v, t) will be added to the blocked path set. With (u, v, t) blocked, the algorithm will find the path (s, u, t) in the next iteration and return it as the final result. However, the shortest 1-smooth path is (s, v, t). It was missed because the prefix (s, v) is not optimal in w^* and was therefore pruned at v by (s, u, v). We will refer to this variant from now on by *heuristic iterative path blocking* (IPB-H). IPB-H will still find an ϵ-smooth path though it may not necessarily be the shortest.

To find shortest smooth path with Dijkstra's algorithm, we need to adjust the notion of optimality used to compare labels. It might be necessary to keep a label with suboptimal distance from the start as in the example from Figure 2 where the label for (s, v) needs to be kept at v despite being longer than (s, u, v). This leads to a *label-correcting* variation of Dijkstra's algorithm with possibly multiple labels per vertex. A *label* l at a vertex v consists of a distance $\mathtt{D}_l$ from the source, a set of active blocked paths $\mathtt{A}_l$, and a pointer to the parent vertex and label for efficient reconstruction of the labels' path $P(l) = (s, \dots, v)$. The active blocked path set $\mathtt{A}_l$ contains all blocked paths which have a prefix which is a suffix of $P(l)$. A label l can be discarded when v has another label l' with $\mathtt{D}_{l'} \leq \mathtt{D}_l$ and $\mathtt{A}_{l'} \subseteq \mathtt{A}_l$.

The search is initialized with a single label at s with distance zero and an empty set of active blocked paths. When a vertex u with a label l_u is popped from the queue and an arc uv is relaxed, we create a new label l_v as follows: We set the distance $\mathtt{D}_{l_v} = \mathtt{D}_{l_u} + w^*(uv)$ and the parent label to l_u. We also need to keep track of traversed blocked paths. If uv is the first arc of a blocked path $B = (u, v, \dots)$, the path B needs to be added to $\mathtt{A}_{l_v}$. For any active blocked path $B = (\dots, u, x, \dots) \in \mathtt{A}_{l_u}$, we need to check if $x = v$, i.e. uv lies on B. If this is the case, B is contained in $\mathtt{A}_{l_v}$, or, if uv is the last arc of B, the label l_v must be dropped. If uv is not on B, the blocked path is not in $\mathtt{A}_{l_v}$.

An efficient implementation of this algorithm requires careful engineering. For each arc, we keep track of the blocked paths it lies on. Labels use a bitset to store the active blocked paths. This allows for efficient subset checks with bit-wise operations. The bitset size is

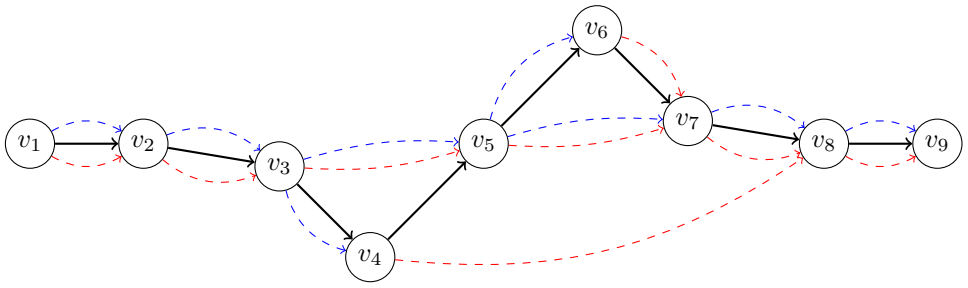


Figure 3 Example path (solid, black) with shortest path tree from v_1 to all vertices on the path (dashed, blue) and reverse shortest path tree from all vertices on the path to v_9 (dashed, red).

determined individually for each vertex by the number of blocked paths the respective vertex lies on. In our implementation, we use at least one 128-bit integer which suffices for most queries. Should the number of blocked paths for a vertex exceed 128, we switch to using a dynamically sized array of integers for that vertex. Additionally, each vertex maintains its own queue of labels ordered by distance from s. When the vertex is popped from the queue, it pops the next label from its queue and propagates only this label. If there are any remaining labels in the queue, the vertex is reinserted into the global queue. Finally, we utilize A* with CCH-Potentials on the volatile weight function to guide the search towards the target. As our experiments show, disallowing non-ϵ-smooth paths increases distances only very slightly. Thus, the heuristic is close to perfect and A* very effective for this problem.

4.2 Efficient UBS Computation

According to Delling et al. [8], the UBS computation is one of the bottlenecks of the IPB approach. They employ a many-to-many algorithm. Here, we introduce an algorithm which can compute exact UBS values of typical paths with only a few shortest path queries. We also present a worst-case example where each subpath has a distinct stretch value. This suggests that achieving subquadratic worst-case running time may not be possible.

Consider a path $P = (s = v_1, \ldots, t = v_k)$ as depicted in Figure 3. Our algorithm works iteratively. We start with the full path and successively remove prefixes and suffixes until the path is empty or only a shortest path remains. We start by computing shortest distances from s to all vertices on the path. This can be done with a single run of Dijkstra's algorithm which can terminate once all v_i have been settled. Beside the shortest distances, this yields a shortest path tree represented though parent pointers. We also run Dijkstra's algorithm from t on the reversed graph which yields a backward shortest path tree to t. Now we find the greatest index i such that $P_{1,i}$ is a prefix of all shortest paths $\mathrm{OPT}(s, v_l)$ where $i < l \leq k$, i.e. the first branching vertex in the forward shortest path tree. In the worst case this may be s. In the example in Figure 3 this is v_3. We analogously obtain the first branching vertex in the reverse shortest path tree to v_k (v_8 in our example). Stated formally, this is the smallest index j such that $P_{j,k}$ is a suffix of all shortest paths $\mathrm{OPT}(v_l, t)$ where $1 \leq l \leq j$. By Observation 1, subpaths starting from vertices in the segment $P_{1,i-1}$ and subpaths ending at vertices from $P_{j+1,k}$ are not relevant to the UBS computation. We exploit this and only check paths starting from v_i or ending at v_j in the current iteration.

We check the stretch of all subpaths $P_{i,l}$ where $i < l \leq j$ with a linear sweep over the v_l. Since $P_{1,i}$ is a prefix of all shortest paths from s, we can compute the distance $\mathcal{D}(v_i, v_l)$ as $\mathcal{D}(s, v_l) - \mathcal{D}(s, v_i)$. Thus, each stretch can be checked in constant time with the distances computed by Dijkstra's algorithm. When we are only interested in violating

Algorithm 2 Path unpacking for Lazy RPHAST.

Data: $\texttt{P}[u]$: parent vertex on the shortest path from u to t, as computed by Dijkstra's algorithm on $\overleftarrow{G^{\downarrow}}$ and an extended Algorithm 1
Data: $\texttt{U}[u]$: whether the path from u to t has been fully unpacked
Function $\texttt{Unpack}(u)$**:**
  if $\neg(\texttt{U}[u] \vee u = t)$ **then**
    $\texttt{ComputeAndMemoizeDist}(u)$;
    $\texttt{Unpack}(\texttt{P}[u])$;
    if $(u, \texttt{P}[u])$ *is a shortcut for* $(u, v, \texttt{P}[u])$ **then**
      $\texttt{P}[v] \leftarrow \texttt{P}[u]$;
      $\texttt{Unpack}(v)$;
      $\texttt{P}[u] \leftarrow v$;
      $\texttt{Unpack}(u)$;
    $\texttt{U}[u] \leftarrow$ **true**;

subpaths (rather than computing the exact UBS value of P), the sweep can be stopped after the first (i.e. shortest) violating segment has been found. Forbidding the shortest violating segment starting at v_i is sufficient because it is contained in all longer segments. Checking the stretches of the subpaths $P_{l,j}$ where $i \leq l < j$ works analogously.

Having checked all these stretches, we continue with the next iteration by applying the whole algorithm to the subpath $P_{i+1,j-1}$. We can stop when $i+1 \geq j-1$ or when the entire considered path is a shortest path between its endpoints.

This algorithm can be adopted to efficiently compute other path quality measures such as the *local optimality* [1].

4.2.1 Worst-Case Running Time

This algorithm performs great when long segments are shortest paths, which will often be the case when searching shortest smooth paths. But in the worst case, it still has to check $\Theta(n^2)$ subpath stretches. Consider a complete graph with unit weights and the path $P = (v_1, \ldots, v_n)$. In this graph, the shortest path between any two vertices is always the direct arc and the distance is exactly one. Thus, the shortest path tree from any vertex is a star with the direct arcs and our algorithm can only advance by a single vertex in each iteration. This results in a worst case running time of n runs of Dijkstra's algorithm.

We suspect that it is not possible to compute the UBS asymptotically faster. Consider the same graph as before but with weights of unique powers of two for the arcs of the path. Now any subpath has a unique length. As all subpaths of three or more vertices still have a shortest distance of one between their endpoints, there are $\Theta(n^2)$ unique stretch values. Thus, computing the UBS of the whole path without checking all $\Theta(n^2)$ stretch values should be difficult if not impossible.

4.2.2 Lazy RPHAST with Path Unpacking

While this algorithm typically needs few stretch checks, running Dijkstra's algorithm a couple of times is still prohibitively slow on large road networks. Luckily, we can speed these computations up drastically by employing Lazy RPHAST, which we already used as an A* heuristic in the shortest path search phase. Recall that Lazy RPHAST allows us to select

one target vertex and then to compute shortest distances quickly from many vertices to this target. For the efficient UBS computation, we use two instantiations of this algorithm. In each iteration, we select both endpoints of the considered path and compute distances from and to the endpoints for all vertices on the path. However, we also need the shortest path trees. We therefore extend Lazy RPHAST to also compute shortest path trees.

Dijkstra's algorithm on $\overleftarrow{G^{\downarrow}}$ yields initial parent pointers. We adjust Algorithm 1 to continue to maintain these parent pointers during arc relaxation. Thus, after having called `ComputeAndMemoizeDist`, we have the shortest path through the CH search space in G^+. Algorithm 2 depicts the routine to efficiently unpack shortcuts on this path and retrieve shortest path trees in the original graph. We use a bitvector `U` (using a clearlist for fast reinitialization) to mark vertices for which the shortest path has already been fully unpacked which is checked before any actual work is performed. Then, we have to call `ComputeAndMemoizeDist` to ensure that the path through the CH search space has been obtained for u. For vertices encountered through recursive shortcut unpacking this might have not happened before. In the next step, we can now recursively unpack the full path up to the parent $\texttt{P}[u]$ of our current vertex u. Now, all that remains is to unpack the arc $(u, \texttt{P}[u])$ if it is a shortcut. If so, the middle vertex v is set in `P` as the vertex between u and $\texttt{P}[u]$ and `Unpack` is invoked recursively first for v and then again for u to unpack the arcs $(v, \texttt{P}[v])$ and (u, v).

4.3 Iterative Path Fixing

With an efficient algorithm to find UBS-violating segments we can introduce another natural heuristic to find short smooth paths: Find the shortest path with respect to w^* and replace each UBS violating subpath $(v_i, \ldots, v_j)$ with $\text{OPT}_w(v_i, v_j)$. The result may still contain UBS violating subpaths. In this case, we iteratively continue to replace violating segments. When a path contains overlapping violating subpaths, we replace the first, ignore following overlapping subpaths and continue with the next non-overlapping segment. We denote this algorithm as *iterative path fixing* (IPF).

5 Evaluation

In this section, we present our experimental results. Our benchmark machine runs openSUSE Leap 15.3 (kernel 5.3.18), and has 192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each of which has eight cores clocked at 3.5 GHz and 8 × 64 KiB of L1, 8 × 1 MiB of L2, and 24.75 MiB of shared L3 cache. All running times are sequential. We implement our algorithms in Rust[1] and compile them with `rustc 1.58.0-nightly (b426445c6 2021-11-24)` in the release profile with the `target-cpu=native` option.

Table 1 shows the road networks we use in our experiments alongside sequential preprocessing times. OSM Europe is the same network used in [8] and publicly available.[2] The DIMACS Europe instance was made available by PTV[3] for the 9th DIMACS implementation challenge [9]. It is not publicly available but can be obtained on request for research purposes[4]. We derived the OSM Germany instance from an early 2020 snapshot of OpenStreetMap and

[1] The code for this paper, all implemented algorithms, scripts to perform experiments and to aggregate the results is available at `https://github.com/kit-algo/traffic_aware`
[2] `https://i11www.iti.kit.edu/resources/roadgraphs.php`
[3] `https://ptvgroup.com`
[4] `https://i11www.iti.kit.edu/resources/roadgraphs.php`

Table 1 Instances used in the evaluation with sequential preprocessing running times to construct a CCH-Potential. Phase 1 needs to be run only once for each graph, Phase 2 once for each weight function, or when a weight function changes.

	Vertices	Edges	Preprocessing [s]	
	$[\cdot 10^6]$	$[\cdot 10^6]$	Phase 1	Phase 2
DIMACS Europe	18.0	42.2	2 260.7	11.3
OSM Europe	173.8	348.0	4 270.0	58.8
OSM Germany	11.1	26.2	1 314.0	7.5

converted into a routing graph using RoutingKit.[5] For this instance, we have proprietary traffic data provided by Mapbox[6] which, unfortunately, we cannot provide. The data includes two live traffic snapshots in the form of OSM node ID pairs and live speeds for the edge between the vertices. One is from Friday 2019/08/02 afternoon and contains 320K vertex pairs and the other from Tuesday 2019/07/16 morning and contains 185K pairs. For both Europe instances, we do not have any real world traffic data. Thus, we resort to the approach suggested in [8] and generate synthetic live traffic: For each road where the average speed is greater than 30 kph, we reduce the speed to 5 kph with a probability of 0.5%.

We evaluate our algorithms by performing batches of point-to-point shortest ϵ-smooth path queries. As the distance between source and target has a significant influence on the performance, we generate different query batches. For each batch, we pick 1000 source vertices uniformly at random. We then run Dijkstra's algorithm from each source vertex on the graph with the smooth weight function. Following the Dijkstra rank methodology, we store every 2^ith settled vertex [16]. This allows evaluating the performance development against varying path lengths. In [8], 1-hour queries were performed. For comparison, we also generate an *1h* batch by picking the first settled vertex with a distance greater than one hour. In addition, we generate a *4h* batch for medium-range queries with the same method and a *random* batch for long range queries where the target is picked uniformly at random.

Preliminary experiments showed that some queries take prohibitively long to answer. Since we are solving an NP-hard problem, this is not very surprising. We abort queries if the algorithm has not found a path with UBS $< (1 + \epsilon)$ after 10 seconds. We report these queries as failed but, nevertheless, do include their running times in our measurements.

We start by evaluating different UBS algorithms in isolation. The paths checked by the UBS algorithms are the paths we find while performing IPB-H to find shortest smooth paths with $\epsilon = 0.2$ on the Dijkstra rank queries. We limit the time per rank and UBS algorithm to one hour. Thus, slow algorithms may not get to check all paths. Our baseline is computing all distances between pairs of path vertices at once with *SSE RPHAST* [6], which to the best of our knowledge is the fastest known many-to-many algorithm. The second algorithm, denoted as *Lazy RPHAST Naive*, uses Lazy RPHAST to compute distances between all pairs of path vertices. The third one is *UBS Trees Dijkstra* which is the non-accelerated, i.e. Dijkstra-based, implementation of the efficient UBS algorithm introduced in Section 4.2. *UBS Trees Lazy RPHAST* denotes the accelerated variant of this algorithm utilizing Lazy RPHAST as described in Section 4.2.2.

[5] `https://github.com/RoutingKit/RoutingKit`
[6] `https://mapbox.com`

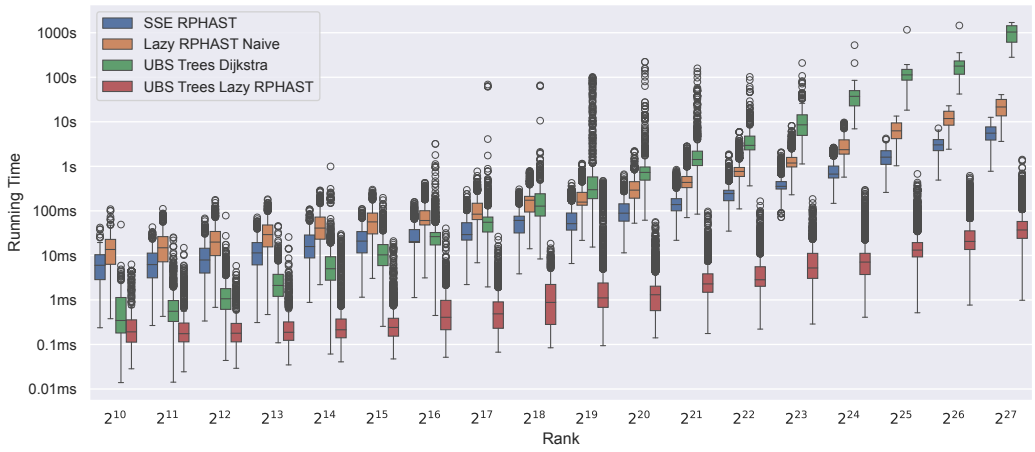


Figure 4 Running times of different UBS checking algorithms for paths encountered by IPB-H when answering queries of different ranks with $\epsilon = 0.2$ on OSM Europe. The boxes cover the range between the first and third quartile. The band in the box indicates the median, the whiskers cover 1.5 times the interquartile range. All other running times are indicated as outliers.

Figure 4 depicts the results of this experiment. We observe that SSE RPHAST is consistently faster than the naive Lazy RPHAST variant by a roughly constant factor. SSE RPHAST was designed as a many-to-many algorithm and is thus more efficient than naively applying a many-to-one algorithm $|P|$ times. The non-accelerated UBS Trees algorithm is very fast for short paths but quickly becomes prohibitively slow for longer paths. Running Dijkstra's algorithm will traverse a large part of the network if source and target are sufficiently far apart from each other. Doing this multiple times is not feasible. However, the accelerated variant beats SSE RPHAST by about two orders of magnitude across all path lengths.

UBS Trees running times have significantly greater variance than the many-to-many algorithms. This is because the amount of work which UBS Trees can avoid varies strongly between different paths. In contrast, the many-to-many-based algorithms will always check $\Theta(|P|^2)$ subpath distances. Note that the UBS Trees Dijkstra outliers disappear because we limit the time per rank and algorithm. If we checked all paths, the outliers would be present too, but the experiment would take prohibitively long.

Next, we evaluate the performance of our query algorithms on realistic queries and instances. Table 2 depicts the results. Both the query set and the instance have a strong influence on the running time. Note that random queries on OSM Germany are on average shorter than four hours which is the reason why the running times on OSM Germany for random queries are faster than for 4h queries. The length increase of the solutions primarily depends on the instance and less on the query set. The synthetic traffic affects DIMACS Europe more strongly than OSM Europe. We suspect that this is because OSM is modeled in much greater detail and contains more shorter arcs. In terms of running time, IPB-H is significantly faster than IPB-E and IPF is significantly faster still, which is roughly what we expected. Conversely, the heuristics find somewhat longer paths than the exact IPB-E algorithm and IPF appears to find worse paths than IPB-H. However, one has to be careful interpreting these numbers as a non-negligible amount of queries did not terminate with IPB-E and IPB-H. Because the length increase numbers are averages over different sets, it is not immediately clear if the differences appear because the heuristics find worse paths or because the heuristics find long solutions where the exact algorithm did not finish within

Table 2 Average performance of our implementations of IPB-E, IPB-H and IPF for different query sets on all instances with $\epsilon = 0.2$. The Increase column denotes the length increase with respect to w^* of the obtained path over OPT_{w^*} and includes only successful queries. The running time column also includes the running time of queries aborted after 10 seconds.

		Increase [%]			Running time [ms]			Failed [%]		
		IPB-E	IPB-H	IPF	IPB-E	IPB-H	IPF	IPB-E	IPB-H	IPF
1h	DIMACS Eur Syn	0.8	2.5	4.1	718.0	168.4	1.5	6.4	1.6	0.0
	OSM Eur Syn	0.2	0.3	0.3	59.8	22.4	2.7	0.4	0.2	0.0
	OSM Ger Fri	0.2	1.5	2.2	1 373.7	219.1	4.7	12.7	1.2	0.0
	OSM Ger Tue	0.1	0.3	0.4	261.5	9.2	1.8	2.2	0.0	0.0
4h	DIMACS Eur Syn	0.8	3.4	5.1	3 513.6	435.4	6.1	33.1	3.1	0.0
	OSM Eur Syn	0.2	0.3	0.3	331.4	73.2	8.4	2.1	0.6	0.0
	OSM Ger Fri	0.2	2.1	4.2	6 597.1	2 568.1	89.2	63.4	15.8	0.0
	OSM Ger Tue	0.1	0.4	0.5	1 449.3	93.1	9.8	13.1	0.0	0.0
Random	DIMACS Eur Syn	0.8	2.8	5.3	6 700.6	2 436.7	30.6	64.2	17.1	0.0
	OSM Eur Syn	0.2	0.4	0.4	4 758.3	654.2	140.3	38.9	3.1	0.0
	OSM Ger Fri	0.2	2.1	4.1	5 771.1	2 419.8	84.5	56.0	16.3	0.0
	OSM Ger Tue	0.1	0.4	0.5	1 366.0	111.6	9.6	12.0	0.0	0.0

10 seconds. For running times, the averages are also difficult to interpret. They are heavily skewed by outliers and there is no reason to assume a normal distribution. In fact, *median* running times for 1h queries of all algorithms on all instances are all below 2 ms. Clearly, drawing statistically sound conclusions from this experiment requires a closer look.

Figure 5 depicts *performance profiles* [12] for running times and obtained path lengths on all queries from Table 2 combined. Investigating queries across all instances combined is reasonable because we study the relative performance of the different algorithms on each query. Let $\mathcal{A}$ be the set of algorithms, $\mathcal{Q}$ the set of queries and $\text{obj}(a, q)$ denote the considered measurement from the computation of $a \in \mathcal{A}$ to answer $q \in \mathcal{Q}$. In our case, this is either the running time or the length with respect to w^* of the computed path. The performance ratio $r(a, q) = \frac{\text{obj}(a,q)}{\min\{\text{obj}(a',q) | a' \in \mathcal{A}\}}$ indicates by what factor a deviates from the best solution or the shortest running time for the query q. The performance profile $\rho_a : [1, \infty) \to [0, 1], \tau \mapsto \frac{|\{q \in \mathcal{Q} | r(a,q) \leq \tau\}|}{|\mathcal{Q}|}$ of a is the fraction of queries for which a is within a factor of τ of the best measurement. For computations that were aborted after 10 seconds, $\text{obj}(a, q) = \infty$. For the sake of completeness, we also include the same performance profiles separated per instance and query set in the appendix (see Figure 6 and 7). However, discussing the results in such detail is beyond the scope of this paper.

The running time performance profile in Figure 5 allows for some more nuanced observations: IPF is the fastest algorithm on about 65% of the queries and almost never more than 10 times slower than the fastest one. Surprisingly, IPB-H is also sometimes the fastest to answer a query (in 35% of the queries) but it may also be up to 300 times slower than the fastest algorithm. However, for 83% of all queries it stays within a factor of 10. The exact algorithm is never the fastest but still within a factor of 10 for 65% of the queries. It still may be several thousand times slower than the fastest algorithm in extreme cases, even with the running time limited to 10 seconds.

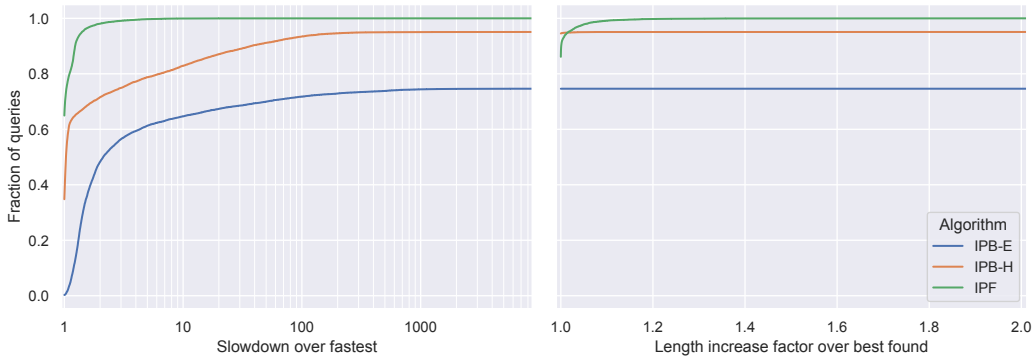


Figure 5 Relative performance profiles of our algorithms on all queries from Table 2.

The path length performance profile also yields useful insights. Since IPB-E is an exact algorithm, its performance profile contains only a single data point, i.e. for all queries which terminated successfully IPB-E finds the shortest path. The line for IPB-H is almost constant. This means, there are only few queries where it does not find the best solution. Even when it does not find the best solution, it is close to the best one, i.e. the maximum length increase factor over the best solution is 1.36 and all other values are below 1.1. It is quite possible that IPB-H found the optimal solution even for some queries where IPB-E did not terminate. The qualitative performance of IPF varies more strongly. It also finds the best solution on 85% of the queries. More than 99% of the obtained solutions are within a factor of 1.2 to the best found. In the worst case IPF found a path 1.96 times the length of the best one found by another algorithm.

In combination with the averages reported in Table 2 we can now draw solid conclusions on the performance of the algorithms. IPF is the algorithm with the most stable running time. Even though it is not always the fastest, it is never much slower than any other algorithm. It is the only algorithm able to answer all queries in less than 10 seconds. In fact, it usually needs only a few milliseconds and only up to several hundreds of milliseconds for extreme cases. It sometimes pays for this with worse solution quality but is still very close to the best found for the vast majority of queries. This makes it an algorithm suitable for practical applications. IPB-H is also a very effective heuristic. It is drastically faster than the exact algorithm and sometimes even faster than IPF. Its performance in terms of quality is much more stable than IPF and often IPB-H will find the best path or something very close to it. The difference in average length increase between IPB-E and IPB-H was not because IPB-H finds much worse paths but because it is able to answer queries which IPB-E cannot answer. However, it still fails to answer about 5% of all queries in less than 10 seconds. The running time of IPB-E varies even more strongly. On the one hand, many easy queries can be answered in a few milliseconds, but on the other hand, 25% of all queries cannot be answered in less than 10 seconds. The feasibility of solving the problem to exactness with IPB-E strongly depends on the distance of queries and on the smoothness of w^*.

For our final experiment, we evaluate the performance of our algorithms with different choices for ϵ with 1000 queries of 1h range on OSM Europe. Table 3 depicts the results. This experiment was also performed in [8] but with only 100 queries. Given the observation from the previous experiment, it should be clear that reported averages allow only for very rough comparisons. However, it is the only data available to compare against related work. Also

Table 3 Average performance of our implementations of IPB-E, IPB-H and IPF for different values of ϵ with 1h queries on OSM Europe with synthetic live traffic. The Increase column denotes the length increase with respect to w^* of the shortest smooth path over the shortest w^* path. It includes only values from successful queries. All other columns indicate average values over all queries, including the ones terminated after 10 seconds.

		Increase	Iterations	Blocked	Running time [ms]			Failed
ϵ		[%]		paths	A*	UBS	Total	[%]
	IPB-E	0.43	137.90	676.2	307.6	22.7	335.9	2.4
0.01	IPB-H	0.56	22.38	24.9	52.8	21.0	74.0	0.6
	IPF	0.61	1.73	-	-	-	2.3	0.0
	IPB-E	0.34	68.10	351.7	132.5	14.8	150.3	0.9
0.05	IPB-H	0.39	32.78	39.8	19.6	38.7	58.6	0.5
	IPF	0.41	1.54	-	-	-	2.3	0.0
	IPB-E	0.27	47.35	256.4	103.3	12.7	118.3	0.8
0.10	IPB-H	0.33	27.10	27.1	3.5	28.9	32.7	0.3
	IPF	0.34	1.45	-	-	-	2.7	0.0
	IPB-E	0.23	24.92	141.7	51.1	7.5	59.7	0.4
0.20	IPB-H	0.26	19.33	19.0	2.6	19.6	22.4	0.2
	IPF	0.28	1.36	-	-	-	2.1	0.0
	IPB-E	0.16	13.64	80.0	41.1	3.8	45.6	0.1
0.50	IPB-H	0.17	19.54	18.9	2.5	19.4	22.1	0.2
	IPF	0.19	1.26	-	-	-	2.0	0.0
	IPB-E	0.11	10.51	55.5	28.1	4.4	33.4	0.2
1.00	IPB-H	0.12	15.13	14.3	2.4	9.6	12.2	0.1
	IPF	0.14	1.19	-	-	-	2.5	0.0

note that due to the presence of heavy outliers, performing too few queries can distort the numbers drastically. For example, when we ran the same experiment with only 100 queries, the average running times of IPB-H were an order of magnitude faster.

We observe similar trends as the authors of [8]. The smaller the choice of ϵ, the harder the problem becomes. Consequently, the length increase, the number of iterations, the number of blocked paths and the running time increase. However, for our implementation of IPB-H, we measure slightly bigger path increases and slightly more iterations. Our implementation of IPB-H achieves running times two orders of magnitude faster than the CRP-based IPB-H implementation in [8]. One reason for this is our UBS algorithm which only needs a couple of milliseconds for all values of ϵ. In [8], the UBS checking phase takes between 1.3 and 1.9 *seconds*. The CH-Potentials-based shortest path phase is also very efficient across the entire range of ϵ values. Even with many blocked paths, the path lengths increase only little and the CH-Potentials heuristic remains tight and yields good speed-ups. Our exact IPB-E implementation is still an order of magnitude faster than the IPB-H implementation in [8].

6 Conclusion

In this paper, we studied the shortest smooth path problem and proved its NP-completeness. We introduced a new algorithm for practically efficient UBS computation. This algorithm can compute the exact UBS of typically occurring paths with very few shortest path computations. It outperforms state-of-the-art exact UBS algorithms by around two orders of magnitude and makes computing exact UBS values feasible in practice. Also, it can be used for other path quality measures such as local optimality.

We adapted the existing IPB-H algorithm and realized it with our new UBS algorithm and A* with CH-Potentials. This realization of IPB-H outperforms the original implementation by two orders of magnitude. Also, we present necessary modifications to make the algorithm exact. IPB-E is still about an order of magnitude faster than the CRP-based heuristic implementation. As IPB-H and IPB-E are not always able to find solutions in reasonable time, we introduce another heuristic, IPF. It can consistently find smooth paths even for random queries on massive continental sized instances in a few tenths of milliseconds.

For future work we would like to apply our algorithms not only to live traffic but also to predicted traffic, i.e. find smooth paths in a time-dependent setting. Further, it would be interesting to study what causes IPB-H to be so much faster than IPB-E while retaining most of the quality. Maybe this could be traced to specific structures in road networks which then could be exploited to speed up IPB-E.

References

1 Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative Routes in Road Networks. *ACM Journal of Experimental Algorithmics*, 18(1):1–17, 2013.

2 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 Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.

3 Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1.4):1–43, April 2013.

4 Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Dynamic Time-Dependent Route Planning in Road Networks with User Preferences. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*, volume 9685 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2016.

5 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning in Road Networks. *Transportation Science*, 51(2):566–591, 2017. doi:10.1287/trsc.2014.0579.

6 Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Faster Batched Shortest Paths in Road Networks. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 52–63, 2011.

7 Daniel Delling and Giacomo Nannicini. Core Routing on Dynamic Time-Dependent Road Networks. *Informs Journal on Computing*, 24(2):187–201, 2012.

8 Daniel Delling, Dennis Schieferdecker, and Christian Sommer. Traffic-Aware Routing in Road Networks. In *Proceedings of the 34rd International Conference on Data Engineering*. IEEE Computer Society, 2018. doi:10.1109/ICDE.2018.00172.

9 Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

10 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 21(1):1.5:1–1.5:49, April 2016. doi:10.1145/2886843.

11 Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

12 Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

13 Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness*. W. H. Freeman and Company, 1979.

14 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, August 2012.

15 Peter E. Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

16 Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

17 Ben Strasser, Dorothea Wagner, and Tim Zeitz. Space-efficient, Fast and Exact Routing in Time-Dependent Road Networks. *Algorithms*, 14(3), January 2021. URL: `https://www.mdpi.com/1999-4893/14/3/90`.

18 Ben Strasser and Tim Zeitz. A Fast and Tight Heuristic for A* in Road Networks. In David Coudert and Emanuele Natale, editors, *19th International Symposium on Experimental Algorithms (SEA 2021)*, volume 190 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.SEA.2021.6`.

A Detailed Performance Profiles by Instance

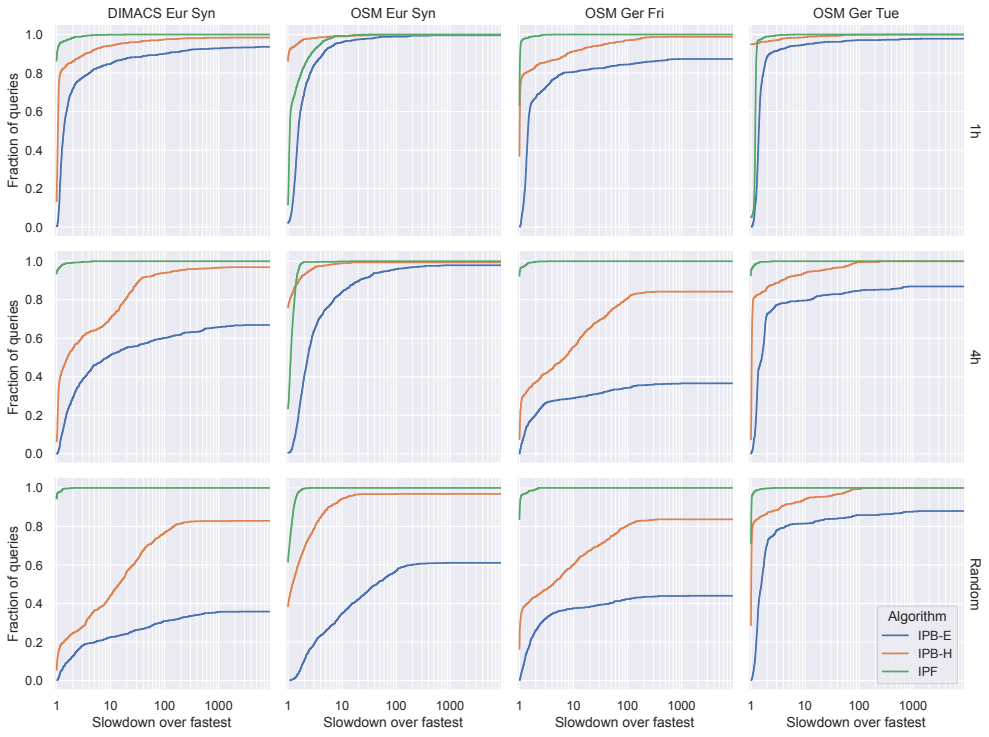


Figure 6 Relative performance profile for the running time of our algorithms on all queries from Table 2 split by graph and query set.

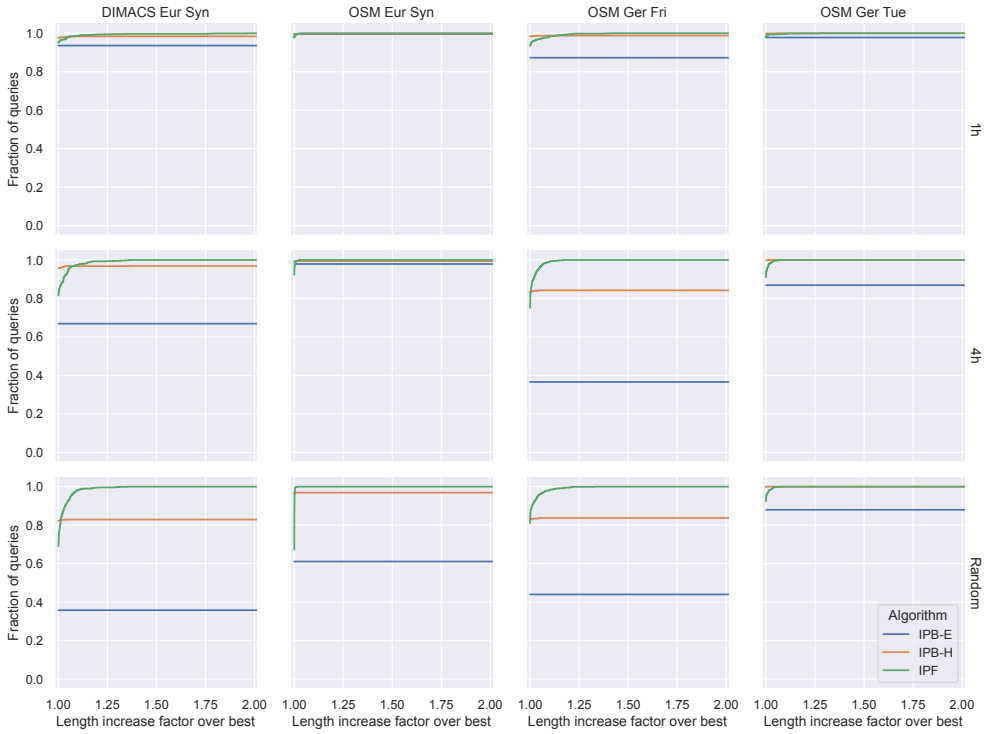


Figure 7 Relative performance profile for solution quality of our algorithms on all queries from Table 2 split by graph and query set.

Fast Succinct Retrieval and Approximate Membership Using Ribbon

Peter C. Dillinger
Meta, Seattle, WA, USA

Lorenz Hübschle-Schneider
Karlsruhe Institute of Technology, Germany

Peter Sanders
Karlsruhe Institute of Technology, Germany

Stefan Walzer
Universität Köln, Germany

Abstract

A *retrieval* data structure for a static function $f : S \to \{0,1\}^r$ supports queries that return $f(x)$ for any $x \in S$. Retrieval data structures can be used to implement a static approximate membership query data structure (AMQ), i.e., a Bloom filter alternative, with false positive rate 2^{-r}. The information-theoretic lower bound for both tasks is $r|S|$ bits. While *succinct* theoretical constructions using $(1 + o(1))r|S|$ bits were known, these could not achieve very small overheads in practice because they have an unfavorable space–time tradeoff hidden in the asymptotic costs or because small overheads would only be reached for physically impossible input sizes. With *bumped ribbon retrieval (BuRR)*, we present the first practical succinct retrieval data structure. In an extensive experimental evaluation BuRR achieves space overheads well below 1 % while being faster than most previously used retrieval data structures (typically with space overheads at least an order of magnitude larger) and faster than classical Bloom filters (with space overhead $\geq 44\,\%$). This efficiency, including favorable constants, stems from a combination of simplicity, word parallelism, and high locality.

We additionally describe *homogeneous ribbon filter AMQs*, which are even simpler and faster at the price of slightly larger space overhead.

2012 ACM Subject Classification Theory of computation → Data compression; Information systems → Point lookups

Keywords and phrases AMQ, Bloom filter, dictionary, linear algebra, randomized algorithm, retrieval data structure, static function data structure, succinct data structure, perfect hashing

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.4

Related Version There is a preprint [21] and an earlier technical report [22].
Full Version: `https://arxiv.org/abs/2109.01892`

Supplementary Material The code and scripts used in our experiments are available under a permissive license at github:
Software (Source Code): `https://github.com/lorenzhs/BuRR`
archived at `swh:1:dir:fa31381ae0e372bb33819932c0bcc1c51dcc0dfa`
Software (Source Code): `https://github.com/lorenzhs/fastfilter_cpp`
archived at `swh:1:dir:66c22ae4f3cc568d78c7b0d719988984a6688801`

Funding *Stefan Walzer*: DFG grant WA 5025/1-1.

1 Introduction

A *retrieval data structure* (sometimes called "static function") represents a function $f : S \to \{0,1\}^r$ for a set $S \subseteq \mathcal{U}$ of n keys from a universe $\mathcal{U}$ and $r \in \mathbb{N}$. A query for $x \in S$ must return $f(x)$, but a query for $x \in \mathcal{U} \setminus S$ may return any value from $\{0,1\}^r$.

The information-theoretic lower bound for the space needed by such a data structure is nr bits in the general case.[1] This significantly undercuts the $\Omega((\log|\mathcal{U}|+r)n)$ bits[2] needed by a dictionary, which must return "None" for $x \in \mathcal{U} \setminus S$. The intuition is that dictionaries have to store $f \subseteq \mathcal{U} \times \{0,1\}^r$ as a set of key-value pairs while retrieval data structures, surprisingly, need not store the keys. We say a retrieval data structure using s bits has *(space) overhead* $\frac{s}{nr} - 1$.

The starting point for our contribution is a *compact* retrieval data structure from [20], i.e. one with overhead $\mathcal{O}(1)$. After minor improvements, we first obtain *standard ribbon retrieval*. All theoretical analysis assumes computation on a word RAM with word size $\Omega(\log n)$ and that hash functions behave like random functions.[3] The *ribbon width* w is a parameter that also plays a role in following variants.

▶ **Theorem 1** (similar to [20]). *For any $\varepsilon > 0$, an r-bit standard ribbon retrieval data structure with ribbon width $w = \frac{\log n}{\varepsilon}$ has construction time $\mathcal{O}(n/\varepsilon^2)$, query time $\mathcal{O}(r/\varepsilon)$ and overhead $\mathcal{O}(\varepsilon)$.*

We then combine standard ribbon retrieval with the idea of *bumping*, i.e., a convenient subset $S' \subseteq S$ of keys is handled in the first *layer* of the data structure and the small rest is *bumped* to recursively constructed subsequent layers. The resulting *bumped ribbon retrieval* (BuRR) data structure has much smaller overhead for any given ribbon width w.

▶ **Theorem 2.** *An r-bit BuRR data structure with ribbon width $w = \mathcal{O}(\log n)$ and $r = \mathcal{O}(w)$ has expected construction time $\mathcal{O}(nw)$, space overhead $\mathcal{O}(\frac{\log w}{rw^2})$, and query time $\mathcal{O}(1 + \frac{rw}{\log n})$.*

Table 1 Performance of various r-bit retrieval data structures with $r = \mathcal{O}(\log n)$. **Bold** overhead indicates that the data structure is (or can be configured to be) succinct. The parameters $k \in \mathbb{N}$ and $\varepsilon > 0$ are constants with respect to n. The parameter $C \in \mathbb{N}$ is typically n^α for constant $\alpha \in (0,1)$.

		Year	$t_{\text{construct}}$	t_{query}	multiplicative overhead	shard size	Solver
Standard Ribbon	[34]	2001	$\mathcal{O}(n \log k)$	$\mathcal{O}(\log k)^\dagger$	$\frac{1}{k}$	–	peeling
	[40]	2009	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\boldsymbol{\mathcal{O}(\frac{\log\log n}{(\log n)^{1/2}})}$	$\sqrt{\log n}$	lookup table
	[9]	2013	$\mathcal{O}(n)$	$\mathcal{O}(1)$	0.2218	–	peeling
	[4]	2013	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\boldsymbol{\mathcal{O}(\frac{\log^2 \log n}{r \log n})}$	$\mathcal{O}(\frac{\log^2 \log n}{r \log n})$	–
	[39]	2014	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\Omega(1/r)$	$\mathcal{O}(1)$	sorting/sharding
	[27]	2016	$\mathcal{O}(nC^2)$	$\mathcal{O}(1)$	$0.024 + \mathcal{O}(\frac{\log n}{C})$	C	structured Gauss
⟶	[20]	2019	$\mathcal{O}(n/\varepsilon^2)$	$\mathcal{O}(r/\varepsilon)$	ε	–	Gauss
	[20]	2019	$\mathcal{O}(n/\varepsilon)$	$\mathcal{O}(r)$	$\varepsilon + \mathcal{O}(\frac{\log n}{n^\varepsilon})$	n^ε	Gauss
	[18]	2019	$\mathcal{O}(nC^2)$	$\mathcal{O}(r)$	$\boldsymbol{\mathcal{O}(\frac{\log n}{C})}$	C	structured Gauss
	[44]	2021	$\mathcal{O}(nk)$	$\mathcal{O}(k)$	$(1 + o_k(1))\mathrm{e}^{-k}$	–	peeling
	BuRR		$\mathcal{O}(nw)$	$\mathcal{O}(1 + \frac{rw}{\log n})$	$\boldsymbol{\mathcal{O}(\frac{\log w}{rw^2})}$	–	on-the-fly Gauss
	↪ with $w = \Theta(\log n)$:		$\mathcal{O}(n \log n)$	$\mathcal{O}(r)$	$\boldsymbol{\mathcal{O}(\frac{\log\log n}{r \log^2 n})}$	–	on-the-fly Gauss

† Expected query time. Worst case query time is $\mathcal{O}(D)$.

In particular, BuRR can be configured to be *succinct*, i.e., can be configured to have an overhead of $o(1)$ while retaining constant access time for small r. Construction time is slightly superlinear. Note that succinct retrieval data structures were known before, even

[1] If f has low entropy then *compressed static functions* [31, 4, 28] can do better and even machine learning techniques might help; see e.g. [42].

[2] This lower bound holds when $|\mathcal{U}| = \Omega(n^{1+\delta})$ for $\delta > 0$. The general bound is $\log\binom{|\mathcal{U}|}{n} + nr$ bits.

[3] This is a standard assumption in many papers and can also be justified by standard constructions [17].

with asymptotically optimal construction and query times of $\mathcal{O}(n)$ and $\mathcal{O}(1)$, respectively [40, 4]. Seeing the advantages of BuRR requires a closer look. Details are given in Section 5, but the gist can be seen from Table 1: Among the previous succinct retrieval data structures (overheads set in bold font), only [18] can achieve small overhead in a *tunable* way, i.e., independently of n using an appropriate tuning parameter $C = \omega(\log n)$. However, this approach suffers from comparatively high constructions times. [40] and [4] are not tunable and only *barely* succinct with significant overhead in practice. A quick calculation to illustrate: Neglecting the factors hidden by $\mathcal{O}$-notation, the overheads are $\frac{\log\log n}{\sqrt{\log n}}$ and $\frac{\log^2 \log n}{r \log n}$, which is at least 75% and 7% for $r = 8$ and any $n \leq 2^{64}$. A similar estimation for BuRR with $w = \Theta(\log n)$ suggests an overhead of $\frac{\log\log n}{r \log^2 n} \approx 0.1\%$ already for $r = 8$ and $n = 2^{24}$. Moreover, by tuning the ribbon width w, a wide range of trade-offs between small overhead and fast running times can be achieved.

Overall, we believe that asymptotic analyses struggle to tell the full story due to the extremely slow decay of some "$o(1)$" terms. We therefore accompany the theoretical account with experiments comparing BuRR to other efficient (compact or succinct) retrieval data structures. We do this in the use case of data structures for approximate membership and also invite competitors not based on retrieval into the ring such as (blocked) Bloom filters and Cuckoo filters.

Data structures for approximate membership. Retrieval data structures are an important basic tool for building compressed data structures. Perhaps the most widely used application is associating an r-bit fingerprint with each key from a set $S \subseteq \mathcal{U}$, which allows implementing an *approximate membership query data structure* (*AMQ*, aka *Bloom filter replacement* or simply *filter*) that supports membership queries for S with *false positive rate* $\varphi = 2^{-r}$. A membership query for a key $x \in \mathcal{U}$ will simply compare the fingerprint of x with the result returned by the retrieval data structure for x. The values will be the same if $x \in S$. Otherwise, they are the same only with probability 2^{-r}.

In addition to the AMQs following from standard ribbon retrieval and BuRR, we also present homogeneous ribbon filters, which are not directly based on retrieval.

▶ **Theorem 3.** *Let $r \in \mathbb{N}$ and $\varepsilon \in (0, \frac{1}{2}]$. There is $w \in \mathbb{N}$ with $\frac{w}{\max(r, \log w)} = \mathcal{O}(1/\varepsilon)$ such that the homogeneous ribbon filter with ribbon width w has false positive rate $\varphi \leq (1+\varepsilon^2)2^{-r}$ and space overhead $\mathcal{O}(\varepsilon)$. On a word RAM with word size $\geq w$ expected construction time is $\mathcal{O}(n/\varepsilon)$ and query time is $\mathcal{O}(r)$.*

Experiments. Figure 1 shows some of the results explained in detail later in the paper. In the depicted parallel setting, ribbon-based AMQs (blue) are the fastest static AMQs when an overhead less than $\approx 44\%$ is desired (where "fastest" considers a somewhat arbitrary weighting of construction and query times). The advantage is less pronounced in the sequential setting.

Why care about space? Especially in AMQ applications, retrieval data structures occupy a considerable fraction of RAM in large server farms continuously drawing many megawatts of power. Even small reductions (say 10 %) in their space consumption thus translate into considerable cost savings. Whether or not these space savings should be pursued at the price of increased access costs depends on the number of queries per second. The lower the access frequency, the more worthwhile it is to occasionally spend increased access costs for a permanently lowered memory budget. Since the false-positive rate also has an associated cost (e.g. additional accesses to disk or flash) it is also subject to tuning. The entire set of

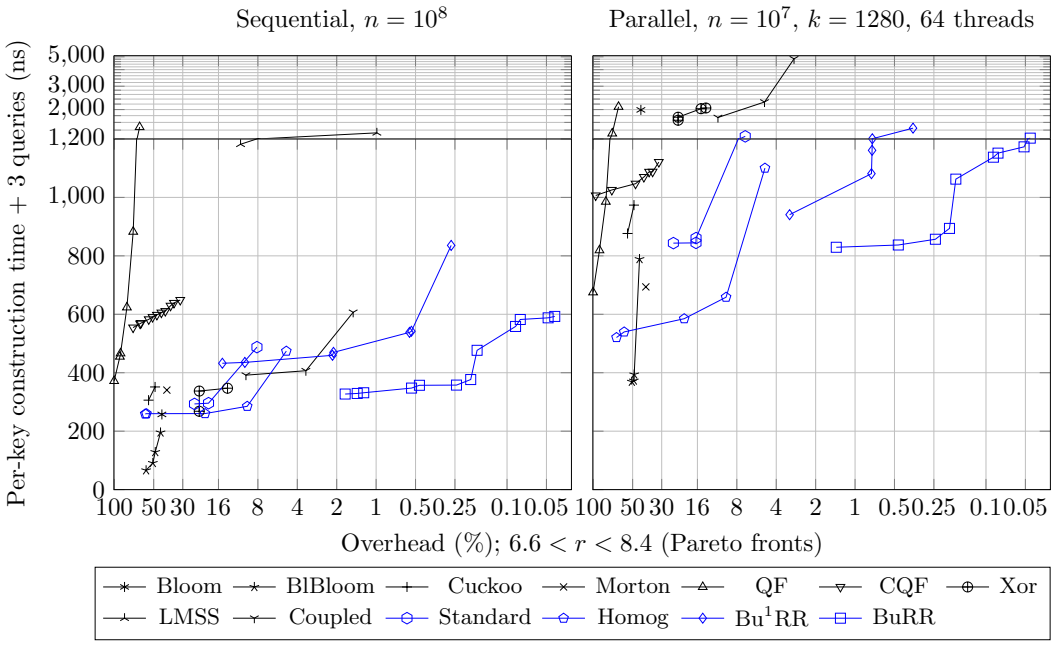


Figure 1 Performance–overhead trade-off for measured false-positive rate in 0.003–0.01 (i.e., $r \approx 8$), for different AMQs and inputs. Ribbon-based data structures are in blue. For each category of approaches, only variants are shown that are not Pareto-dominated by variants in the same category. Sequential benchmarks use a single filter of size n while the parallel benchmark uses 1280 filters of size n and utilizes 64 cores. Logarithmic vertical axis above 1200 ns.

Pareto-optimal variants with respect to tradeoffs between space, access time, and FP rate is relevant for applications. For instance, sophisticated implementations of LSM-trees use multiple variants of AMQs at once based on known access frequencies [14]. Similar ideas have been used in compressed data bases [38].

Outline. The paper is organized as follows (section numbers in parentheses). After important preliminaries (2), we explain our data structures and algorithms in broad strokes (3) and summarize our experimental findings (4). We then summarize related work (5). In the full paper [21] we give a detailed theoretical analysis, extensively describe the design space of BuRR, and discuss additional experiments.

2 Linear Algebra Based Retrieval Data Structures and SGAUSS

A simple, elegant and highly successful approach for compact and succinct retrieval uses linear algebra over the finite field $\mathbb{Z}_2 = \{0, 1\}$ [16, 27, 1, 40, 12, 9, 18, 20]. Refer to Section 5 for a discussion of alternative and complementary techniques.

The train of thought is this: A natural idea would be to have a hash function point to a location where the key's information is stored while the key itself need not be stored. This fails because of hash collisions. We therefore allow the information for each key to be dispersed over several locations. Formally we store a table $Z \in \{0, 1\}^{m \times r}$ with $m \geq n$ entries of r bits each and to define $f(x)$ as the bit-wise xor of a set of table entries whose positions

$h(x) \subseteq [m]$ are determined by a hash function h.[4] This can be viewed as the matrix product $\vec{h}(x)Z$ where $\vec{h}(x) \in \{0,1\}^m$ is the characteristic (row)-vector of $h(x)$. For given h, the main task in building the data structure is to find the right table entries such that $\vec{h}(x)Z = f(x)$ holds for every key x. This is equivalent to solving a system of linear equations $AZ = \mathbf{b}$ where $A = (\vec{h}(x))_{x \in S} \in \{0,1\}^{n \times m}$ and $\mathbf{b} = (f(x))_{x \in S} \in \{0,1\}^{n \times r}$. Note that rows in the constraint matrix A correspond to keys in the input set S. In the following, we will thus switch between the terms "row" and "key" depending on which one is more natural in the given context.

An encouraging observation is that even for $m = n$, the system $AZ = \mathbf{b}$ is solvable with constant probability if the rows of A are chosen uniformly at random [13, 40]. With linear query time and cubic construction time, we can thus achieve optimal space consumption. For a practically useful approach, however, we want the 1-entries in $\vec{h}(x)$ to be sparse and highly localized to allow cache-efficient queries in (near) constant time and we want a (near) linear time algorithm for solving $AZ = \mathbf{b}$. This is possible if $m > n$.

A particularly promising approach in this regard is SGAUSS from [20] that chooses the 1-entries within a narrow range. Specifically, it chooses w random bits $c(x) \in \{0,1\}^w$ and a random *starting position* $s(x) \in [m - w - 1]$, i.e., $\vec{h}(x) = 0^{s(x)-1}c(x)0^{m-s(x)-w+1}$. For $m = (1+\varepsilon)n$ some value $w = \mathcal{O}(\log(n)/\varepsilon)$ suffices to make the system $AZ = \mathbf{b}$ solvable with high probability. We call w the *ribbon width* because after sorting the rows of A by $s(x)$ we obtain a matrix which is not technically a band matrix, but which likely has all 1-entries within a narrow *ribbon* close to the diagonal. The solution Z can then be found in time $\mathcal{O}(n/\varepsilon^2)$ using Gaussian elimination [20] and bit-parallel row operations; see also Figure 2 (a).

3 Ribbon Retrieval and Ribbon Filters

We advance the linear algebra approach to the point where space overhead is almost eliminated while keeping or improving the running times of previous constructions.

Ribbon solving. Our first contribution is a simple algorithm we could not resist to also call *ribbon* as in ***R**apid **I**ncremental **B**oolean **B**anding **ON** the fly*. It maintains a system of linear equations in row echelon form as shown in Figure 2 **(b)**. It does so *on-the-fly*, i.e. while equations arrive one by one in arbitrary order. For each index i of a column there may be at most one equation that has its leftmost one in column i. When an equation with row vector a arrives and its slot is already taken by a row a', then ribbon performs the row operation $a \leftarrow a \oplus a'$, which eliminates the 1 in position i, and continues with the modified row. An invariant is that rows have all their nonzeroes in a range of size w, which allows to process rows with a small number of bit-parallel word operations. This insertion process is *incremental* in that insertions do not modify existing rows. This improves performance and allows to cheaply roll back the most recent insertions which will be exploited below. It is a non-trivial insight that the order in which equations are added does not significantly affect the expected number of row operations. This is made precise and proved in the full paper.

When all rows are processed we perform back-substitution to compute the solution matrix Z. At least for small r, *interleaved representation* of Z works well, where blocks of size $w \times r$ of Z are stored column-wise. A query for x can then retrieve one bit of $f(x)$ at a time by

[4] In this paper, $[k]$ can stand for $\{0, \ldots, k-1\}$ or $\{1, \ldots, k\}$ (depending on the context), and $a..b$ stands for $\{a, \ldots, b\}$.

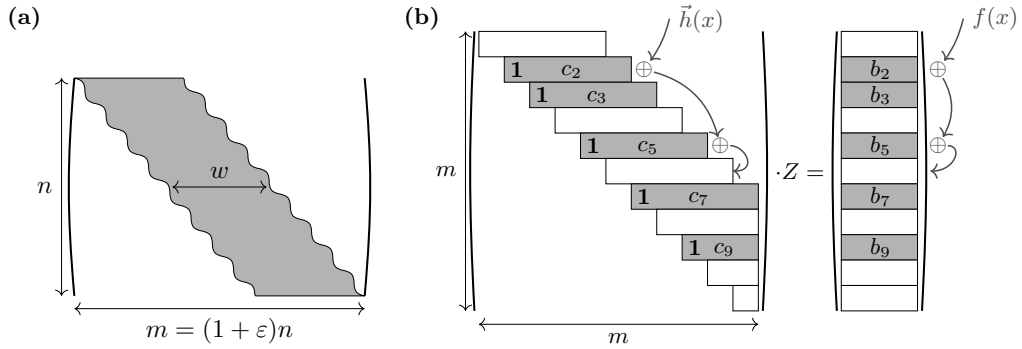


Figure 2 **(a)** Typical shape of the random matrix A with rows $(\vec{h}(x))_{x\in S}$ sorted by starting positions. The shaded "ribbon" region contains random bits. Gaussian elimination never causes any fill-in outside of the ribbon.
(b) Shape of the linear system M in row echelon form maintained using Boolean banding on the fly. In gray we visualize the insertion of a key x where (i) $\vec{h}(x)$ has its left-most 1 in position $s(x) = 2$, (ii) after xoring the second row of M to $\vec{h}(x)$, the left-most 1 is in position 5 and (iii) xoring the fifth row as well, the left-most 1 is in position 6. The resulting row fills the previously empty sixth row of M and $f(x) \oplus b_2 \oplus b_5$ is added as right hand side.

applying a population count instruction to pieces of rows retrieved from at most two of these blocks. This is particularly advantageous for negative queries to AMQs (i.e. queries of elements not in the set), where only two of r bits need to be retrieved on average. More details are given in the full paper.

3.1 Standard Ribbon

When employing no further tricks, we obtain *standard ribbon retrieval*, which is essentially the same data structure as in [20] except with a different solver that is faster by noticeable constant factors. A problem is that w becomes prohibitively large when n is large and ε is small. For example, experiments show that for $\varepsilon \leq 3.5\%$ and construction success rate $\geq 50\%$, standard word size $w = 64$ only scales to around $n \leq 10^4$ and more expensive $w = 128$ only scales to around $n \leq 10^6$. To some degree this can be mitigated by sharding techniques [43], but in this paper we pursue a more ambitious route.

3.2 Bumped Ribbon Retrieval

Our main contribution is *bumped ribbon retrieval (BuRR)*, which reduces the required ribbon width to a constant that only depends on the targeted space efficiency. BuRR is based on two ideas.

Bumping. The ribbon solving approach manages to insert most rows (representing most keys of S) even when w is small. Thus, by eliminating those rows/keys that cause a linear dependency, we obtain a compact retrieval data structure for a large subset of S. The remaining keys are *bumped*, meaning they are handled by a fallback data structure which, by recursion, can be a BuRR data structure again. We show that only $\mathcal{O}(\frac{n \log w}{w})$ keys need to be bumped in expectation. Thus, after a constant number of layers (we use 4), a less ambitious retrieval data structure can be used to handle the few remaining keys without bumping.

The main challenge is that we need additional metadata to encode which keys are bumped. The basic *bumped retrieval* approach is adopted from the updateable retrieval data structure *filtered retrieval (FiRe)* [39]. To shrink the input size by a moderate constant factor, FiRe needs a constant number of bits per key (around 4). This leads to very high space overhead for small r. A crucial observation for BuRR is that bumping can be done with granularity much coarser than per-key. We will bump keys based on their starting position and say *position i is bumped* to indicate that all keys with $s(x) = i$ are bumped. Bumping by position is sufficient because linear dependencies in A are largely unrelated to the actual bit patterns $c(x)$ but mostly caused by fluctuations in the number of keys mapped to different parts of the matrix A. By selectively bumping ranges of positions in overloaded parts of the system, we can obtain a solvable system. Furthermore, our analysis shows that we can drastically limit the spectrum of possible bumping ranges; see below.

Overloading. Besides metadata, space overhead results from the $m - n + n_b$ excess slots of the table where n_b is the number of bumped keys. Trying out possible values of $\varepsilon = \frac{m-n}{n} > 0$ one sees that the overhead due to excess slots is always $\Omega(1/w)$ and will thus dominate the overhead due to metadata. However, we show that by choosing $\varepsilon < 0$ (of order $-\varepsilon = \mathcal{O}(\frac{\log w}{w})$), i.e., by *overloading* the table, we can almost completely eliminate excess table slots so that the minuscule amount of metadata becomes the dominant remaining overhead. There are many ways to decide and encode which keys are bumped. Here, we outline a simple variant that achieves very good performance in practice and is a generalization of the theoretically analyzed approach. We expand on the much larger design space of BuRR in the full paper.

Deciding what to bump. We subdivide the possible starting positions into *buckets* of width $b = \mathcal{O}(w^2/\log w)$ and allow to bump a single initial range of each bucket. The keys (or more precisely pairs of hashes and the value to be retrieved) are sorted according to the bucket addressed by the starting position $s(x)$. We use a fast in-place integer sorter for this purpose [2]. Then buckets are processed one after the other from *left to right*. Within a bucket, however, keys are inserted into the row echelon form from *right to left*. The reason for this is that insertions of the previous bucket may have "spilled over" causing additional load on the left of the bucket – an issue we wish to confront as late as possible. See also Figure 3.

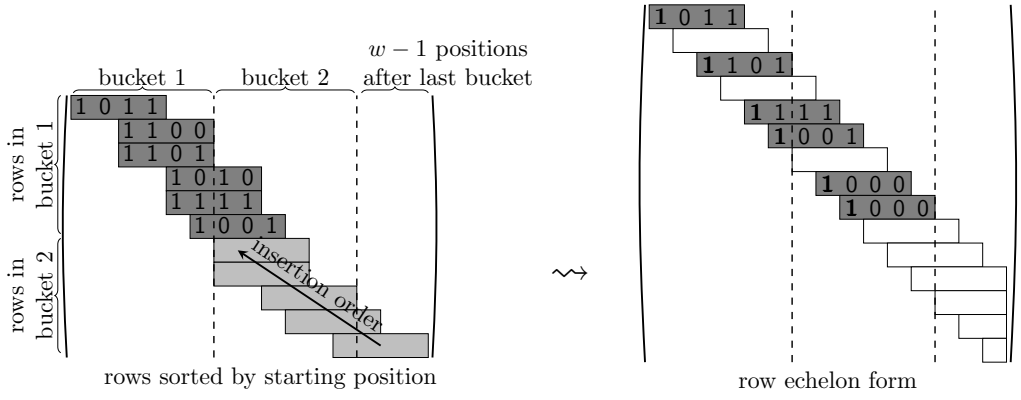


Figure 3 Illustration of BuRR construction with $n = 11$ keys, $m = 2b + w - 1 = 15$ table positions, ribbon width $w = 4$ and bucket size $b = 6$. Keys of the first bucket were successfully inserted (from right to left) into row echelon form with two insertions "overflowing" into the second bucket. Insertions of the second bucket's rows will be attempted next, in the indicated order.

If all keys of a bucket can be successfully inserted, no keys of the bucket are bumped. Otherwise, suppose the first failed insertion for a bucket $[i, i+b)$ concerns a key where $s(x) = i + k$ is the k-th position of the bucket. We *could* decide to bump all keys x' of the bucket with $s(x') \leq i + k$, which would require storing the *threshold* k using $\mathcal{O}(\log w)$ bits and which would yield an overhead of $\mathcal{O}(\log^2(w)/w^2)$ due to metadata. *Instead*, to reduce this overhead to $\mathcal{O}(\log(w)/w^2)$, we only allow a constant number of threshold values. This means that we find the smallest threshold value t with $t \geq k$ representable by metadata and bump all keys x' with $s(x') \leq i + t$. This requires rolling back the insertions of keys x' with $s(x') \in [k, t]$ by clearing the most recently populated rows from the row echelon form. One good compromise between space and speed stores 2 bits per bucket encoding the threshold values $\{0, \ell, u, b\}$, for suitable ℓ and u. The special case $\ell = u = \frac{3}{8}w$ is used in our analysis. Another slightly more compact variant "1^+-bit" stores one bit encoding threshold values from the set $\{0, t\}$, for a suitable t, and additionally stores a hash table of exceptions for thresholds $> t$.

Running times. With these ingredients we obtain Theorem 2 stated on page 2. It implies constant query time[5] if $rw = \mathcal{O}(\log n)$ and linear construction time if $w \in \mathcal{O}(1)$. For wider ribbons, construction time is slightly superlinear. However, in practice this does not necessarily mean that BuRR is slower than other approaches with asymptotically better bounds as the factor w involves operations with very high locality. An analysis in the external memory model reveals that BuRR construction is possible with a single scan of the input and integer sorting of n objects of size $\mathcal{O}(\log n)$ bits; see the full paper for details.

3.3 Homogeneous Ribbon Filter

For the application of ribbon to AMQs, we can also compute a *uniformly random* solution of the *homogeneous* equation system $AZ = 0$, i.e., we compute a retrieval data structure that will retrieve 0^r for all keys of S but is unlikely to produce 0^r for other inputs. Since $AZ = 0$ is always solvable, there is no need for bumping. The crux is that the false positive rate is no longer 2^{-r} but higher. In the full paper we show that with table size $m = (1 + \varepsilon)n$ and $\varepsilon = \Omega(\frac{\max(r, \log w)}{w})$ the difference is negligible, thereby showing Theorem 3. Homogeneous ribbon AMQs are simpler and faster than BuRR but have higher space overhead. Our experiments indicate that together, BuRR and homogeneous ribbon AMQs cover a large part of the best tradeoffs for static AMQs.

3.4 Analysis outline

To get an intuition for the relevant linear systems, it is useful to consider two simplifications. First, assume that $\vec{h}(x)$ contains a block of w uniformly random *real* numbers from $[0, 1]$ rather than w random bits. Secondly, assume that we sort the rows by starting position and use Gaussian elimination rather than ribbon to produce a row echelon form. In Figure 4 **(a)** we illustrate for such a matrix with $\times$-marks where the pivots would be placed and in yellow the entries that are eliminated (with one row operation each); both with probability 1, i.e. barring coincidences where a row operation eliminates more than one entry. The $\times$-marks trace a diagonal through the matrix except that the green column and the red row are skipped

[5] It should be noted that the proof invokes a lookup table in one case to speed up the computation of a matrix vector product. In Section 5, we argue that lookup tables should be avoided in practice. Technically, our *implementation* using *interleaved representation* has a query time of $\mathcal{O}(r)$.

because the end of the (gray) area of nonzeroes is reached. "Column failures" correspond to free variables and therefore unused space. "Row failures" correspond to linearly dependent equations and therefore failed insertions. This view remains largely intact when handling *Boolean* equations in *arbitrary* order except that the *ribbon diagonal*, which we introduce as an analogue to the trace of pivot positions, has a more abstract meaning and probabilistically suffers from row and column failures depending on its *distance* to the ribbon border.

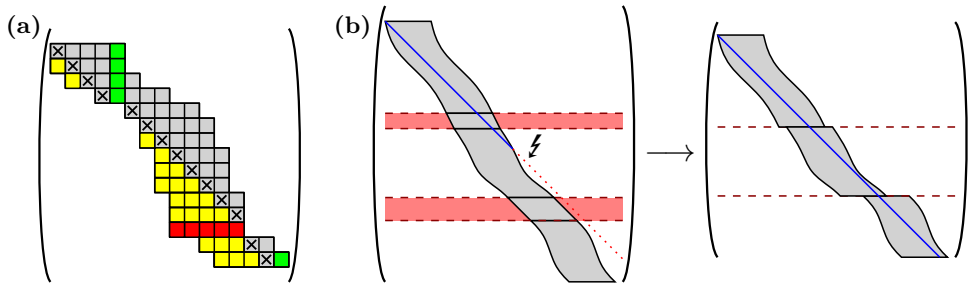


Figure 4 **(a)** The simplified ribbon diagonal (made up of ×-marks) passing through A. **(b)** The idea of BuRR: When starting with an "overloaded" linear system and removing sets of rows strategically, we can often ensure that the ribbon diagonal does not collide with the ribbon border (except possibly in the beginning and the end).

The idea of standard ribbon is to give the gray ribbon area an expected slope of less than 1 such that row failures are unlikely. BuRR, as illustrated in Figure 4 **(b)** largely avoids both failure types by using a slope bigger than 1 but removing ranges of rows in strategic positions. Homogeneous ribbon filters, despite being the simplest approach, have the most subtle analysis as both failure types are allowed to occur. While row failures cannot cause outright construction failure, they are linked to a compromised false positive rate in a non-trivial way. Our proofs involve mostly simple techniques as would be used in the analysis of linear probing, which is unsurprising given that [20] has already established a connection to Robin Hood hashing. We also profit from queuing theory via results we import from [20].

3.5 Further results

We have several further results around variants of BuRR that we summarize here. See the full paper for detail.

Perhaps most interesting is **bump-once ribbon retrieval (Bu^{1}RR)**, which improves the worst-case query time by guaranteeing that each key can be retrieved from one out of two layers – its *primary layer* or the next one. The primary layer of the keys is now distributed over all layers (except for the last). When building a layer, the keys bumped from the previous layer are inserted into the row echelon form first. The layer sizes have to be chosen in such a way that no bumping is needed for these keys with high probability. Only then are the keys with the current layer as their primary layer inserted – now allowing bumping.

For building large retrieval data structures, **parallel construction** is important. Doing this directly is difficult for ribbon retrieval since there is no efficient way to parallelize back-substitutions. However, we can partition the equation system into parts that can be solved independently by bumping w consecutive positions. Note that this can be done transparently to the query algorithm by using the bumping mechanism that is present anyway.

For large r, we accelerate queries by working with **sparse bit patterns** that set only a small fraction of the w bits in the window used for BuRR. In some sense, we are covering here the middle ground between ribbon and spatial coupling [44]. Experiments indicate that

setting 8 out of 64 bits indeed speeds up queries for $r \in \{8, 16\}$ at the price of increased (but still small) overhead. Analysis and further exploration of this middle ground may be an interesting area for future work.

4 Summary of Experimental Findings

We performed extensive experiments to evaluate our ribbon-based data structures and competitors. We summarize our findings here with details provided in the full paper.

Implementation Details. We implemented BuRR in C++ using template parameters that allow us to navigate a large part of the design space mapped in the full paper. (Recall that r is the retrieval width, ϵ the overloading factor, w the ribbon width, and b the bucket width; t, ℓ, and u are bumping thresholds.) Input keys themselves are only hashed once to a 64-bit *master-hash-code (MHC)* that is subsequently used when further hash values are needed. For this, fast linear congruential mapping is used. The table is stored in an *interleaved fashion*, i.e., it is organized as rm/w words of w bits each where word i represents bit $i \bmod r$ of w subsequent table entries. This organization allows the extraction of one retrieved bit from two adjoining machine words using population-count instructions. Interleaved representation is advantageous for uses of BuRR as an AMQ data structure since a negative query only has to extract two bits in expectation. Moreover, the implementation directly works for any value of r. The default data structure has four layers, the last of which uses $w' := \min(w, 64)$ and $\varepsilon \geq 0$, where ε is increased in increments of 0.05 until no keys are bumped. For 1^+-bit, we choose $t := \lceil -2\varepsilon b + \sqrt{b/(1+\varepsilon)}/2 \rceil$ and $\varepsilon := -2/3 \cdot w/(4b + w)$. For 2-bit, parameter tuning showed that $\ell := \lceil (0.13 - \varepsilon/2)b \rceil$, $u := \lceil (0.3 - \varepsilon/2)b \rceil$, and $\varepsilon := -3/w$ work well for $w = 32$; for $w \geq 64$, we use $\ell = \lceil (0.09 - 3\varepsilon/4)b \rceil$, $u = \lceil (0.22 - 1.3\varepsilon)b \rceil$, and $\varepsilon := -4/w$.

In addition, there is a prototypical implementation of Bu^{1}RR from [22]. Both BuRR and Bu^{1}RR build on the same software for ribbon solving from [22]. For validation we extend the experimental setup used for Cuckoo and Xor filters [29], with our code and scripts available at github.com/lorenzhs/fastfilter_cpp and github.com/lorenzhs/BuRR.

Experimental Setup. All experiments were run on a machine with an AMD EPYC 7702 processor with 64 cores, a base clock speed of 2.0 GHz, and a maximum boost clock speed of 3.35GHz. The machine is equipped with 1 TiB of DDR4-3200 main memory and runs Ubuntu 20.04. We use `clang++` 11.0 with optimization flags `-O3 -march=native`. During sequential experiments, only a single core was used at any time to minimize interference.

We looked at different input sizes $n \in \{10^6, 10^7, 10^8\}$. Like most studies in this area, we first look at a **sequential** workload on a powerful processor with a considerable number of cores. However, this seems unrealistic since in most applications, one would not let most cores lay bare but use them. Unless these cores have a workload with very high locality this would have a considerable effect on the performance of the AMQs. We therefore also look at a scenario that might be the opposite extreme to a sequential unloaded setting. We run the benchmarks on all available hardware threads in **parallel**. Construction builds many AMQs of size n in parallel. Queries select AMQs randomly. This emulates a large AMQ that is parallelized using sharding and puts very high load on the memory system.

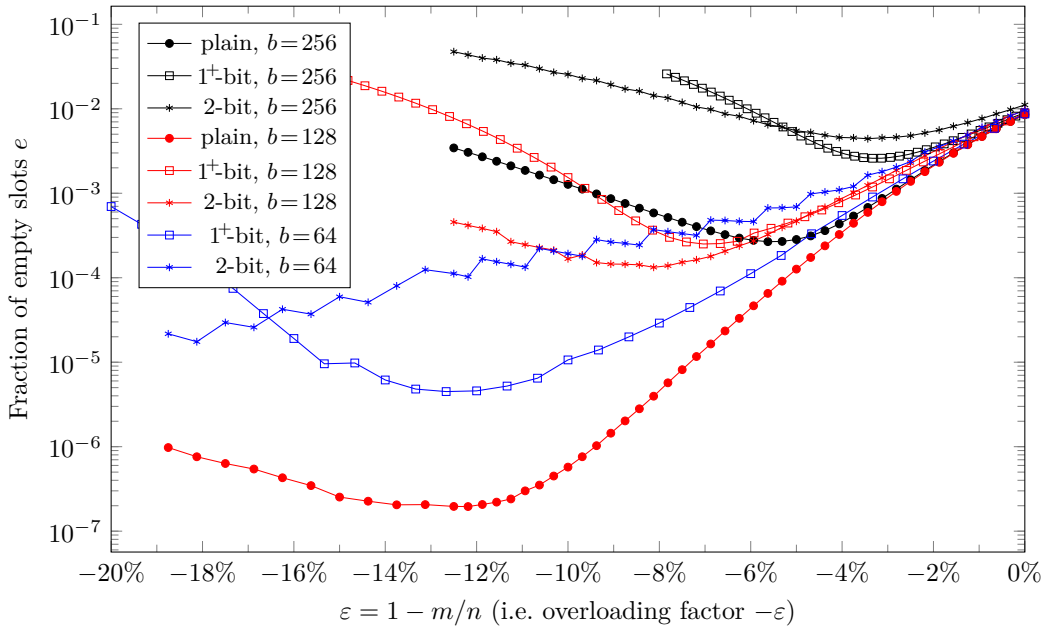


Figure 5 Fraction of empty slots for various configurations of bumped ribbon retrieval with $w = 64$, depending on the overloading factor $-\varepsilon$.

Experimental Results. Two preliminary remarks are in order: Firstly, since every retrieval data structure can be used as a filter but not vice versa, our experiments are for filters, which admits a larger number of competitors. Secondly, to reduce complexity (for now), our speed ranking considers the sum of construction time per key and three query times.[6]

Space Overhead of BuRR. Figure 5 plots the fraction e of empty slots of BuRR for $w = 64$ and several combinations of bucket size b and different threshold compression schemes. Similar plots are given in the full paper for $w = 32$, $w = 128$, and for $w = 64$ with sparse coefficients. Note that (for an infinite number of layers), the overhead is about $o = e + \mu/(rb(1-e))$ where r is the number of retrieved bits and μ is the number of metadata bits per bucket. Hence, at least when μ is constant, the overhead is a monotonic function in e and thus minimizing e also minimizes overhead.

We see that for small $|\varepsilon|$, e decreases exponentially. For sufficiently small b, e can get almost arbitrarily small. For fixed $b > w$, e eventually reaches a local minimum because with threshold-based compression, a large overload enforces large thresholds ($> w$) and thus empty regions of buckets. Which actual configuration to choose depends primarily on r. Roughly, for larger r, more and more metadata bits (i.e., small b, higher resolution of threshold values) can be invested to reduce e. For fixed b and threshold compression schemes, one can choose ε to minimize e. One can often choose a larger ε to get slightly better performance due to less bumping with little impact on o. Perhaps the most delicate tuning parameters are the thresholds to use for 2-bit and 1^+-bit compression. Indeed, in Figure 5 1^+-bit compression

[6] Queries measured in three settings: Positive keys, negative keys and a mixed data set (50 % chance of being positive). The latter is not an average of the first two due to branch mispredictions. In the appendix, we also measure the individual operations resulting in similar conclusions.

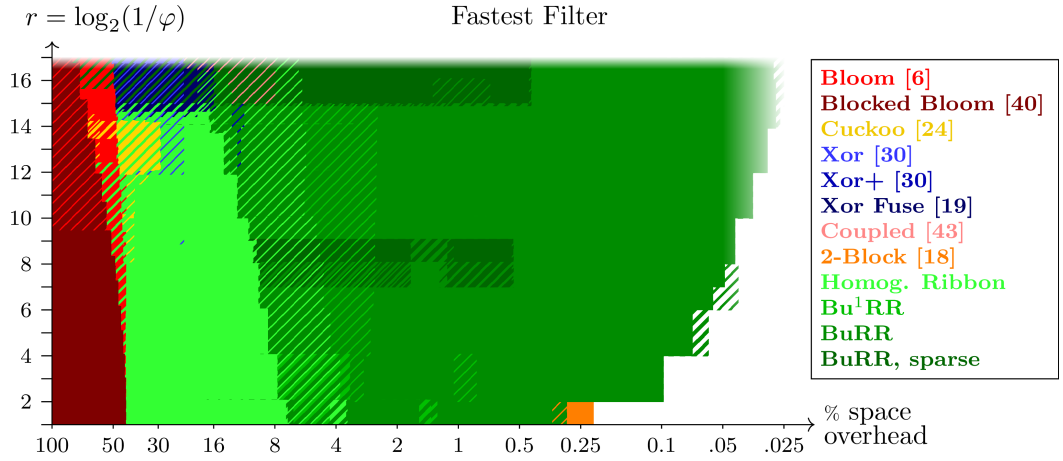


Figure 6 Fastest AMQ category for different choices of overhead and false-positive rate $\varphi = 2^{-r}$. Shaded regions indicates a dependency on the input type. Ranking metric: construction time per key plus time for three queries, of which one is positive, one negative, and one mixed (50 % chance of either).

has lower e than 2-bit compression for $b = 64$ but higher e for $b = 128$. We expect that 2-bit compression could always achieve smaller e than 1^+-bit compression, but we have not found choices for the threshold values that always ensure this.

Ribbon yields the fastest static AMQs for overhead $< 44\%$. Consider Figure 1 on page 4, where we show the tradeoff between space overhead and computation cost for a range of AMQs for false positive rate $\varphi \approx 2^{-8}$ (i.e., $r = 8$ for BuRR) and large inputs.[7] In the parallel workload on the right all cores access many AMQs randomly.

Only three AMQs have Pareto-optimal configurations for this case: BuRR for space overhead below 5 % (actually achieving between 1.4 % and 0.2 % for a narrow time range of 830–890 ns), homogeneous ribbon for space overhead below 44 % (actually achieving between 20 % and 10 % for a narrow time range 580–660 ns), and *blocked Bloom filters* [41] with time around 400 ns at the price of space overhead of around 50 %. All other tried AMQs are dominated by homogeneous ribbon and BuRR. Somewhat surprisingly, this even includes plain Bloom filters [6] which are slow because they incur several cache faults for each insertion and positive query. Since plain Bloom filters are extensively used in practice (often in cases where a static interface suffices), we conclude that homogeneous ribbon and BuRR are fast enough for a wide range of applications, opening the way for substantial space savings in those settings. BuRR is at least twice as fast as all tried retrieval data structures.[8] The filter data structures that support counting and deletion (Cuckoo filters [24] and the related Morton filters [10] as well as the quotient filters QF [35] and CQF [5]) are slower than the best static AMQs.

[7] Small deviations of parameters are necessary because not all filters support arbitrary parameter choices. Also note that different filters have different functionality: (Blocked) Bloom allows dynamic insertion, Cuckoo, Morton and Quotient additionally allow deletion and counting. Xor [9, 19, 30, 37], Coupled [44], LMSS [34] and all ribbon variants are static retrieval data structures.

[8] FiRe [39] is likely to be faster but has two orders of magnitude higher overhead; see the full paper for more details.

The situation changes slightly when going to a sequential workload with large inputs as shown on the left of Figure 1. Blocked Bloom and BuRR are still the best filters for large and small overhead, respectively. But now homogeneous ribbon and (variants of) the hypergraph peeling based Xor filters [30, 19] share the middle-ground of the Pareto curve between them. Also, plain Bloom filters are almost dominated by Xor filters with half the overhead. The reason is that modern CPUs can handle several main memory accesses in parallel. This is very helpful for Bloom and Xor, whose queries do little else than computing the logical (x)or of a small number of randomly chosen memory cells. Nevertheless, the faster variants of BuRR are only moderately slower than Bloom and Xor filters while having at least an order of magnitude smaller overheads.

Further Results. Other claims supported by our data are:

- **Good ribbon widths are $w = 32$ and $w = 64$.** Ribbon widths as small as $w = 16$ can achieve small overhead but at least on 64-bit processors, $w \in \{32, 64\}$ seems most sensible. The case $w = 32$ is only 15–20 % faster than $w = 64$ while the latter has about four times less overhead. Thus the case $w = 64$ seems the most favorable one. This confirms that the linear dependence of the construction time on w is to some extent hidden behind the cache faults which are similar for both values of w (this is in line with our analysis in the external memory model).
- **Bu^{1}RR is slower than BuRR** by about 20 %, which may be a reasonable price for better worst-case query time in some real-time applications.[9]
- **The 1^+-bit variant of BuRR is smaller but slower** than the variant with 2-bit metadata per bucket, as expected, though not by a large margin.
- **Smaller inputs and smaller r change little.** For inputs that fit into cache, the Pareto curve is still dominated by blocked Bloom, homogeneous ribbon, and BuRR, but the performance penalty for achieving low overhead increases. For $r = 1$ we have data for additional competitors. GOV [28], which relies on structured Gaussian elimination, is several times slower than BuRR and exhibits an unfavorable time–overhead tradeoff. 2-block [18] uses two small dense blocks of nonzeroes and can achieve very small overhead at the cost of prohibitively expensive construction.
- **For large r, Xor filters and Cuckoo filters come into play.** Figure 6 shows the fastest AMQ depending on overhead and false positive rate $\varphi = 2^{-r}$ up to $r = 16$. While blocked Bloom, homogeneous ribbon, and BuRR cover most of the area, they lose ground for large r because their running time depends on r. Here Xor filters and Cuckoo filters make an appearance.
- **Bloom filters and Ribbon filters are fast for *negative queries*** where, on average, only two bits need to be retrieved to prove that a key is not in the set. This improves the relative standing of plain Bloom filters on large and parallel workloads with mostly negative queries.
- **Xor filters [30] and Coupled [44] have fast queries** since they can exploit parallelism in memory accesses. They suffer, however, from slow construction on large sequential inputs due to poor locality, and exhibit poor query performance when accessed from many threads in parallel. For small n, large r, and overhead between 8 % and 20 %, Coupled becomes the fastest AMQ.

[9] Part of the performance difference might be due to implementation details; see the full paper.

5 Related Results and Techniques

We now take the time to review some related work on retrieval including all approaches listed in Table 1.

Related Problems. An important application of retrieval besides AMQs is encoding perfect *hash functions* (PHF), i.e. an injective function $p : S \to [(1+\varepsilon)|S|]$ for given $S \subseteq \mathcal{U}$. Objectives for p are compact encoding, fast evaluation and small $\varepsilon \geq 0$. Consider a result from cuckoo hashing [25, 26, 33], namely that given four hash functions $h_1, h_2, h_3, h_4 : S \to [1.024|S|]$ there exists, with high probability, a choice function $f : S \to [4]$ such that $x \mapsto h_{f(x)}(x)$ is injective. A 2-bit retrieval data structure for f therefore gives rise to a perfect hash function [9]; see also [12]. Retrieval data structures can also be used to directly store compact names of objects, e.g., in column-oriented databases [39]. This takes more space than perfect hashing but allows to encode the ordering of the keys into the names.

In retrieval for AMQs and PHFs the stored values $f(x) \in \{0,1\}^r$ are uniformly random. However, some authors consider applications where $f(x)$ has a skewed distribution and the overhead of the retrieval data structure is measured with respect to the 0-*th order empirical entropy* of f [31, 4, 28]. Note that once we can do 1-bit retrieval with low overhead, we can use that to store data with prefix-free *variable-bit-length encoding* (e.g. Huffman or Golomb codes). We can store the k-th bit of $f(x)$ as data to be retrieved for the input tuple (x, k). This can be further improved by storing R 1-bit retrieval data structures where $R = \max_{x \in S} |f(x)|$ [31, 4, 28]. By interleaving these data structures, one can make queries almost as fast as in the case of fixed r.

More Linear Algebra based approaches. It has long been known that some matrices with random entries are likely to have full rank, even when sparse [13] and density thresholds for random k-XORSAT formulas to be solvable – either at all [23, 15] or with a linear time peeling algorithm [36, 32] – have been determined.

Building on such knowledge, a solution to the retrieval problem was identified by Botelho, Pagh and Ziviani [8, 7, 9] in the context of perfect hashing. In our terminology, their rows $\vec{h}(x)$ contain 3 random 1-entries per key which makes $AZ = \boldsymbol{b}$ solvable with peeling, provided $m > 1.22n$.

Several works develop the idea from [9]. In [27, 28] only $m > 1.089n$ is needed in principle (or $m > 1.0238n$ for $|\vec{h}(x)| = 4$) but a Gaussian solver has to be used. More recently in the *spatial coupling* approach [44] $\vec{h}(x)$ has k random 1-entries within a small window, achieving space overhead $\approx e^{-k}$ while still allowing a peeling solver. With some squinting, a class of linear erasure correcting codes from [34] can be interpreted as a retrieval data structure of a similar vein, where $|\vec{h}(x)| \in \{5, \ldots, k\}$ is random with expectation $\mathcal{O}(\log k)$.

Two recent approaches also based on sparse matrix solving are [18, 20] where $\vec{h}(x)$ contains two blocks or one block of random bits. Our ribbon approach builds on the latter.

We end this section with a discussion of seemingly promising techniques and give reasons why we choose not to use them in this paper. Some more details are also discussed in the experimental section of the full paper.

Shards. A widely used technique in hashing-based data structures is to use a splitting hash function to first divide the input set into many much smaller sets (shards, buckets, chunks, bins,...) that can then be handled separately [28, 3, 18, 20, 1, 40]. This incurs only linear time overhead during preprocessing and constant time overhead during a query, and

allows to limit the impact of superlinear cost of further processing to the size of the shard. Even to ribbon, this could be used in multiple ways. For example, by statically splitting the table into pieces of size n^ε for standard ribbon, one can achieve space overhead $\varepsilon + \mathcal{O}(n^{-\varepsilon})$, preprocessing time $\mathcal{O}(n/\varepsilon)$, and query time $\mathcal{O}(r)$ [20]. This is, however, underwhelming on reflection. Before arriving at the current form of BuRR, we designed several variants based on sharding but never achieved better overhead than $\Omega(1/w)$. The current overhead of $\mathcal{O}\big(\log w/w^2\big)$ comes from using the splitting technique in a "soft" way – keys are assigned to buckets for the purpose of defining bumping information but the ribbon solver may implicitly allocate them to subsequent buckets.

Table lookup. The first asymptotically efficient succinct retrieval data structure we are aware of [40] uses two levels of sharding to obtain very small shards of size $\mathcal{O}\big(\sqrt{\log n}\big)$ with small asymptotic overhead. It then uses dense random matrices per shard to obtain per-shard retrieval data structures. This can be done in constant time per shard by tabulating the solutions of all possible matrices. This leads to a multiplicative overhead of $\mathcal{O}\big(\log\log n/\sqrt{\log n}\big)$. Belazzougui and Venturini [4] use slightly larger shards of size $\mathcal{O}((1+\log\log(n)/r)\log\log(n)/\log n)$. Using carefully designed random lookup tables they show that linear construction time, constant lookup time, and overhead $\mathcal{O}\big((\log\log n)^2/\log n\big)$ is possible. We discussed on page 3 why we suspect large overhead for [40] and [4] in practice.

In general, lookup tables are often problematic for compressed data structures in practice – they cause additional space overhead and cache faults. Even if the table is small and fits into cache, this may yield efficient benchmarks but can still cause cache faults in practical workloads where the data structure is only a small part in a large software system with a large working set.

Cascaded bumping. Hash tables consisting of multiple shrinking levels are also used in *multilevel adaptive hashing* [11] and *filter hashing* [25]. While similar to BuRR in this sense, they do not maintain bumping information. This is fine for storing key-value pairs because all levels can be searched for a requested key. But it is unclear how the idea would work in the context of retrieval, i.e. without storing keys.

6 Conclusion and Future Work

BuRR is a considerable contribution to close a gap between theory and practice of retrieval and static approximate membership data structures. From the theoretical side, BuRR is succinct while achieving constant access cost for small number of retrieved bits ($r = \mathcal{O}(\log(n)/w)$). In contrast to previous succinct constructions with better asymptotic running times, its overhead is *tunable* and already small for realistic values of n. In practice, BuRR is faster than widely used data structures with much larger overhead and reasonably simple to implement. Our results further strengthen the success of linear algebra based solutions to the problem. Our on-the-fly approach shows that Gauss-like solvers can be superior to peeling-based greedy solvers even with respect to speed.

While the wide design space of BuRR leaves room for further practical improvements, we see the main open problems for large r. In practice, peeling based solvers (e.g., [44]) might outperform BuRR if faster construction algorithms can be found – perhaps using ideas like overloading and bumping. In theory, existing succinct data structures (e.g. [40, 4]) allow constant query time but have high space overhead for realistic input sizes. Combining constant cost per element for large r with small (preferably tunable) space overhead therefore remains a theoretical promise yet to be convincingly redeemed in practice.

References

1 Martin Aumüller, Martin Dietzfelbinger, and Michael Rink. Experimental variations of a theoretically good retrieval data structure. In *Proc. 17th ESA*, pages 742–751, 2009. doi:10.1007/978-3-642-04128-0_66.

2 Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms. *CoRR*, 2020. arXiv:2009.13569.

3 Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. Cache-oblivious peeling of random hypergraphs. In *Proc. DCC*, pages 352–361, 2014. doi:10.1109/DCC.2014.48.

4 Djamal Belazzougui and Rossano Venturini. Compressed static functions with applications. In Sanjeev Khanna, editor, *Proc. 24th SODA*, pages 229–240. SIAM, 2013. doi:10.1137/1.9781611973105.17.

5 Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, 2012. doi:10.14778/2350229.2350275.

6 Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. doi:10.1145/362686.362692.

7 Fabiano Cupertino Botelho. *Near-Optimal Space Perfect Hashing Algorithms*. PhD thesis, Federal University of Minas Gerais, 2008. URL: http://cmph.sourceforge.net/papers/thesis.pdf.

8 Fabiano Cupertino Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proc. 10th WADS*, pages 139–150, 2007. doi:10.1007/978-3-540-73951-7_13.

9 Fabiano Cupertino Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Inf. Syst.*, 38(1):108–131, 2013. doi:10.1016/j.is.2012.06.002.

10 Alex D. Breslow and Nuwan Jayasena. Morton filters: fast, compressed sparse cuckoo filters. *VLDB J.*, 29(2-3):731–754, 2020. doi:10.1007/s00778-019-00561-0.

11 Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In David S. Johnson, editor, *Proc. 1st SODA*, pages 43–53. SIAM, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320181.

12 Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proc. 15th SODA*, pages 30–39. SIAM, 2004. URL: http://dl.acm.org/citation.cfm?id=982792.982797.

13 Colin Cooper. On the rank of random matrices. *Random Structures & Algorithms*, 16(2):209–232, 2000. doi:10.1002/(SICI)1098-2418(200003)16:2<209::AID-RSA6>3.0.CO;2-1.

14 Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 79–94, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3035918.3064054.

15 Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In *Proc. 37th ICALP (1)*, pages 213–225, 2010. doi:10.1007/978-3-642-14165-2_19.

16 Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In *Proc. 35th ICALP (1)*, pages 385–396, 2008. doi:10.1007/978-3-540-70575-8_32.

17 Martin Dietzfelbinger and Michael Rink. Applications of a splitting trick. In *Proc. 36th ICALP (1)*, pages 354–365, 2009. doi:10.1007/978-3-642-02927-1_30.

18 Martin Dietzfelbinger and Stefan Walzer. Constant-time retrieval with $O(\log m)$ extra bits. In *Proc. 36th STACS*, pages 24:1–24:16, 2019. doi:10.4230/LIPIcs.STACS.2019.24.

19 Martin Dietzfelbinger and Stefan Walzer. Dense peelable random uniform hypergraphs. In *Proc. 27th ESA*, pages 38:1–38:16, 2019. doi:10.4230/LIPIcs.ESA.2019.38.

20 Martin Dietzfelbinger and Stefan Walzer. Efficient Gauss elimination for near-quadratic matrices with one short random block per row, with applications. In *Proc. 27th ESA*, pages 39:1–39:18, 2019. doi:10.4230/LIPIcs.ESA.2019.39.

21 Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership using ribbon. *CoRR*, abs/2106.12270, 2021. arXiv:2109.01892.

22 Peter C. Dillinger and Stefan Walzer. Ribbon filter: practically smaller than Bloom and Xor. *CoRR*, 2021. arXiv:2103.02515.

23 Olivier Dubois and Jacques Mandler. The 3-XORSAT threshold. In *Proc. 43rd FOCS*, pages 769–778, 2002. doi:10.1109/SFCS.2002.1182002.

24 Bin Fan, David G. Andersen, and Michael Kaminsky. Cuckoo filter: Better than Bloom. *;login:*, 38(4), 2013. URL: https://www.usenix.org/publications/login/august-2013-volume-38-number-4/cuckoo-filter-better-bloom.

25 Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005. doi:10.1007/s00224-004-1195-x.

26 Nikolaos Fountoulakis and Konstantinos Panagiotou. Sharp load thresholds for cuckoo hashing. *Random Struct. Algorithms*, 41(3):306–333, 2012. doi:10.1002/rsa.20426.

27 Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In *Proc. 15th SEA*, pages 339–352, 2016. doi:10.1007/978-3-319-38851-9_23.

28 Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of ([compressed] static | minimal perfect hash) functions. *Information and Computation*, 2020. doi:10.1016/j.ic.2020.104517.

29 Thomas Mueller Graf and Daniel Lemire. fastfilter_cpp, 2019. URL: https://github.com/FastFilter/fastfilter_cpp.

30 Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than Bloom and cuckoo filters. *ACM J. Exp. Algorithmics*, 25:1–16, 2020. doi:10.1145/3376122.

31 Jóhannes B. Hreinsson, Morten Krøyer, and Rasmus Pagh. Storing a compressed function with constant time access. In *Proc. 17th ESA*, pages 730–741, 2009. doi:10.1007/978-3-642-04128-0_65.

32 Svante Janson and Malwina J. Luczak. A simple solution to the k-core problem. *Random Struct. Algorithms*, 30(1-2):50–62, 2007. doi:10.1002/rsa.20147.

33 Marc Lelarge. A new approach to the orientation of random hypergraphs. In *Proc. 23rd SODA*, pages 251–264. SIAM, 2012. doi:10.1137/1.9781611973099.23.

34 Michael Luby, Michael Mitzenmacher, Mohammad Amin Shokrollahi, and Daniel A. Spielman. Efficient erasure correcting codes. *IEEE Trans. Inf. Theory*, 47(2):569–584, 2001. doi:10.1109/18.910575.

35 Tobias Maier, Peter Sanders, and Robert Williger. Concurrent expandable AMQs on the basis of quotient filters. In *Proc. 18th SEA*, pages 15:1–15:13, 2020. doi:10.4230/LIPIcs.SEA.2020.15.

36 Michael Molloy. Cores in random hypergraphs and Boolean formulas. *Random Struct. Algorithms*, 27(1):124–135, 2005. doi:10.1002/rsa.20061.

37 Thomas Mueller Graf and Daniel Lemire. Binary fuse filters: Fast and smaller than xor filters. *ACM Journal of Experimental Algorithmics*, 27, March 2022. doi:10.1145/3510449.

38 Ingo Müller, Cornelius Ratsch, and Franz Färber. Adaptive string dictionary compression in in-memory column-store database systems. In *Proc. 17th EDBT*, pages 283–294, 2014. doi:10.5441/002/edbt.2014.27.

39 Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and perfect hashing using fingerprinting. In *Proc. 14th SEA*, pages 138–149, 2014. doi:10.1007/978-3-319-07959-2_12.

40 Ely Porat. An optimal Bloom filter replacement based on matrix solving. In *Proc. 4th CSR*, pages 263–273, 2009. doi:10.1007/978-3-642-03351-3_25.

41 Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient Bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009. doi:10.1145/1498698.1594230.

42 Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. Partitioned learned bloom filter. *CoRR*, abs/2006.03176, 2020. arXiv:2006.03176.

43 Stefan Walzer. *Random Hypergraphs for Hashing-Based Data Structures*. PhD thesis, Technische Universität Ilmenau, 2020. URL: https://www.db-thueringen.de/receive/dbt_mods_00047127.

44 Stefan Walzer. Peeling close to the orientability threshold: Spatial coupling in hashing-based data structures. In *Proc. 32nd SODA*, pages 2194–2211. SIAM, 2021. doi:10.1137/1.9781611976465.131.

A Further Experimental Data

The following figures and tables contain

Figures 7 and 8. Performance-overhead trade-off of AMQs for very high false positive rate ($\approx 50\%$) and very low false positive rate ($\approx 0.01\%$) roughly corresponding to performance of 1-bit retrieval and 16-bit retrieval for the retrieval-based AMQs.

Figures 9–11. Performance-overhead trade-off of AMQs as in Figure 1, but seperately for positive queries, negative queries and construction.

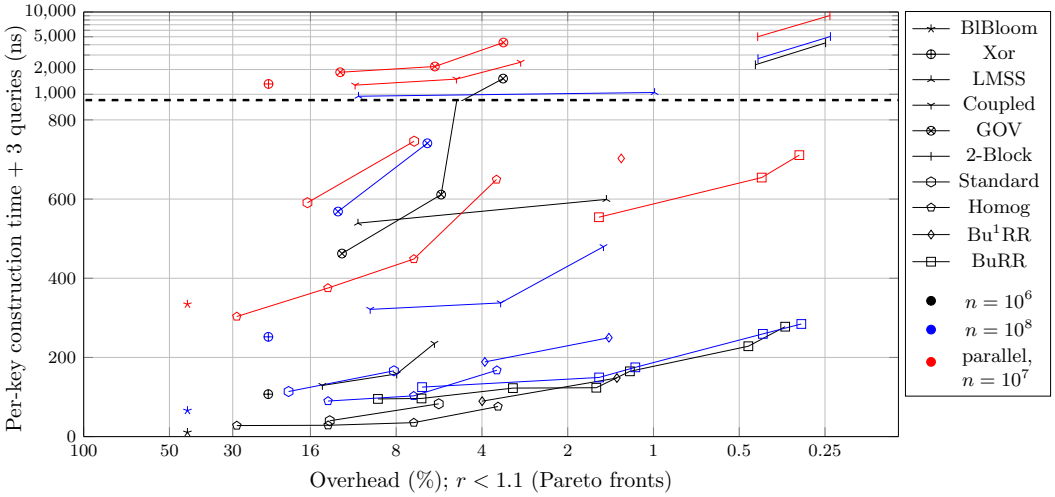


Figure 7 Performance–overhead trade-off for false-positive rate $> 46\,\%$ for different AMQs and different inputs. This large false-positive rate is the only one for which we have implementations for GOV [28] and 2-block [18]. Note that the vertical axis switches to a logarithmic scale above 900 ns.

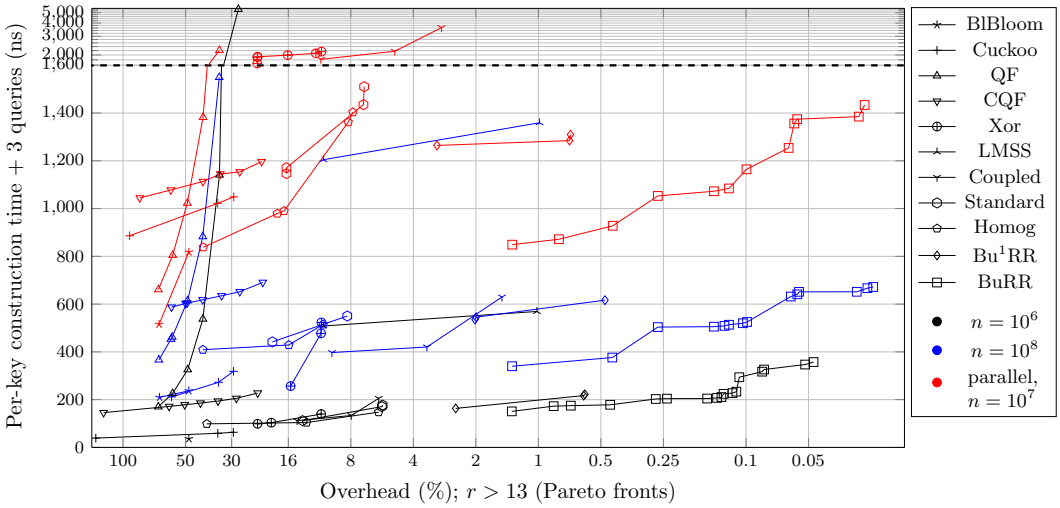


Figure 8 Performance–overhead trade-off for false-positive rate $< 2^{-13} \approx 0.01\,\%$ for different AMQs and different inputs. Logarithmics vertical axis above 1600 ns.

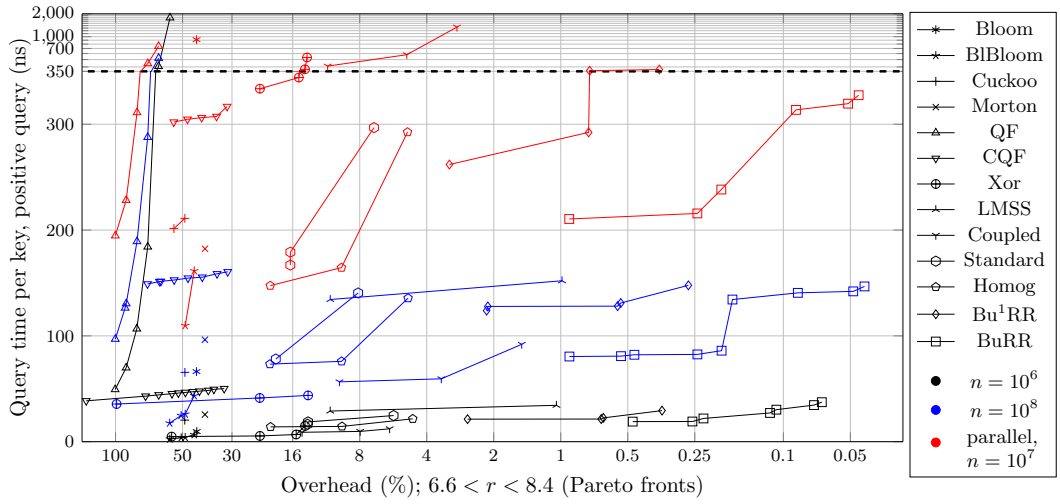


Figure 9 Query time–overhead trade-off for positive queries, false-positive rate between 0.3 % and 1 % for different AMQs and different inputs. Note that Xor filters have excellent query time sequentially where random fetches can be performed in parallel but are far from optimal in the parallel setting where the total number of memory accesses matters most. Logarithmic vertical axis above 350 ns.

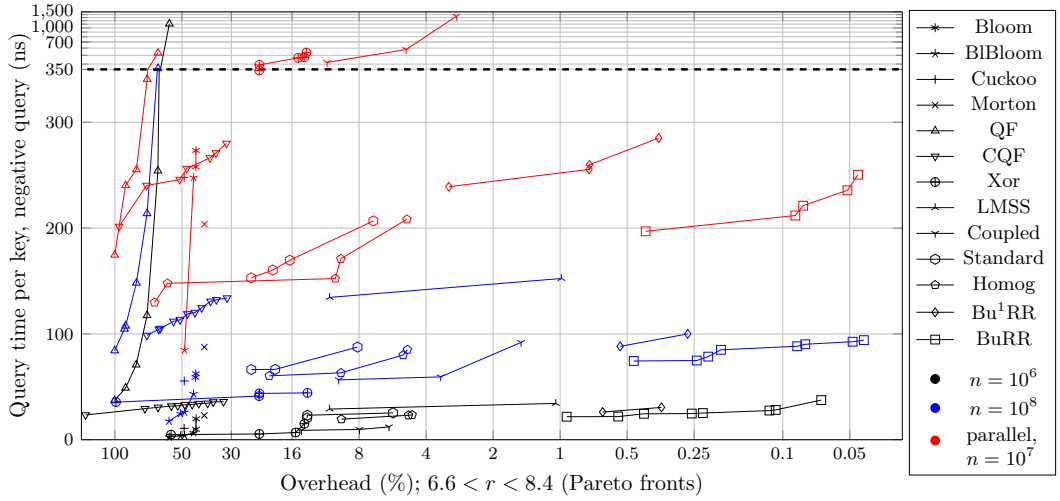


Figure 10 Query time–overhead trade-off for negative queries, false-positive rate between 0.3 % and 1 % for different AMQs and different inputs. Again, Xor filters perform well sequentially but suffer in the parallel case. Logarithmic vertical axis above 350 ns.

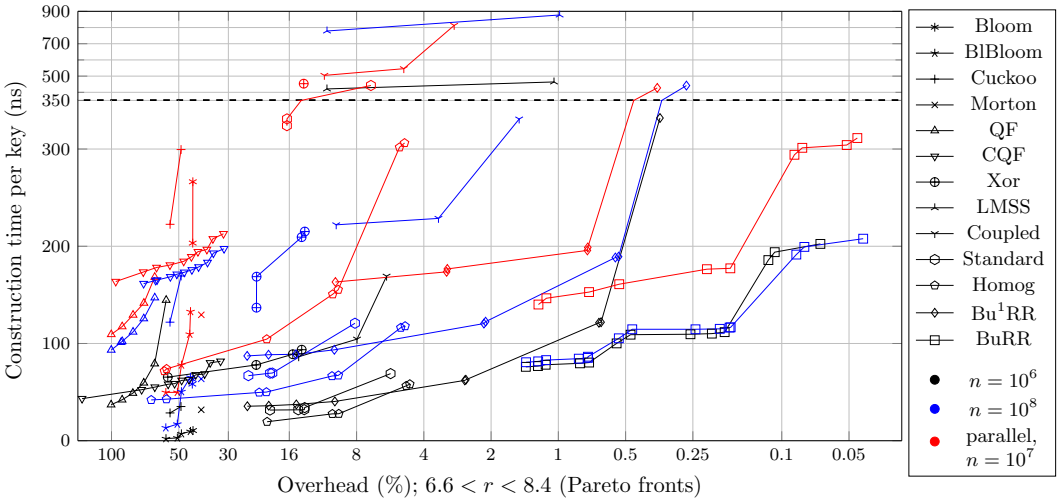


Figure 11 Construction time–overhead trade-off for false-positive rate between 0.3 % and 1 % for different AMQs and different inputs. Compressed vertical axis above 350 ns.

Parallel Flow-Based Hypergraph Partitioning

Lars Gottesbüren
Karlsruhe Institute of Technology, Karlsruhe, Germany

Tobias Heuer
Karlsruhe Institute of Technology, Karlsruhe, Germany

Peter Sanders
Karlsruhe Institute of Technology, Karlsruhe, Germany

Abstract

We present a shared-memory parallelization of *flow-based refinement*, which is considered the most powerful iterative improvement technique for hypergraph partitioning at the moment. Flow-based refinement works on bipartitions, so current sequential partitioners schedule it on different block pairs to improve k-way partitions. We investigate two different sources of parallelism: a parallel scheduling scheme and a parallel maximum flow algorithm based on the well-known push-relabel algorithm. In addition to thoroughly engineered implementations, we propose several optimizations that substantially accelerate the algorithm in practice, enabling the use on extremely large hypergraphs (up to 1 billion pins). We integrate our approach in the state-of-the-art parallel multilevel framework Mt-KaHyPar and conduct extensive experiments on a benchmark set of more than 500 real-world hypergraphs, to show that the partition quality of our code is on par with the highest quality sequential code (KaHyPar), while being an order of magnitude faster with 10 threads.

2012 ACM Subject Classification Mathematics of computing → Hypergraphs; Mathematics of computing → Graph algorithms

Keywords and phrases multilevel hypergraph partitioning, shared-memory algorithms, maximum flow

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.5

Related Version *Full Version*: `https://arxiv.org/abs/2201.01556`

Supplementary Material
Software (Multilevel Framework): `https://github.com/kahypar/mt-kahypar`
Software (Flow-Based Refinement): `https://github.com/larsgottesbueren/WHFC/tree/parallel`
Dataset (Benchmark Set & Experimental Results): `https://algo2.iti.kit.edu/heuer/sea22/`

Funding This work was partially supported by DFG grants WA654/19-2 and SA933/11-1. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

1 Introduction

Balanced hypergraph partitioning is a classical NP-hard optimization problem with numerous applications. Hypergraphs are a generalization of graphs, where each hyperedge can connect an arbitrary number of vertices. The problem is to partition the vertices of a hypergraph $H = (V, E, \omega)$ into k disjoint blocks $V_1, \dots V_k$ of roughly equal size $(\forall V_i : |V_i| \leq (1+\varepsilon)\frac{|V|}{k})$, such that an objective function defined on the hyperedges is minimized. In this work, we consider the connectivity metric $\sum_{e \in E}(\lambda(e)-1)\cdot\omega(e)$ where $\lambda(e) := |\{V_i \mid e \cap V_i \neq \emptyset\}|$ denotes the number of different blocks connected by hyperedge $e \in E$ and $\omega(e)$ denotes its weight. Often balanced partitioning is used as an acceleration technique for other applications, such as quantum circuit simulation [28], sharding distributed databases [14, 32], load balancing [12], route planning [16, 29], or boosting cache utilization in a search engine backend [7].

There is a substantial amount of literature, which is why we refer to survey articles [4, 9, 45, 50] for a summary. Most of the work focuses on heuristics, with the multilevel paradigm emerging as the most successful approach [2, 12, 18, 26, 34, 40, 49, 50]. Most partitioners use move-based heuristics such as label propagation [48] or variations of the Kernighan-Lin [37] or Fiduccia-Mattheyses [20] algorithms for local search. These heuristics move nodes greedily to different blocks according to their reduction in the objective function and are known to get stuck in local minima [38].

In this situation, maximum flows are an excellent tool as they correspond to (unbalanced) minimum cuts, thus offering a more global view than local move-based routines. Due to their complexity [55], they were long overlooked for partitioning, but have since enjoyed wide-spread adoption [5, 16, 29, 39, 49, 55] in many different algorithmic contexts.

Contribution. In this paper, we parallelize *flow-based refinement*, a powerful technique that is the last missing component in a series of works [26, 27] on parallelizing the state-of-the-art multilevel hypergraph partitioner KaHyPar [50]. Flow-based refinement operates on bipartitions, or on two blocks at a time if used for $k > 2$. Scheduling independent block pairs gives some trivial parallelism. One contribution we make is to improve the parallelism in the scheduler by relaxing certain constraints and showing how to deal with the resulting race conditions. For small k this is still insufficiently parallel, which is why we also parallelize the refinement on two blocks. We adapt an existing parallel flow algorithm to handle the incremental flow problems of the FlowCutter refinement algorithm [23, 24, 29]. Additionally, we engineer an efficient implementation, proposing several optimizations that reduce running time in practice, and fix a so far undocumented bug in the parallel flow algorithm. The result is a parallel partitioner that achieves the same solution quality as the highest quality sequential framework (KaHyPar), but in a fast parallel code. Using 10 threads, our code is an order of magnitude faster than sequential KaHyPar with flow-based refinement.

Outline. The paper is organized as follows. In Section 2 we introduce notation, terminology, and some algorithmic preliminaries. Section 3 briefly deals with related work. More details on additional related work are given in the main sections 5–8, closer to where particular parts are needed. In Section 4, we give an overview of the different components in the framework and how they interact. We complement the algorithmic discussion with extensive experiments in Section 9, before concluding in Section 10.

2 Preliminaries

Hypergraphs. A *weighted hypergraph* $H = (V, E, c, \omega)$ is defined as a set of vertices V and a set of hyperedges/nets E with vertex weights $c : V \to \mathbb{R}_{>0}$ and net weights $\omega : E \to \mathbb{R}_{>0}$, where each net e is a subset of the vertex set V. The vertices of a net are called its *pins*. We extend c and ω to sets in the natural way, i.e., $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. A vertex v is *incident* to a net e if $v \in e$. $\mathrm{I}(v)$ denotes the set of all incident nets of v. The *degree* of a vertex v is $\deg(v) := |\mathrm{I}(v)|$. The *size* $|e|$ of a net e is the number of its pins. We call two nets e_i and e_j *identical* if $e_i = e_j$. Given a subset $V' \subset V$, the *subhypergraph* $H_{V'}$ is defined as $H_{V'} := (V', \{e \cap V' \mid e \in E : e \cap V' \neq \emptyset\})$.

Balanced Hypergraph Partitioning. A *k-way partition* of a hypergraph H is a function $\Pi : V \to \{1, \ldots, k\}$. The blocks $V_i := \Pi^{-1}(i)$ of Π are the inverse images. We call Π *ε-balanced* if each block V_i satisfies the *balance constraint*: $c(V_i) \leq L_{\max} := (1 + \varepsilon)\lceil \frac{c(V)}{k} \rceil$ for

some parameter $\varepsilon \in (0, 1)$. For each net e, $\Lambda(e) := \{V_i \mid V_i \cap e \neq \emptyset\}$ denotes the *connectivity set* of e. The *connectivity* $\lambda(e)$ of a net e is $\lambda(e) := |\Lambda(e)|$. A net is called a *cut net* if $\lambda(e) > 1$. A node u that is incident to at least one cut net is called *boundary node*. The number of pins of a net e in block V_i is denoted by $\Phi(e, V_i) := |e \cap V_i|$. The *quotient graph* $\mathcal{Q} := (\Pi, E_\Pi := \{(V_i, V_j) \mid \exists e \in E : \{V_i, V_j\} \subseteq \Lambda(e)\})$ contains an edge between each pair of adjacent blocks of a k-way partition Π. Given parameters ε and k, and a hypergraph H, the *balanced hypergraph partitioning problem* is to find an ε-balanced k-way partition Π that minimizes an objective function defined on the hyperedges. In this work, we minimize the *connectivity metric* $(\lambda - 1)(\Pi) := \sum_{e \in E} (\lambda(e) - 1)\, \omega(e)$.

Flows. A flow network $\mathcal{N} = (\mathcal{V}, \mathcal{E}, c)$ is a directed graph with a dedicated source $s \in \mathcal{V}$ and sink $t \in \mathcal{V}$ in which each edge $e \in \mathcal{E}$ has capacity $c(e) \geq 0$. An (s, t)-flow is a function $f : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$ that satisfies the *capacity constraint* $\forall u, v \in \mathcal{V} : f(u, v) \leq c(u, v)$, the *skew symmetry constraint* $\forall u, v \in \mathcal{V} : f(u, v) = -f(v, u)$ and the *flow conservation constraint* $\forall u \in \mathcal{V} \setminus \{s, t\} : \sum_{v \in \mathcal{V}} f(u, v) = 0$. The value of a flow $|f| := \sum_{v \in \mathcal{V}} f(s, v) = \sum_{v \in \mathcal{V}} f(v, t)$ is defined as the total amout of flow transferred from s to t. An (s, t)-flow f is a maximum (s, t)-flow if there exists no other (s, t)-flow f' with $|f| < |f'|$. The *residual capacity* is defined as $r_f(e) = c(e) - f(e)$. An edge e is *saturated* if $r_f(e) = 0$. $\mathcal{N}_f = (\mathcal{V}, \mathcal{E}_f, r_f)$ with $\mathcal{E}_f := \{(u, v) \in \mathcal{V} \times \mathcal{V} \mid r_f(u, v) > 0\}$ is the *residual network*. The max-flow min-cut theorem states that the value $|f|$ of a maximum (s, t)-flow equals the weight of a minimum cut that separates s and t [21]. This is also called a *minimum* (s, t)*-cut*. The source-side cut can be computed by exploring the nodes reachable from the source via residual edges (for example via BFS), and analogously the sink-side cut from the sink.

Push-Relabel Algorithm. The *push-relabel* [22] maximum flow algorithm stores a distance label $d(u)$ and an excess value $\text{exc}(u) := \sum_{v \in \mathcal{V}} f(v, u)$ for each node. It maintains a *preflow* [36] which is a flow where the conservation constraint is replaced by $\text{exc}(u) \geq 0$. A valid distance labeling is defined by the conditions $\forall (u, v) \in \mathcal{E}_f d(u) \leq d(v) + 1$, $d(s) = |\mathcal{V}|$, $d(t) = 0$. A node $u \in \mathcal{V}$ is *active* if $\text{exc}(u) > 0$. An edge $(u, v) \in \mathcal{E}$ is *admissible* if $r_f(u, v) > 0$ and $d(u) = d(v) + 1$. A *push*(u, v) operation sends $\delta = \min(\text{exc}(u), r_f(u, v))$ flow units over (u, v). It is applicable if u is active and (u, v) is admissible. A *relabel*(u) operation updates the distance label of u to $\min(\{d(v) + 1 \mid r_f(u, v) > 0\})$, which is applicable if u is active, and has no admissible edges. The distance labels are initialized to $\forall u \in \mathcal{V} \setminus \{s\} : d(u) = 0$ and $d(s) = |V|$ and all source edges are saturated. Efficient variants use the *discharge* routine, which repeatedly scans the edges of an active node until its excess is zero. All admissible edges are pushed and at the end of a scan, the node is relabeled. Discharging active nodes in FIFO order results in an $\mathcal{O}(|\mathcal{V}|^3)$ time algorithm. The *global relabeling* heuristic [13] frequently assigns exact distance labels by performing a reverse BFS from the sink, to reduce relabel work in practice. Note that preflows already induce minimum sink-side cuts, so if only a minimum cut is required, the algorithm can already stop once no active nodes with distance label $< n$ exist.

Flows on Hypergraphs. The *Lawler expansion* [41] of a hypergraph $H = (V, E, c, \omega)$ is a graph consisting of V and two nodes $e_{\text{in}}, e_{\text{out}}$ for each $e \in E$, with directed edges $\forall u \in V, e \in \text{I}(u) : (u, e_{\text{in}}), (e_{\text{out}}, u)$ with infinite capacity and *bridge edges* $\forall e \in E : (e_{\text{in}}, e_{\text{out}})$ with capacity $\omega(e)$. A minimum (s, t)-cut in the Lawler expansion directly corresponds to one in the hypergraph (since only bridging edges have finite capacity).

Algorithm 1 Parallel Flow-Based Refinement.

Input: Hypergraph $H = (V, E, c, \omega)$, k-way partition Π of H

1 $\mathcal{Q} \leftarrow$ `buildQuotientGraph`(H, Π) *// Section 5*
2 **while** $\exists$ *active* $(V_i, V_j) \in \mathcal{Q}$ **do in parallel** *// Section 5*
3 $B := B_i \cup B_j \leftarrow$ `constructRegion`(H, V_i, V_j) *// $B_i \subseteq V_i, B_j \subseteq V_j$, Section 6*
4 $(\mathcal{N}, s, t) \leftarrow$ `constructFlowNetwork`(H, B) *// Section 6*
5 $(M, \Delta_{\text{exp}}) \leftarrow$ `FlowCutterRefinement`$(\mathcal{N}, s, t)$ *// Section 7 and 8*
6 **if** $\Delta_{exp} \geq 0$ *// potential improvement*
7 $\Delta_{\lambda-1} \leftarrow$ `applyMoves`(H, Π, M) *// Section 5*
8 **if** $\Delta_{\lambda-1} > 0$ mark V_i and V_j as active *// found improvement*
9 **else if** $\Delta_{\lambda-1} < 0$ `revertMoves`(H, Π, M) *// no improvement*

3 Related Work

The most well-known sequential algorithms are PaToH [12], hMetis [34, 35], and KaHyPar [1, 23, 51]. Notable parallel algorithms are Parkway [53] and Zoltan [18] for distributed memory, as well as BiPart [42] and Mt-KaHyPar [26, 27] for shared memory.

All of these follow the multilevel paradigm that proceeds in three phases: First, the hypergraph is *coarsened* to obtain a hierarchy of successively smaller and structurally similar hypergraphs by *contracting* pairs or clusters of vertices. Once the coarsest hypergraph is small enough, an *initial partition* into k blocks is computed. Subsequently, the contractions are reverted level-by-level, and, on each level, *local search* heuristics are used to improve the partition from the previous level (*refinement phase*).

Sanders and Schulz [49] propose an algorithm to improve the edge cut of bipartitions with flow-based refinement. Their general idea is to grow a size-constrained region around the cut edges of a bipartition. Afterwards, they compute a minimum (s, t)-cut in the subgraph induced by the region and apply it to the original graph, if it satisfies the balance constraint. They extended their algorithm to k-way partitions by scheduling it on pairs of adjacent blocks. Heuer et al. [30] integrated this approach into their hypergraph partitioner KaHyPar. This was improved by Gottesbüren et al. [23] by replacing the bipartitioning routine with FlowCutter [24, 29]. Flow-based refinement substantially improved the solution quality (cut and connectivity metric) of the partitions produced by KaHyPar, making it the method of choice for high-quality hypergraph partitioning [50]. We explain the flow-based refinement routine of KaHyPar in more detail in the main part.

4 Framework Overview

We first give an overview of how the different framework components interact, before providing descriptions in their respective sections. For this we follow the high level structure shown in Algorithm 1. We start with a parallel scheduling scheme of block pairs based on the quotient graph in Section 5, see line 1 and 2. For each block pair, we extract a subhypergraph constructed around the boundary nodes of the blocks, which yields a flow network, see line 3 and 4 and Section 6. On each network we run FlowCutter (line 5), whose partition we convert into a set of moves M and an expected connectivity reduction Δ_{exp}. FlowCutter and its parallelization are discussed in Section 7 and 8 respectively. If FlowCutter claims an improvement, i.e., $\Delta_{\text{exp}} \geq 0$, we apply the moves to the global partition and compute the exact reduction $\Delta_{\lambda-1}$, based on which we either mark the blocks for further refinement,

or revert the moves, see line 8 and 9. We distinguish between expected Δ_{exp} and actual improvement $\Delta_{\lambda-1}$, due to concurrency conflicts that arise in the scheduler, which is described again in Section 5.

5 Parallel Active Block Scheduling

Sanders and Schulz [49] propose the active block scheduling algorithm to apply their flow-based refinement algorithm for bipartitions on k-way partitions. Their algorithm proceeds in rounds. In each round, it schedules all pairs of adjacent blocks where at least one is marked as *active*. Initially, all blocks are marked as active. If a search on two blocks improves the edge cut, both are marked as active for the next round.

Parallelization. A simple scheme would be to schedule block pairs that form a maximum matching in the quotient graph $\mathcal{Q}$ in parallel. This allows searches to operate in independent regions of the hypergraph and thus avoids conflicts between different block pairs. However, this scheme restricts the available parallelism to at most $\frac{k}{2}$ threads. Thus, we do not enforce any constraints on the block pairs processed concurrently, e.g., there can be multiple threads running on the same block and they can also share some of their nodes as illustrated in Figure 1 (right). We use $\min(t, \tau \cdot k)$ threads to process the active block pairs in parallel, where t is the number of available threads in the system. The parameter τ controls the available parallelism in the scheduler. With higher values of τ, more block pairs are scheduled in parallel. This can lead to interference between searches that operate on overlapping regions. Lower values for τ can reduce these conflicts but put more emphasis on good parallelization of 2-way refinement to achieve good speedups. In practice, we choose $\tau = 1$.

Our parallel active block scheduling algorithm uses one concurrent FIFO queue A to schedule active block pairs. Each block pair is associated with a round and each round uses an array of size k to mark blocks that become active in the next round. If a search finds an improvement on two blocks (V_i, V_j) and V_i or V_j becomes active, we push all adjacent blocks into A if they are not contained yet (marked using an atomic test-and-set instruction). If either V_i or V_j is already active, we insert (V_i, V_j) into A, if it is not already contained in A. Thus, active block pairs of different rounds are stored interleaved in A and the end of a round does not induce a synchronization point as in the original algorithm [49]. For processing in the first round, we sort active block pairs in descending order of improvement they contributed on previous levels, with ties broken by larger cut size. A round ends when all of its block pairs have been processed and all prior rounds have ended. If the relative improvement at the end of a round is less than 0.1%, we immediately terminate the algorithm. In Appendix A, we describe how to construct and maintain the cut hyperedges between block pairs that induce the quotient graph and are used for the network construction.

Apply Moves. We integrate flow-based refinement into Mt-KaHyPar [26, 27], which provides data structures to concurrently access and modify the partition Π, block weights $c(V_i)$, connectivity sets $\Lambda(e)$ and pin counts $\Phi(e, V_i)$ of each $e \in E$ and V_i. When applying a move sequence M to the global partition Π (each move $m := (u, V_i, V_j) \in M$ moves a node u from its current block V_i to V_j), there are three conflict types that can occur: balance constraint violations, $\Delta_{\lambda-1} \neq \Delta_{\text{exp}}$, i.e., the expected does not match the actual connectivity reduction, and nodes may no longer be in the block expected by M. These conflicts arise, because concurrently scheduled block pairs are not independent, which causes data races on the partition assignment Π, pin counts $\Phi(e, V_i)$ and connectivity sets $\Lambda(e)$. These are concurrently

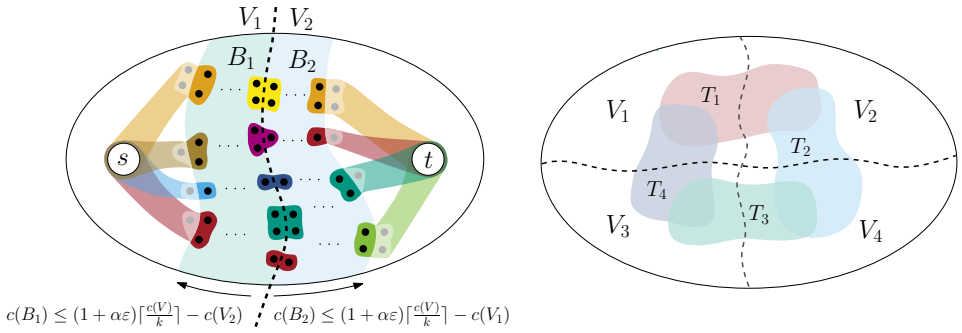


Figure 1 Illustrates the flow network construction algorithm (left) and an how we schedule block pairs of the quotient graph $\mathcal{Q}$ in parallel (right, T_i denotes the search region of thread i).

read by the network construction and modified when moves are applied. Updates after the construction are not observed, and thus the state at the time a refinement finishes may differ from the expected state. Since the running time to apply moves is negligible compared to solving flow problems (see Figure 9 in Section 9), we can afford to use a lock so that only one thread applies moves at a time to address these conflicts. We remove all nodes from M that are not in their expected block. Afterwards, we compute the block weights if all remaining moves were applied. If balanced, we perform the moves, during which we compute $\Delta_{\lambda-1}$. For $m = (u, V_i, V_j)$ and $e \in I(u)$, we add $\omega(e)$ to $\Delta_{\lambda-1}$ if $\Phi(e, V_i)$ decreases to zero and $-\omega(e)$ if $\Phi(e, V_j)$ increases to one (connectivity metric is $\sum_{e \in E}(\lambda(e) - 1)\,\omega(e)$). If $\Delta_{\lambda-1} < 0$, we revert all moves.

Implementation Details. KaHyPar [30] established pruning rules to skip unpromising flow computations that we use as well: skip if cut is small or no improvement found on previous levels. We additionally introduce a time limit to abort long-running flow computations.

6 Network Construction

To improve the cut of a bipartition $\Pi = \{V_1, V_2\}$, we grow a size-constrained region B around the cut hyperedges of Π. We then contract all nodes in $V_1 \setminus B$ to the source s and $V_2 \setminus B$ to the sink t [24, 49] as illustrated in Figure 1 (left) and obtain a coarser hypergraph $\mathcal{H}$. We implemented two parallel algorithms to construct $\mathcal{H}$, which are preferable in different situations. These are described in more detail in Appendix B. The flow network $\mathcal{N}$ is then given by the Lawler expansion of $\mathcal{H}$ (see Section 2). Note that reducing the hyperedge cut of a bipartition induced by two adjacent blocks of a k-way partition Π_k optimizes the connectivity metric of Π_k [30].

Sanders and Schulz [49] grow a region $B := B_1 \cup B_2$ with $B_1 \subseteq V_1$ and $B_2 \subseteq V_2$ around the cut hyperedges of Π via two breadth-first-searches (BFS) as illustrated in Figure 1 (left). The first BFS is initialized with all boundary nodes of block V_1 and continues to add nodes to B_1 as long as $c(B_1) \leq (1+\alpha\varepsilon)\lceil\frac{c(V_1)+c(V_2)}{2}\rceil - c(V_2)$, where α is an input parameter. The second BFS that constructs B_2 proceeds analogously. For $\alpha = 1$, each flow computation yields a balanced bipartition with a possibly smaller cut in the original hypergraph, since only nodes of B can move to the opposite block ($c(B_1) + c(V_2) \leq (1+\varepsilon)\lceil\frac{c(V_1)+c(V_2)}{2}\rceil$ and vice versa). Larger values for α lead to larger flow problems with potentially smaller minimum cuts, but also increase the likelihood of violating the balance constraint. However, this is not a problem since the flow-based refinement routine guarantees balance through incremental minimum

Algorithm 2 FlowCutter Core.

```
1  S ← {s}, T ← {t}
2  while true do
3  |  augment flow to maximality regarding S, T
4  |  derive source- and sink-side cut S_r, T_r ⊂ V
5  |  if (S_r, V \ S_r) or (V \ T_r, T_r) balanced
6  |  |  return balanced partition
7  |  if c(S_r) ≤ c(T_r)
8  |  |  S ← S_r ∪ selectPiercingNode()
9  |  else
10 |  |  T ← T_r ∪ selectPiercingNode()
```

cut computations (see Section 7). In practice, we use $\alpha = 16$ (also used in KaHyPar [23, 30]). We additionally restrict the distance of each node $v \in B$ to the cut hyperedges to be smaller than or equal to a parameter δ $(= 2)$. We observed that it is unlikely that a node *far* way from the cut is moved to the opposite block by the flow-based refinement.

7 Flow-Based Refinement

In this section we discuss the flow-based refinement on a bipartition. We introduce the aforementioned FlowCutter algorithm [29, 55]. It is parallelized by plugging in a parallel maximum flow algorithm, which we discuss in the next section. To speed up convergence and make parallelism worthwhile, we propose an optimization named *bulk piercing*.

Core Algorithm. FlowCutter solves a sequence of incremental maximum flow problems until a balanced bipartition is found. Algorithm 2 shows pseudocode for the approach. In each iteration, first the previous flow (initially zero) is augmented to a maximum flow regarding the current source set S and sink set T. Subsequently, the node sets $S_r, T_r \subset \mathcal{V}$ of the source- and sink-side cuts are derived. This is done via residual (parallel) BFS (forward from S for S_r, backward from T for T_r). The node sets induce two bipartitions $(S_r, \mathcal{V} \setminus S_r)$ and $(\mathcal{V} \setminus T_r, T_r)$. If neither is balanced, all nodes on the side with smaller weight are transformed to a source (if $c(S_r) \leq c(T_r)$) or a sink otherwise. To find a different cut in the next iteration, one additional node is added, called *piercing node*. Thus, the bipartitions contributed by the currently smaller side will be more balanced in future iterations. Since the smaller side is grown, this process will converge to a balanced partition.

Piercing. For our purpose, there are two important piercing node selection heuristics: *avoid augmenting paths* [29, 55] and *distance from cut* [23]. Whenever possible, a node that is not reachable from the source or sink should be picked, i.e., $v \in \mathcal{V} \setminus (S_r \cup T_r)$. Such nodes do not increase the weight of the cut, while improving balance. As a secondary criterion, larger distances from the original cut are preferred, to reconstruct parts of it.

Most Balanced Cut. Once the partition is balanced, we continue to pierce as long as the cut does not increase. This is repeated with different random choices since it is fast (no flow augmentation). More balance gives other refinement algorithms more leeway for improvement. An equivalent heuristic was already employed in previous flow-based refinement [47, 49].

Bulk Piercing. The complexity of FlowCutter is $O(\zeta m)$, where ζ is the final cut weight, and $m = |\mathcal{E}|$. This bound stems from a pessimistic implementation that augments one flow unit in $O(m)$ work [29, 55]. For refinement, the performance is much better in practice, as the first cut is often close to the final cut. Only few augmenting iterations are needed and much less than $O(m)$ work is spent per flow unit [24], with most work spent on the initial flow.

Still, the flow augmented per iteration is often small: at most the capacity of edges incident to the piercing node. On large instances, we observed that the number of required iterations increases substantially. We propose to accelerate convergence by piercing multiple nodes per iteration, as long as we cannot avoid augmenting paths and are far from balance. To ensure a poly-log iteration bound, we set a geometrically shrinking goal of weight to add to each side per iteration. The initial goal for the source side is set to $\beta(\frac{c(\mathcal{V})}{2} - c(s))$, where $\beta \in (0, 1)$ is the geometric shrinking factor that is multiplied with the term in each iteration, and $\frac{c(\mathcal{V})}{2} - c(s)$ is the weight to add for perfect balance.

If a goal is not met, its remainder is added to next iteration's goal. We track the average weight added per node and from this estimate the number of piercing nodes needed to match the current goal. To boost measurement accuracy, we pierce one node for the first few rounds. The sides have distinct measurements and goals, so that we do not pierce too aggressively when the smaller side flips. This scheme (with $\beta = 0.55$) reduces running time on our largest instances from beyond two hours (time limit) to less than 10 minutes, while not incurring any quality penalties on either small or large instances, as shown in our technical report [25].

8 Parallel Maximum Flow Algorithm

Maximum flow algorithms are notoriously difficult to parallelize efficiently [6, 10, 33, 52]. The synchronous push-relabel approach of Baumstark et al. [10] is a recent algorithm that sticks closely to sequential FIFO and thus shows good results. We first outline their algorithm, then describe a so far undocumented bug followed by our fix, and conclude with implementation details and intricacies of using FlowCutter with preflows. Note that a maximum preflow already yields a minimum cut, which suffices for our purpose.

Synchronous Parallel Push-Relabel. The algorithm proceeds in rounds in which all active nodes are discharged in parallel. The flow is updated globally, the nodes are relabeled locally and the excess differences are aggregated in a second array using atomic instructions. After all nodes have been discharged, the distance labels d are updated to the local labels d' and the excess deltas are applied. The discharging operations thus use the labels and excesses from the previous round. This is repeated until there are no nodes with $\text{exc}(v) > 0$ and $d(v) < n$ left. To avoid concurrently pushing flow on residual arcs in both directions (race condition on flow values), a deterministic winning criterion on the old distance labels is used to determine which direction to push, if both nodes are active. If an arc cannot be pushed due to this, the discharge terminates after the current scan, as the node may not be relabeled in this round. The rounds are interleaved with global relabeling [13], after linear push and relabel work, using parallel reverse BFS.

A Bug in the Synchronous Algorithm. The parallel discharge routine does not protect against push-relabel conflicts [33] as illustrated in Figure 2. In particular the winning criterion does not help. A node u may be relabeled too high if it is concurrently pushed to through a residual arc (v, u) with $d'(v) = d(u) + 1$. The arc (u, v) may not be observed as residual yet, and thus u may set its new label $d'(u) > d'(v) + 1$, violating label correctness. The bug

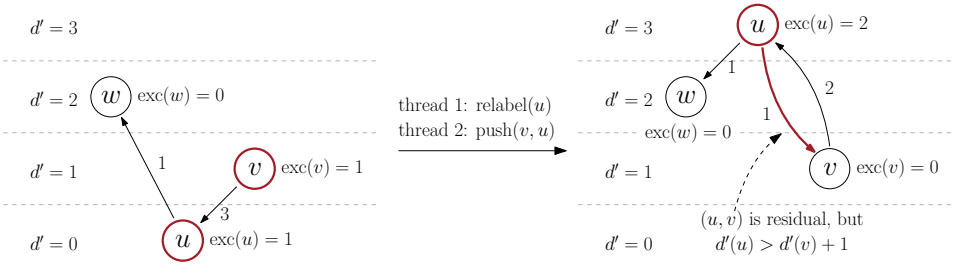


Figure 2 Illustrates a push-relabel conflict in the parallel discharge routine (adapted from Ref. [33]). The numbers on the arcs denote their residual capacities.

becomes noticeable when the algorithm terminates prematurely with incorrect distances. Our fix is to collect mislabeled excess nodes during global relabeling. When the algorithm would terminate, we run global relabeling, and restart the main loop if new active nodes are found. The additional work is already accounted for, because we need to extract the sink-side cut anyways.

Restricting Capacities. Recall that only bridge edges $(e_{\text{in}}, e_{\text{out}})$ have finite capacity ($\omega(e)$) in the Lawler network. Since $(e_{\text{in}}, e_{\text{out}})$ is the only outgoing edge of e_{in} with non-zero capacity, the flow (but not preflow) on edges (u, e_{in}) is also bounded by $\omega(e)$. Adding these capacities during the preflow stage is a trivial optimization, but it reduces running time for one flow computation on our largest instance from over two hours to 14 seconds, when using 16 cores. It also boosts the available parallel work, since hypernodes are not immediately relieved of all their excess. Without this optimization the minimum cut contains only bridge edges, but now may contain edges (u, e_{in}). This matters when tracking cut hyperedges (for collecting piercing candidates), which are detected by checking if e_{in} and e_{out} are on different sides. Therefore, we do not check the capacity and visit e_{in} nodes during forward residual BFSs.

Avoid Pushing Flow Back. Once the correct flow value is found, the algorithm could terminate in theory. This is often achieved in very few discharging rounds ($< 1\%$). Furthermore, we observed that the number of active nodes follows a power law distribution. At this point flow is only pushed back to the source. We terminate once all nodes with $\text{exc}(u) > 0$ have $d(u) \geq n$, which is most often detected by global relabeling. Due to little work per round, it takes many rounds to trigger. We perform additional relabeling, if the flow value has not changed for some rounds (500), and only few active nodes (< 1500) were available in each.

Source-Side Cut. A maximum preflow only yields a sink-side cut via the reverse residual BFS, but we also need the source-side cut. We can run flow decomposition [13] to push excess back to the source, to obtain an actual flow. However, flow decomposition is difficult to parallelize [10]. Instead, we initialize the forward residual BFS with all non-sink excess nodes. This finds the reverse paths that carry flow from the source to the excess nodes, which is what we need.

Sink-Side Piercing. Furthermore, when transforming a node with positive excess to a sink, its excess must be added to the flow value. This only happens when piercing, as sink-side nodes have no excess, if they are not sinks yet.

Algorithm 3 Parallel Multilevel Hypergraph Partitioning (Mt-KaHyPar-D-F).

Input: Hypergraph $H = (V, E)$, number of blocks k
Output: k-way partition Π of H

```
1  H_1 ← H, 𝓗 ← ⟨H_1⟩, i ← 1
2  while |V_i| is not small enough do                    // Coarsening Phase
3  |  C ← compute node clustering  // Uses the heavy-edge rating function [1, 12, 34]
4  |  H_{i+1} ← H_i.contract(C), 𝓗 ← 𝓗 ∪ ⟨H_{i+1}⟩, i ← i + 1
5  Π ← initialPartition(H_i, k)
6  for l = i to 1 do                                     // Uncoarsening Phase
7  |  Π ← project Π onto H_l
8  |  while improvement relevant do
9  |  |  Π ← labelPropagationRefinement(H_l, Π)
10 |  |  Π ← fmLocalSearch(H_l, Π)
   |  |   // Extends Mt-KaHyPar-D with flow-based refinement
11 |  |  Π ← flowBasedRefinement(H_l, Π)
12 return Π
```

Maintain Distance Labels. Finally, we want to reuse the distance labels to avoid re-initialization overheads. However, as the labels are a lower bound on the distance from the sink, piercing on the sink side invalidates the labels. Additionally, no new excess nodes are created. In this case, we run global relabeling to fix the labels and collect the existing excess nodes, before starting the main discharge loop. When piercing on the source side the labels remain valid and new excesses are created. These are added to the active nodes and we do not run an additional global relabeling. The existing excess nodes are collected during regular global relabel runs; at the latest for the termination check.

9 Experiments

We implemented the flow-based refinement routine in the shared-memory hypergraph partitioner Mt-KaHyPar[1], which is implemented in C++17, parallelized using the TBB library [46], and compiled using g++9.2 with the flags `-O3 -mtune=native -march=native`. Mt-KaHyPar provides two partitioners: Mt-KaHyPar-D [26] (**D**efault setting) opts for the traditional $\mathcal{O}(\log n)$ level approach by contracting a vertex clustering on each level and Mt-KaHyPar-Q [27] (**Q**uality setting) implements a parallel version of the n-level scheme [1, 44, 51] that (un)contracts only a single vertex on each level. We refer to the corresponding versions that use flow-based refinement as Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F (-**F**lows, integration is described in the next paragraph). For parallel partitioners we add a suffix to their name to indicate the number of threads used, e.g. Mt-KaHyPar-Q-F 64 for 64 threads. We omit the suffix for sequential partitioners. Note that we performed extensive parameter tuning experiments (e.g., τ, β, δ and the effects of bulk piercing) which we present only in the technical report [25] due to space constraints.

The Multilevel Partitioning Algorithm. The high-level pseudocode of Mt-KaHyPar-D is shown in Algorithm 3. Mt-KaHyPar-D uses a clustering-based coarsening algorithm and parallel multilevel recursive bipartitioning with work-stealing to compute an initial

[1] Mt-KaHyPar is available from https://github.com/kahypar/mt-kahypar

k-way partition of the coarsest hypergraph [26, 27]. In each refinement step, we first run label propagation refinement [48] followed by a highly-localized version of the FM algorithm [1, 2, 20, 49] (see Line 9 and 10). We run our flow-based refinement as the third component. We run all refinement algorithms on each level multiple times in combination and stop if the relative improvement is less than 0.25%.

Instead of contracting a node clustering, Mt-KaHyPar-Q contracts only a single vertex on each level. In the uncoarsening phase, it assembles independent contractions in a batch and uncontracts them in parallel. This induces a hierarchy with $\mathcal{O}(|V|)$ levels and refinement steps, which would incur too much overhead when we use flow-based refinement. Thus, we use approximately $\mathcal{O}(\log n)$ synchronization points similar to Mt-KaHyPar-D and perform FM local search followed by our flow-based refinement. We do not run label propagation here since there were no quality benefits.

Setup. For comparison with sequential partitioners, we use the established benchmark set of Heuer and Schlag [31] (referred to as set A, 488 hypergraphs). For these experiments, we use $k \in \{2, 4, 8, 16, 32, 64, 128\}$, $\varepsilon = 0.03$, ten different seeds and a time limit of eight hours. The experiments are done on a cluster of Intel Xeon Gold 6230 processors (2 sockets with 20 cores each) running at 2.1 GHz with 96GB RAM (machine A). To measure speedups and to compare our implementation with other parallel partitioners, we use a benchmark set composed of 94 large hypergraphs (referred to as set B) that was initially assembled to evaluate Mt-KaHyPar-D [26]. On set B, we evaluate $k \in \{2, 8, 16, 64\}, \varepsilon = 0.03$ and use three seeds each with a time limit of two hours. These experiments are run on an AMD EPYC Rome 7702P (one socket with 64 cores) running at 2.0–3.35 GHz with 1024GB RAM (machine B). The parameter space on set B is restricted, since we only have access to one machine of type B. We describe the sources and properties of the instances in Appendix C[2].

Methodology. Each partitioner optimizes the connectivity metric, which we also refer to as the quality of a partition. For each instance (hypergraph and k), we aggregate running times using the arithmetic mean over all seeds. To further aggregate over multiple instances, we use the geometric mean for absolute running times and self-relative speedups. For runs that exceeded the time limit, we use the time limit itself in the aggregates. In plots, we mark these instances with ⏲ if *all* runs of that algorithm timed out.

To compare the solution quality of different algorithms, we use *performance profiles* [19]. Let $\mathcal{A}$ be the set of algorithms we want to compare, $\mathcal{I}$ the set of instances, and $q_A(I)$ the quality of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. For each algorithm A, we plot the fraction of instances (y-axis) for which $q_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} q_{A'}(I)$, where τ is on the x-axis. Achieving higher fractions at lower τ-values is considered better. For $\tau = 1$, the y-value indicates the percentage of instances for which an algorithm performs best. The ⏲ tick indicates the fraction of instances for which *all* runs of that algorithm timed out.

Medium-Sized Instances. On set A, we compare Mt-KaHyPar with KaHyPar-HFC [24, 30] (similar components as Mt-KaHyPar-Q-F) which is currently the best sequential partitioner in terms of solution quality [50], the recursive bipartitioning version (hMetis-R) of hMetis 2.0 [34], as well as the default (PaToH-D) and quality preset (PaToH-Q) of PaToH 3.3 [12]. All configurations of Mt-KaHyPar use 10 threads.

[2] Benchmark sets and results are available from `https://algo2.iti.kit.edu/heuer/sea22`

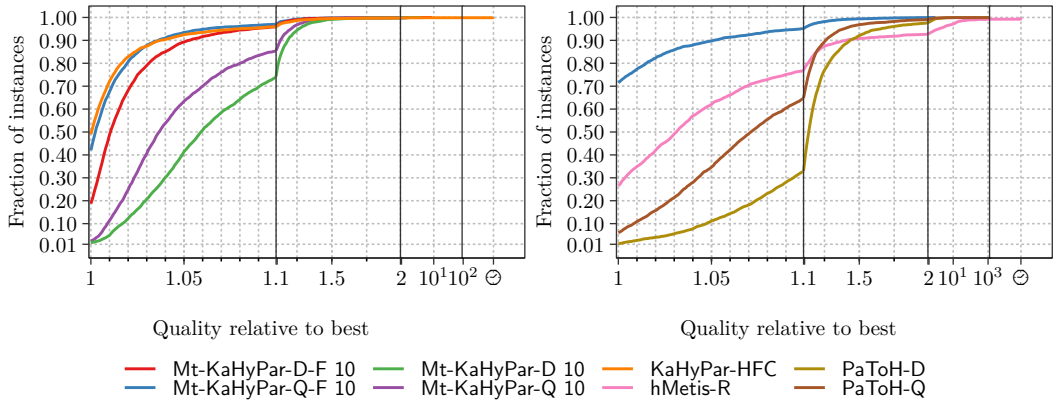


Figure 3 Solution quality of Mt-KaHyPar-Q-F compared with different partitioners on set A.

Figure 3 compares the solution quality of Mt-KaHyPar with different partitioners on set A (see Figure 5 (left) for running times). In an individual comparison, Mt-KaHyPar-Q-F finds better partitions than PaToH-D, PaToH-Q, Mt-KaHyPar-D, Mt-KaHyPar-Q, Mt-KaHyPar-D-F, hMetis-R and KaHyPar-HFC on 94.7%, 87.7%, 97.3%, 95.7%, 73.5%, 74.5% and 51.3% of the instances, respectively.

The median improvement of Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F compared to the configurations that use no flow-based refinement is 4.2% and 2.7% while only incuring a slowdown by a factor of 3.1 (gmean time 2.73s vs 0.89s) and 1.7 (5.08s vs 2.99s). To put this into perspective, the quality preset of PaToH (PaToH-Q) improves the default preset (PaToH-D) by 5.3% in the median and is a factor of 5 slower (5.86s vs 1.17s). The median improvement of hMetis-R compared to PaToH-Q is 2.6% while it is a factor of 15.9 slower (93.21s vs 5.86s). The solutions produced by Mt-KaHyPar-Q-F are 3% better than those of hMetis-R in the median and it has a similar running time as PaToH-Q (5.08s vs 5.86s). If we compare our two partitioners that use flow-based refinement, we can see that Mt-KaHyPar-Q-F gives only minor quality improvements over Mt-KaHyPar-D-F (median improvement is 0.6% whereas without flow-based refinement it is 1.9%). This demonstrates the effectiveness of flow-based refinement. The solution quality of Mt-KaHyPar-Q-F and KaHyPar-HFC are on par, while Mt-KaHyPar-Q-F is an order of magnitude faster with 10 threads (5.08s vs 48.98s). In conclusion, we achieved the solution quality of the currently hiqhest quality sequential partitioner in a fast parallel code.

Large Instances. On set B, we compare Mt-KaHyPar with the parallel algorithms Zoltan 3.83 [18] and BiPart [42], as well as PaToH-D (which is fast enough for set B as opposed to other sequential algorithms). All parallel algorithms use 64 threads.

Figure 4 compares the solution quality of Mt-KaHyPar with different partitioners on set B (see Figure 5 (right) for running times). The quality of the partitions produced by Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F are comparable while Mt-KaHyPar-D-F is a factor of 1.9 faster (gmean time 30.38s vs 58.24s). Therefore, we focus on Mt-KaHyPar-D-F in this evaluation. In an individual comparison, Mt-KaHyPar-D-F finds better partitions than BiPart, Zoltan, PaToH-D, Mt-KaHyPar-D, Mt-KaHyPar-Q and Mt-KaHyPar-Q-F on 97.3%, 96%, 92%, 91%, 85.1% and 47.3% of the instances, respectively.

The median improvement of Mt-KaHyPar-D-F and Mt-KaHyPar-Q-F compared to the configurations that use no flow-based refinement is 5.2% and 3.4% while they are slower by a factor of 6.6 (30.38s vs 4.63s) and 1.9 (58.24s vs 29.99s). Both the improvements and

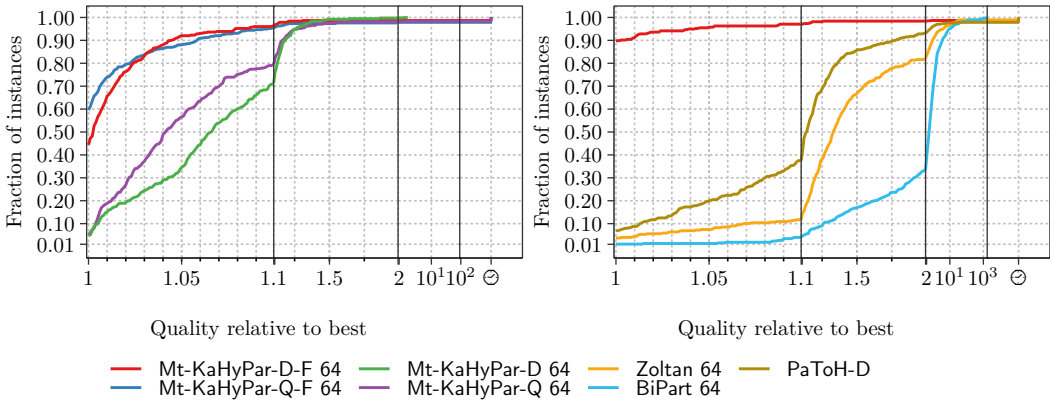


Figure 4 Solution quality of Mt-KaHyPar-D-F compared with different partitioners on set B.

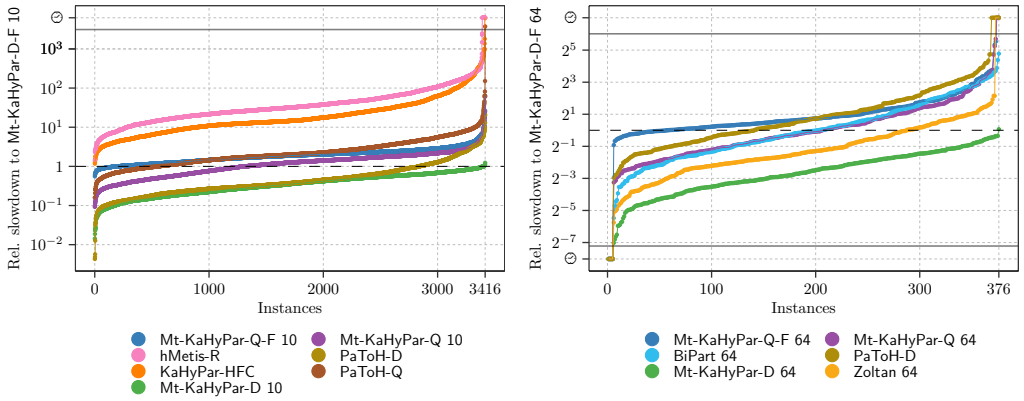


Figure 5 Running times relative to Mt-KaHyPar-D-F on set A (left) and B (right). The ⏲ axis markers represent timeouts for the baseline Mt-KaHyPar-D-F (at the bottom) or the compared algorithm (at the top).

slowdowns are more pronounced here than on set A. The slowdowns are expected since the size of the flow problems scales linearly with instance sizes, while the complexity of the flow-based refinement routine does not. Mt-KaHyPar-D-F (30.38s) is slower than Zoltan (12.6s) and BiPart (29.19s), but faster than PaToH-D (50.3s). However, Mt-KaHyPar-D-F computes partitions that are 33% better than Zoltan's and twice as good as BiPart's in the median.

Scalability. Figure 6 shows self-relative speedups with varying number of threads $t \in \{4, 16, 64\}$. In the plot, we represent the speedup of each instance as a point and the centered rolling geometric mean with a window size of 25 as a line. The x-axis shows the sequential running time of Mt-KaHyPar-D-F for each instance.

We found that any one of the common (hyper)graph metrics (such as number of pins) is not well correlated with speedups (or running times for that matter), since the running time depends on a variety of different factors (metrics and events that trigger repetitions). Fitting suitable parameters for a combination of the metrics seems much more complicated than plotting against sequential running time, which is often nicely correlated with speedups. Furthermore, the longer an algorithm runs sequentially, the more important is an efficient parallelization to achieve reasonable running times.

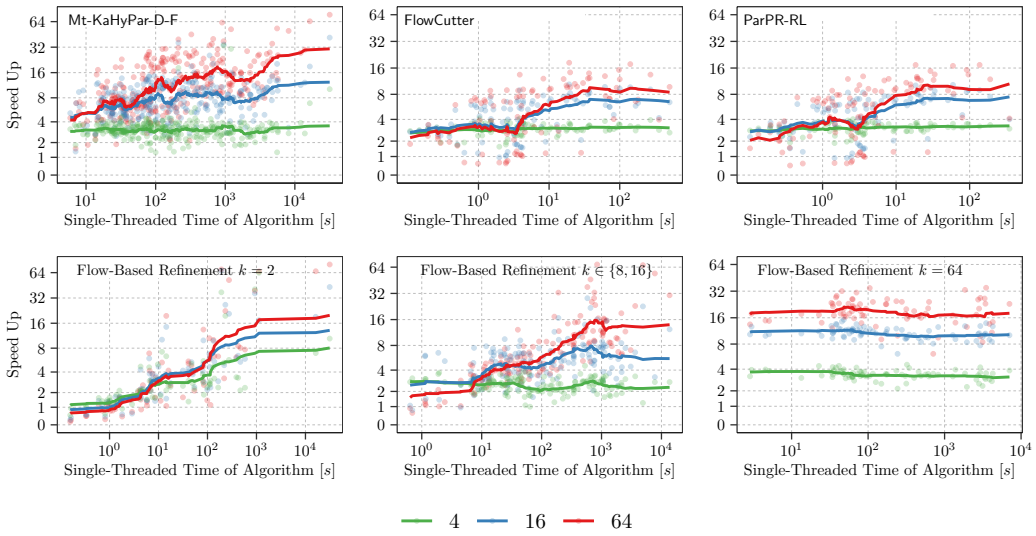


Figure 6 Speedups of Mt-KaHyPar-D-F and the flow-based refinement routine (for different values of k) as well as of the FlowCutter and parallel flow algorithm (ParPR-RL).

To assess FlowCutter and parallel push-relabel (referred to as ParPR-RL), we extract flow networks from bipartitions of the instances in set B[3]. The results are shown in the top-middle and -right plot. With 4 threads, we observe near-perfect speedups throughout, with fairly small variance. For $t = 16, 64$, the parallelization overheads are only outweighed for longer running instances, with more threads becoming worthwhile at about 10 seconds of sequential time. Unfortunately, we even experience some minor slowdowns and the speedups are strongly scattered. The maximum achieved speedups are 10.4, 18.4 for FlowCutter and 10.3, 17.3 for ParPR-RL. These results match what we expected from Ref. [10]. Restricted to instances with sequential running time ≥ 10 seconds, the geometric mean speedups are 6.5 and 8.6 for FlowCutter and 7, 9.9 for ParPR-RL.

To evaluate the speedups of Mt-KaHyPar-D-F, we use a subset of set B (76 out of 94 hypergraphs)[4]. We measure the full partitioning process (top left) and just flow-based refinement including scheduling (bottom row). Note that we use a sequential push-relabel when there are enough flow problems that can be solved independently. The geometric mean speedup of Mt-KaHyPar-D-F is 3.1 for $t = 4$, 7.4 for $t = 16$ and 10.62 for $t = 64$. If we only consider instances with a single-threaded running time ≥ 100s, we achieve a geometric mean speedup of 14.5 for $t = 64$. For $k = 2$, the scalability of the flow-based refinement routine largely depends on FlowCutter as the only parallelism source. We can see that the speedups of the two are comparable (compare Figure 6 top-middle with bottom-left). There are a few outliers (e.g. `nlpkkt200` with a speedup of 80.05 for $t = 64$) where the flow network construction dominates the execution time for $t = 1$. For $k = 64$ and $t = 64$, we achieve a

[3] The instances are available from `https://algo2.iti.kit.edu/heuer/sea22`.

[4] Subset contains all hypergraphs on which Mt-KaHyPar-D-F 64 was able to complete in under 600 seconds for $k \in \{2, 8, 16, 64\}$. We omit scalability experiments with Mt-KaHyPar-Q-F due to the long time requirements and because flow-based refinement is used in the same context in Mt-KaHyPar-D-F. This experiment still took 4 weeks on machine B.

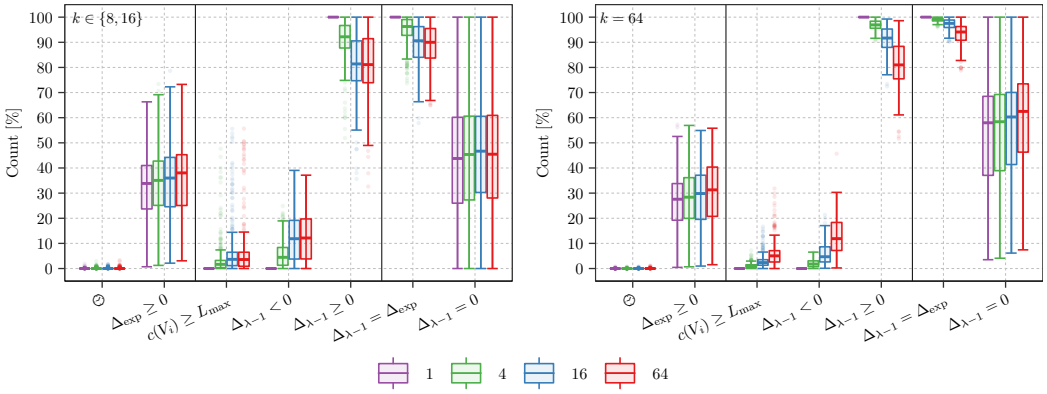


Figure 7 Conflicts for $k \in \{8, 16\}$ (left) and $k = 64$ (right) on set B. For each instance, we count the refinements that exceed the time limit (⏲), the potential improvements ($\Delta_{exp} \geq 0$) and move sequences that violate the balance constraint ($c(V_i) \geq L_{max}$) or degrade ($\Delta_{\lambda-1} < 0$) or improve the connectivity metric ($\Delta_{\lambda-1} \geq 0$). For move sequences with $\Delta_{\lambda-1} \geq 0$, we count if the actual improvements equals the expected ($\Delta_{\lambda-1} = \Delta_{exp}$) and zero-gain improvements ($\Delta_{\lambda-1} = 0$).

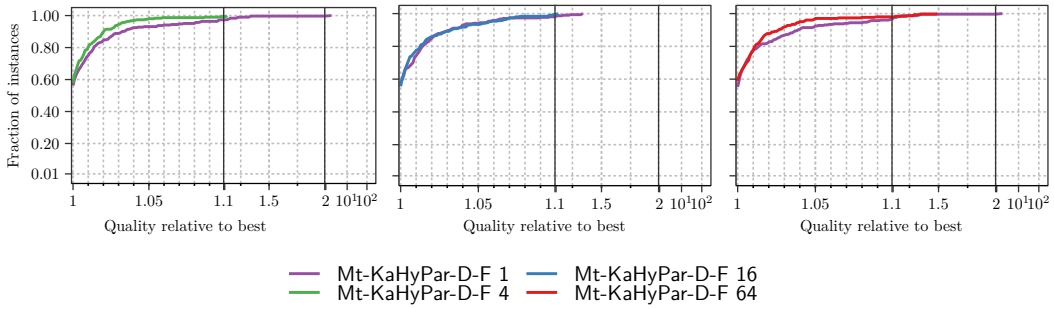


Figure 8 Performance profiles comparing solution quality of Mt-KaHyPar-D-F with increasing number of threads on set B.

geometric mean speedup of 18.48. In this case, all parallelism is leveraged in the scheduler, and none in FlowCutter, which explains why the speedups are more reliable than for other k. For $k \in \{8, 16\}$, both parallelism sources are used and speedups are better than for $k = 2$.

Search Interference. Figure 7 gives an overview on the different types of conflicts in the flow-based refinement routine (as explained in Section 5) and how often they occur. In the median, 33.38% of flow-based refinements find a potential improvement ($\Delta_{exp} \geq 0$), of which we successfully apply 85.62% to the global partition for $t = 64$ (90.48% for $t = 16$ and 96.73% for $t = 4$). For $t = 64$, 2.55% of the move sequences violate the balance constraint (2% for $t = 16$ and 0.77% for $t = 4$) and 8.06% would actually degrade the solution quality before being reverted (4.74% for $t = 16$ and 1.72% for $t = 4$). However, increasing the number of threads does not adversely affect the solution quality of Mt-KaHyPar-D-F (see Figure 8), as repetitions from multiple rounds and on different levels can compensate effectively.

Detailed Running Times of Flow-Based Refinement. Figure 9 shows the running times of the different phases of the flow-based refinement routine relative to its total running time. For $k \leq 16$, FlowCutter dominates the running time. For $k = 64$, the flow network construction and FlowCutter have the same share on the total running time, while applying move sequences and growing the region B are negligible.

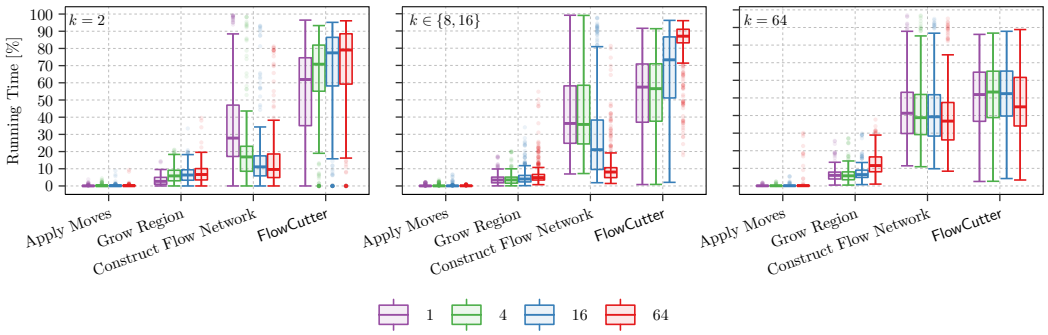


Figure 9 Running times of the different phases of the flow-based refinement routine relative to its total running time for $k = 2$ (left), $k \in \{8, 16\}$ (middle) and $k = 64$ (right) on set B.

10 Conclusion and Future Work

This work marks the end of a series of publications with the aim to transfer techniques used in modern sequential partitioning algorithms into the shared-memory context without comprises in solution quality. The result is a set of parallel algorithms unified in one framework [26, 27] (Mt-KaHyPar) that outperforms all popular hypergraph partitioners. Summarizing our experimental results, we obtain good speedups for larger values of k even on small instances, where scheduling provides lots of parallelism. This is more difficult for small k where the parallelism stems from the flow algorithm, yet we still obtain good speedups that match those in Ref. [10]. In particular on long-running instances, the speedups are on par with those for large k. Using 10 threads, our system is 10 times faster than the sequential state-of-the-art system KaHyPar with flow-based refinement, while achieving the same solution quality.

Future work includes a deterministic version of parallel flow-based refinement, as well as a highly localized version used in an n-level partitioner that only constructs small flow problems around uncontracted nodes.

References

1 Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a Direct k-way Hypergraph Partitioning Algorithm. In *19th Workshop on Algorithm Engineering & Experiments (ALENEX)*, pages 28–42. SIAM, January 2017. doi:10.1137/1.9781611974768.3.

2 Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. High-Quality Shared-Memory Graph Partitioning. In *European Conference on Parallel Processing (Euro-Par)*, pages 659–671. Springer, August 2017. doi:10.1007/978-3-319-96983-1_47.

3 Charles J. Alpert. The ISPD98 Circuit Benchmark Suite. In *International Symposium on Physical Design (ISPD)*, pages 80–85, April 1998. doi:10.1145/274535.274546.

4 Charles J. Alpert and Andrew B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration*, 19(1-2):1–81, 1995. doi:10.1016/0167-9260(95)00008-4.

5 Reid Andersen. and Kevin J. Lang. An Algorithm for Improving Graph Partitions. In *Proc. of the 19th ACM-SIAM Symposium on Discrete Algorithms*, pages 651–660. Society for Industrial and Applied Mathematics, 2008. doi:10.5555/1347082.1347154.

6 Richard J. Anderson and João C. Setubal. A Parallel Implementation of the Push-Relabel Algorithm for the Maximum Flow Problem. *J. Parallel Distributed Comput.*, 29(1):17–26, 1995. doi:10.1006/jpdc.1995.1103.

7 Aaron Archer, Kevin Aydin, Mohammad Hossein Bateni, Vahab S. Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. Cache-Aware Load Balancing of Data Center Applications. *Proceedings of the VLDB Endowment*, 12(6):709–723, 2019. doi:10.14778/3311880.3311887.

8 Cevdet Aykanat, Berkant Barla Cambazoglu, and Bora Uçar. Multi-level Direct k-way Hypergraph Partitioning With Multiple Constraints and Fixed Vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, 2008. doi:10.1016/j.jpdc.2007.09.006.

9 David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph Partitioning and Graph Clustering*, volume 588. American Mathematical Society Providence, RI, 2013. doi:10.1090/conm/588.

10 Niklas Baumstark, Guy E. Blelloch, and Julian Shun. Efficient implementation of a synchronous parallel push-relabel algorithm. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, volume 9294 of *Lecture Notes in Computer Science*, pages 106–117. Springer, 2015. doi:10.1007/978-3-662-48350-3_10.

11 Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. The SAT Competition 2014. http://www.satcompetition.org/2014/, 2014.

12 Ümit V. Catalyurek and Cevdet Aykanat. Hypergraph-Partitioning-based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999. doi:10.1109/71.780863.

13 Boris V. Cherkassky and Andrew V. Goldberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19(4):390–410, 1997. doi:10.1007/PL00009180.

14 Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment*, 3(1):48–57, 2010. doi:10.14778/1920841.1920853.

15 Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, November 2011. doi:10.1145/2049662.2049663.

16 Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. In *Proc. of the 25th International Parallel and Distributed Processing Symposium*, pages 1135–1146, 2011. doi:10.1109/IPDPS.2011.108.

17 Mehmet Deveci, Kamer Kaya, and Ümit V. Çatalyürek. Hypergraph Sparsification and Its Application to Partitioning. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 200–209, 2013. doi:10.1109/ICPP.2013.29.

18 Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Catalyurek. Parallel Hypergraph Partitioning for Scientific Computing. In *IEEE Transactions on Parallel and Distributed Systems*, pages 10–pp. IEEE, 2006. doi:10.1109/IPDPS.2006.1639359.

19 Elizabeth D. Dolan and Jorge J. Moré. Benchmarking Optimization Software with Performance Profiles. *Mathematical Programming*, 91(2):201–213, 2002. doi:10.1007/s101070100263.

20 Charles M. Fiduccia and Robert M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Conference on Design Automation (DAC)*, pages 175–181, 1982. doi:10.1145/800263.809204.

21 Lester Randolph Ford and Delbert R Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956. doi:10.4153/CJM-1956-045-5.

22 Andrew V. Goldberg and Robert Endre Tarjan. A New Approach to the Maximum-Flow Problem. *Journal of the ACM*, 35(4):921–940, 1988. doi:10.1145/48014.61051.

23 Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. Advanced Flow-Based Multilevel Hypergraph Partitioning. *18th International Symposium on Experimental Algorithms (SEA)*, 2020. doi:10.4230/LIPIcs.SEA.2020.11.

24 Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm. In *27th European Symposium on Algorithms (ESA)*, pages 52:1–52:17, 2019. doi:10.4230/LIPIcs.ESA.2019.52.

25 Lars Gottesbüren, Tobias Heuer, and Peter Sanders. Parallel Flow-Based Hypergraph Partitioning. Technical report, Karlsruhe Institute of Technology, 2022.

26 Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable Shared-Memory Hypergraph Partitioning. In *23st Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, January 2021. doi:10.1137/1.9781611976472.2.

27 Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Shared-Memory n-level Hypergraph Partitioning. In *24th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, January 2022. doi:10.1137/1.9781611977042.11.

28 Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, 2021. doi:10.22331/q-2021-03-15-410.

29 Michael Hamann and Ben Strasser. Graph Bisection with Pareto Optimization. *ACM Journal of Experimental Algorithmics*, 23, 2018. doi:10.1145/3173045.

30 Tobias Heuer, Peter Sanders, and Sebastian Schlag. Network Flow-Based Refinement for Multilevel Hypergraph Partitioning. *ACM Journal of Experimental Algorithmics (JEA)*, 24(1):2.3:1–2.3:36, September 2019. doi:10.1145/3329872.

31 Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In *16th International Symposium on Experimental Algorithms (SEA)*, pages 21:1–21:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, June 2017. doi:10.4230/LIPIcs.SEA.2017.21.

32 Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner. In *Proceedings of the VLDB Endowment*, volume 10, pages 1418–1429, 2017. doi:10.14778/3137628.3137650.

33 Gökçehan Kara and Can C. Özturan. Graph Coloring Based Parallel Push-relabel Algorithm for the Maximum Flow Problem. *ACM Transactions on Mathematical Software*, 45(4):46:1–46:28, 2019. doi:10.1145/3330481.

34 George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999. doi:10.1109/92.748202.

35 George Karypis and Vipin Kumar. Multilevel k-way Hypergraph Partitioning. *VLSI Design*, 2000(3):285–300, 2000. doi:10.1155/2000/19436.

36 Alexander V. Karzanov. Determining the Maximal Flow in a Network by the Method of Preflows. In *Soviet Mathematics Doklady*, volume 15, pages 434–437, 1974.

37 Brian W. Kernighan and Shen Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(2):291–307, February 1970.

38 Balakrishnan Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Trans. Computers*, 33(5):438–446, 1984. doi:10.1109/TC.1984.1676460.

39 Kevin J. Lang and Satish Rao. A Flow-Based Method for Improving the Expansion or Conductance of Graph Cuts. In *Proc. of 10th International Integer Programming and Combinatorial Optimization Conference*, volume 3064 of *LNCS*, pages 383–400. Springer, 2004. doi:10.1007/978-3-540-25960-2_25.

40 Dominique LaSalle and George Karypis. Multi-Threaded Graph Partitioning. In *IEEE Transactions on Parallel and Distributed Systems*, pages 225–236. IEEE, 2013. doi:10.1109/IPDPS.2013.50.

41 Eugene L. Lawler. Cutsets and Partitions of Hypergraphs. *Networks*, 3(3):275–285, 1973. doi:10.1002/net.3230030306.

42 Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. BiPart: A Parallel and Deterministic Hypergraph Partitioner. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–174, 2021. doi:10.1145/3437801.3441611.

43 Zoltán Á. Mann and Pál A. Papp. Formula Partitioning Revisited. In *5th Pragmatics of SAT Workshop*, pages 41–56, 2014. doi:10.29007/9skn.

44 Vitaly Osipov and Peter Sanders. n-Level Graph Partitioning. In *18th European Symposium on Algorithms (ESA)*, pages 278–289. Springer, 2010. doi:10.1007/978-3-642-15775-2_24.

45 David A. Papa and Igor L. Markov. Hypergraph Partitioning and Clustering. In *Handbook of Approximation Algorithms and Metaheuristics*. Citeseer, 2007. doi:10.1201/9781420010749.ch61.

46 Chuck Pheatt. Intel Threading Building Blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

47 Jean-Claude Picard and Maurice Queyranne. On the Structure of All Minimum Cuts in a Network and Applications. *Math. Program.*, 22(1):121, 1982. doi:10.1007/BF01581031.

48 Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E*, 76(3):036106, 2007. doi:10.1103/PhysRevE.76.036106.

49 Peter Sanders and Christian Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *19th European Symposium on Algorithms (ESA)*, pages 469–480. Springer, 2011. doi:10.1007/978-3-642-23719-5_40.

50 Sebastian Schlag. *High-Quality Hypergraph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, 2020. doi:10.5445/IR/1000105953.

51 Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way Hypergraph Partitioning via n-Level Recursive Bisection. In *18th Workshop on Algorithm Engineering & Experiments (ALENEX)*, pages 53–67. SIAM, 2016. doi:10.1137/1.9781611974317.5.

52 Yossi Shiloach and Uzi Vishkin. An O(n^2 log n) Parallel Max-Flow Algorithm. *Journal of Algorithms*, 3(2):128–146, 1982. doi:10.1016/0196-6774(82)90013-X.

53 Aleksandar Trifunovic and William J. Knottenbelt. Parkway 2.0: A Parallel Multilevel Hypergraph Partitioning Tool. In *International Symposium on Computer and Information Sciences*, pages 789–800. Springer, 2004. doi:10.1007/978-3-540-30182-0_79.

54 Natarajan Viswanathan, Charles J. Alpert, Cliff C. N. Sze, Zhuo Li, and Yaoguang Wei. The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite. In *49th Conference on Design Automation (DAC)*, pages 774–782. ACM, June 2012. doi:10.1145/2228360.2228500.

55 Hannah Honghua Yang and D.F. Wong. Efficient Network Flow Based Min-Cut Balanced Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1533–1540, 1996. doi:10.1007/978-1-4615-0292-0_41.

A Quotient Graph Maintenance

For each block pair, we explicitly store the hyperedges connecting the two. This information is required by the flow network construction algorithm to construct the region B. Block pairs that contain at least one hyperedge form the edges of the quotient graph. We construct this data structure by iterating over all hyperedges in parallel and add a hyperedge $e \in E$ to the block pairs contained in $\{\{V_i, V_j\} \subseteq \Lambda(e) \mid i < j\}$.

If we apply a move sequence on the partition, we add all hyperedges $e \in E$ where $\Phi(e, V_j)$ increases to one to all block pairs contained in $\{\{V_j, V_k\} \mid V_k \in \Lambda(e) \setminus \{V_j\}\}$. If $\Phi(e, V_i)$ decreases to zero, we remove e lazily from corresponding block pairs during the flow network construction.

B Network Construction Algorithm

We implemented two construction algorithms that are preferable in different situations. Both construct the hypergraph $\mathcal{H}$ as explained at the beginning of Section 6. In the following, we will denote with $E_B := \{e \in E \mid e \cap B \neq \emptyset\}$ the set of hyperedges that contain nodes of the region B.

The first algorithm iterates over all nets $e \in E_B$. If a pin $p \in e$ is contained in B, we add p to hyperedge e in $\mathcal{H}$. Otherwise, we add the source s or sink t to e, if $p \in V_1$ or $p \in V_2$. The second algorithm iterates over all nodes $u \in B$ and for each net $e \in I(u)$, we insert a pair (e, u) into a vector. Sorting the vector (lexicographically) yields the pin lists of the subhypergraph H_B. Afterwards, we insert each net e in the pin list vector into $\mathcal{H}$ and add the source s or sink t to e, if $\Phi(e, B_1) < \Phi(e, V_1)$ or $\Phi(e, B_2) < \Phi(e, V_2)$.

The first algorithm has linear running time, but has to scan all hyperedges of H in their entirety in the worst case even if most of their pins are not contained in $\mathcal{H}$. The complexity of the second algorithm only depends on the number of pins in $\mathcal{H}$, but requires to sort the pin lists in a temporary vector. We use the second algorithm for hypergraphs with a low density $p := |E|/|V|$ (≤ 0.5) or a large average hyperedge size $\overline{|e|}$ (≥ 100).

Note that both algorithms discard single-pin nets and nets that contain both the source and sink (such nets cannot be removed from the cut).

Parallelization. The first algorithm iterates over all nets $e \in E_B$ in parallel and each thread uses the sequential algorithm to construct a thread-local pin list vector. Afterwards, we use a prefix sum operation to copy the pin lists of each thread to $\mathcal{H}$.

The second algorithm iterates over all nodes $u \in B$ in parallel and then uses hashing to distribute the pairs (e, u) to buckets. Afterwards, we process each bucket in parallel and apply the sequential algorithm to construct the pin list vector of each bucket. Finally, we use a prefix sum operation to copy the pin lists of each bucket to $\mathcal{H}$.

Identical Net Removal. Since some nets of H are only partially contained in $\mathcal{H}$, some of them may become identical. Therefore, we further reduce the size of $\mathcal{H}$ by removing all identical nets except for one representative at which we aggregate their weight. We use the identical net detection algorithm of Aykanat et al. [8, 17]. It uses *fingerprints* $f_e := \sum_{v \in e} v^2$ to eliminate unnecessary pairwise comparisons between nets. Nets with different fingerprints or different sizes cannot be identical. If we insert a net e into $\mathcal{H}$, we store the pair (f_e, e) in a hash table with chaining to resolve collisions (uses concurrent vectors to handle parallel access). We can then use the hash table to perform pin-list comparisons between the nets with the same fingerprint for subsequent net insertions. Note that in the parallel scenario we may not be able to detect all identical nets due to simultaneous insertions into the hash table. However, this does not affect correctness of the refinement, as removing identical nets is only a performance optimization.

C Benchmark Sets

All instances of the benchmark sets used in the experimental evaluation are derived from four sources encompassing three application domains: the ISPD98 VLSI Circuit Benchmark Suite [3], the DAC 2012 Routability-Driven Placement Contest [54], the SuiteSparse Matrix Collection [15], and the 2014 SAT Competition [11]. VLSI instances are transformed into hypergraphs by converting the netlist of each circuit into a set of hyperedges. Sparse matrices

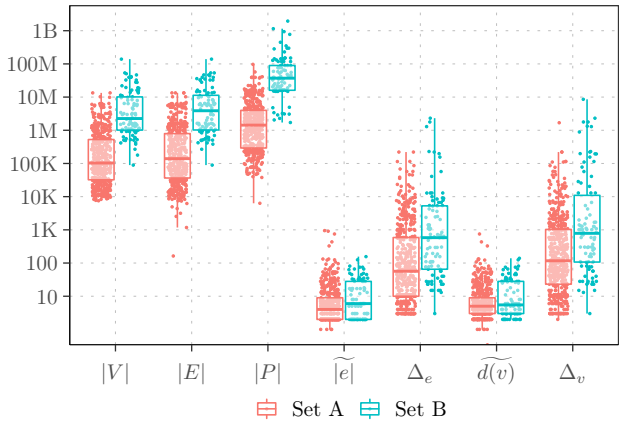


Figure 10 Summary of different properties for our two benchmark sets. It shows for each hypergraph (points), the number of vertices $|V|$, nets $|E|$ and pins $|P|$, as well as the median and maximum net size ($\widetilde{|e|}$ and Δ_e) and vertex degree ($\widetilde{d(v)}$ and Δ_v).

are translated to hypergraphs using the row-net model [12] and SAT instances to three different hypergraph representations: *literal*, *primal*, and *dual* [43, 45] (see Ref. [31] for more details). All hypergraphs have unit vertex and net weights. Figure 10 shows that the hypergraphs of set B are more than an order of magnitude larger than those of set A.

Routing in Multimodal Transportation Networks with Non-Scheduled Lines

Darko Drakulic
NAVER LABS Europe, Meylan, France

Christelle Loiodice
NAVER LABS Europe, Meylan, France

Vassilissa Lehoux[1]
NAVER LABS Europe, Meylan, France

Abstract

Over the last decades, new mobility offers have emerged to enlarge the coverage and the accessibility of public transportation systems. In many areas, public transit now incorporates on-demand transport lines, that can be activated at user need. In this paper, we propose to integrate lines without predefined schedules but with predefined stop sequences into a state-of-the-art trip planning algorithm for public transit, the Trip-Based Public Transit Routing algorithm [30]. We extend this algorithm to non-scheduled lines and explain how to model other modes of transportation, such as bike sharing, with this approach. The resulting algorithm is exact and optimizes two criteria: the earliest arrival time and the minimal number of transfers. Experiments on two large datasets show the interest of the proposed method over a baseline modelling.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Multimodal routing, on-demand public transportation, bicriteria shortest paths

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.6

1 Introduction

Based on modern public transit routing algorithms, hundreds of trip planning applications are used by millions of users every day. They integrate public transit information with road networks and usually compute itineraries combining only public transportation services with walking transfers, which is often referred to as *public transit routing*. In order to offer a more integrated experience to their users, some applications allow for more multimodality, combining public transportation with other available transportation offers, such as taxis, bike sharing or car sharing. We then speak of *multimodal* or *intermodal routing*.

In addition to the classical scheduled public transportation, many transport authorities propose special transportation offers in sub-urban areas, or for elderly or disabled people. They are usually organized as on-demand services, where transportation authorities define lines (sequence of stops) or areas of coverage, but no fixed schedules. For this type of service, users must "activate" the desired trip by contacting the transport agency. A lot of transport authorities in France offer this type of services, for example in Montauban metropolitan area [1] (non-scheduled lines with only a subset of stops activated), in Flers metropolitan area [24] (on-demand transportation between predefined stations during given time intervals), or in Pays de Dreux [16] (on-demand transportation for elderly people from home place to

[1] Corresponding author

any destination within a zone). Note that some transport authorities also provide on-demand services with predefined schedules, but in that case, a public transit routing algorithm can integrate them as classical scheduled lines in itinerary computations.

In this paper, we propose an extension of the Trip-Based Public Transit Routing (TB) algorithm [30] which is a state-of-the-art algorithm for public transit routing. We modify it to deal with non-scheduled lines, that we define as lines that can be activated on given periods and for which a sequence of stops is defined, as well as possibly time-dependent transport duration between stops of the line, but no exact schedules. This model covers also the simpler cases where the line has exactly two stops. Bike sharing for instance could be modeled using non-scheduled lines, for example by creating one line per pair of stations that are reachable from each other. In that case, the bike section is considered as a trip from the algorithm point of view, and using bike sharing increases the number of transfers between trips of the itinerary. Similarly, taxi-like offers that cover some predefined sets of origins and destinations can be modeled as non-scheduled lines. In both cases, we then consider that using those modes is equivalent to taking one additional trip in terms of inconvenience (which is modeled by the number of transfers of an itinerary).

Many public transit routing and multimodal trip planning algorithms have been proposed recently in the literature [5], but to the best of our knowledge, general non-scheduled lines have not been considered explicitly.

In Section 2, we discuss recent algorithms for public transit and multimodal routing and models for the simpler cases that appear in the literature. In Section 3 we introduce the notation and briefly present the Trip-Based Public Transit Routing algorithm [30] that we extend in Section 4 for supporting non-scheduled lines. In Section 5, we present the results of experiments on two real world datasets (Île-De-France and Netherlands), we summarize our work and give directions for future research in Section 6.

2 Related work

In this article, we are mainly interested in two classical criteria to minimize in multimodal routing, which are the number of transfers and the arrival time. The number of transfers represents the inconvenience for the user to change vehicles and is an important criterion for evaluating itineraries. Given a start time, computing the Pareto set for those two criteria is intractable as the size of the Pareto set can be exponential [20]. However, the Pareto front is of polynomial size (bounded by the number of trips) and can be computed in polynomial time, along with one solution per value in the Pareto front (note that this is often referred to computing the Pareto set in the literature, while only a subset of the Pareto set is indeed obtained). We use here the notation of [26] and denote by *complete set* such a solution set. Most recent algorithms, considering either minimum arrival time alone or bicriteria queries, can compute earliest arrival time or Pareto front in time ranging from tens of microseconds to a few hundreds of milliseconds for large public transit networks. Transfer Patterns [4], RAPTOR [13, 10], Connection Scan (CSA) [14], Public Transit Labeling [12] or Trip-Based Public Transit Routing [30] have been specifically designed for those networks as using directly classical methods for road networks does not seem to perform well with the time-dependent schedules [3].

Although, to the best of our knowledge, combination of scheduled and non-scheduled lines in public transit networks has not been studied before, some algorithms can handle more transportation modes in combination to public transit, including bike or car sharing.

In the graph-based approaches, the different networks corresponding to each mode are combined into a single time-dependent or time-expanded graph. The timetable information and transfer constraints (for instance minimum change times at a station) are modeled into the graph structure, often increasing significantly the graph size [25]. In that type of approach, non-scheduled lines can be modeled into the graph as additional arcs and nodes available for some time intervals. In order to take into account the different modes with graph-based modeling, one possible solution is to define an automaton that restricts the possible mode sequences and solve a *label-constrained shortest path problem*. The label-constrained shortest path problem is tractable for regular languages [2] and some authors proposed algorithms allowing for mode sequences using or not public transit. Kirchler et al. [21] propose to adapt the ALT algorithm [18] to take into account predefined mode sequences, resulting in the SDALT algorithm (State-Dependent ALT). They consider a network that includes bike and car sharing. Dibbelt et al. [15] modify Contraction Hierarchy [17] to integrate user defined sequences provided at query time. The resulting algorithm is called User Constrained Contraction Hierarchy. They apply it on networks combining cars and public transportation, but the approach could be applied for bike or car sharing. One of the drawbacks of this algorithm is the preprocessing time (42 minutes on a network with 30.5K stops and 1.6M connections) that doesn't allow for real-time modification of the schedules.

The second main type of approaches consists in using timetable directly without modeling it into a graph. The RAPTOR algorithm [13] is one of these algorithms, using dynamic programming to perform a breadth-first search that labels the stops reached, one additional trip being taken at each iteration of the algorithm. It has been modified in [11, 28] to allow for some more complex mode sequences, such as combining public transit with bike or car sharing, by modeling it similarly as a walking transfer or with a biking part equivalent to a trip. Shortest travel times between stops using bike or car sharing are then precomputed and integrated as alternative ways to change from one line to another. A recent approach [27] combines RAPTOR with ULTRA [7] to reduce the running time and to consider several bike sharing operators simultaneously.

In this article, we are interested in the general case, where the sequence of non-scheduled lines contains more than two elements. However, as bike and car sharing are special cases of non-scheduled lines with two-stop sequences, our algorithm could be used to interleave them with public transportation, even if it is not the main objective here. The proposed method integrates non-scheduled lines in the Trip-Based Public Transit Routing algorithm that we present in Section 3.

3 Preliminaries

We introduce in this section the notation used in the paper, and explain the principle of the Trip-Based Public Transit Routing (TB) algorithm [30].

3.1 Notation

Public transit networks are defined by their stops and trip schedules. A *stop* p is a physical location where passengers can board or alight a public transportation vehicle (e.g. a bus, a tram, a metro). A *trip* t is represented by its *schedule*: a sequence of stops $\overrightarrow{p}(t) = (p_t^1, p_t^2, \dots)$ where the vehicle stops, with arrival time $\tau_{\text{arr}}(t,i)$ and departure time $\tau_{\text{dep}}(t,i)$ at its i^{th} stop p_t^i. A partial order is defined over trips with the same stop sequence $(p^1, p^2, \dots, p^n)$ by the relations $\leq$ and $<$:

$$t \leq t' \Leftrightarrow \forall i \in \{1,2,\dots,n\}, \tau_{\text{arr}}(t,i) \leq \tau_{\text{arr}}(t',i)$$

$$t < t' \Leftrightarrow (t \leq t' \text{ and } \exists i \in \{1,2,\dots,n\}, \tau_{\text{arr}}(t,i) < \tau_{\text{arr}}(t',i))$$

A scheduled line l is then a totally ordered set of trips with the same stop sequence $\overrightarrow{p}(l)$. Note that if there are two trips with the same stop sequence such that one is overtaking the other, they are associated to different lines. L is the set of all scheduled lines, p_l^i represents the i^{th} stop of line l and l_t denotes the line of trip t. For each stop p, we define the set of the lines passing by p with their corresponding stop index at p by $L(p) = \{(l, i) \mid l \in L,\, i \in \{1, 2, \dots | \overrightarrow{p}(l)|\},\, p = p_l^i\}$.

When arriving at stop p_l^i at time τ, it is possible to board a trip t of line l if $\tau \leq \tau_{\text{dep}}(t, i)$. When it exists, we can hence define the earliest trip of line l departing from its i^{th} stop after time τ, that we denote by $\text{earliest}(l, i, \tau)$.

The segment of the trip t between stops of index i and j is denoted by $p_t^i \to p_t^j$ and similarly, a transfer between the i^{th} stop of trip t and the j^{th} stop of trip t' is denoted by $p_t^i \to p_{t'}^j$.

Minimum walking transfer duration (footpath) between stops p and q is denoted by $\Delta\tau_{\text{fp}}(p, q)$ and minimum changing vehicle duration at stop p by $\Delta\tau_{\text{ch}}(p)$ (for example, the duration for changing platforms at the same stop). Transfer $p_t^i \to p_{t'}^j$ is *feasible* if and only if:

$$\begin{aligned} & \tau_{\text{arr}}(t, i) + \Delta\tau_{\text{fp}}(p_t^i, p_{t'}^j) \leq \tau_{\text{dep}}(t', j), \quad \text{if } p_t^i \neq p_{t'}^j \\ \text{or} \quad & \tau_{\text{arr}}(t, i) + \Delta\tau_{\text{ch}}(p_t^i) \leq \tau_{\text{dep}}(t', j), \quad \text{if } p_t^i = p_{t'}^j \end{aligned}$$

Lines without a schedule. Now, we extend the above defined notation to lines without a schedule, whose set is denoted $\widehat{L}$. A non-scheduled line l has a sequence $\overrightarrow{p}(l) = (p_l^1, p_l^2, \dots)$ of stops, but no fixed timetable, as they should be activated at the user's demand. Trips for those lines can be instantiated during given time intervals when the service is available and we can define as before the set of all lines without schedule passing by p. We denote it by $\widehat{L}(p)$.

For easy computation of trip $\text{earliest}(l, i, \tau)$, we defined one union of availability intervals for each stop of the non-scheduled line l, and we denote it by $I(l, i)$ for the i^{th} stop of l. A possible way to define those time intervals is to define them for the first stop and then translate them to the other stops of the line by adding traveling duration between stops. This could be the case for on-demand buses if the bus passes by all the stops when activated. Another possibility is to use the same time interval for all stops. It can be the case for non-scheduled lines defined for bike sharing stations or for taxi-like transportation between two points where the time-intervals represent the service availability period.

An easy way of including non-scheduled lines in existing trip planning algorithms is to discretize the intervals of $I(l, 1)$ and generate all possible trips (e.g. creating one trip every minute). In a context of urban mobility, the intervals can be wide (typically from 7.00 am to 6.00 pm) so this approach can significantly increase the number of trips and the number of possible transfers. For the TB algorithm, it has a significant impact on preprocessing and query times. This approach is used in our experiments as a baseline method.

In some cases, a boarding or alighting duration might be considered for the lines of $\widehat{L}$. For instance, it can model the time needed to buy a bus ticket or to get off with some luggage. For bike sharing rides, the boarding time could be the duration needed to get the bicycle from the station and the alighting time the duration to put it back in place. We denote by $\tau_{\text{bo}}(l)$ the duration necessary for boarding the line and $\tau_{\text{al}}(l)$ the duration necessary for alighting. To remain general, we consider boarding and alighting times for all lines, as we can just set them to 0 when they are not relevant.

3.2 Trip-Based Public Transit Routing

Trip-Based Public Transit Routing [30] is an algorithm for computing a complete solution set for minimum arrival time and number of transfers in public transit networks, considering an origin, a destination and a start time. The author claims to consider maximum departure time as a secondary criterion used to break ties, but it is proven in [22] that there is no guarantee regarding this last criterion.

The TB algorithm is based on the preprocessing of a set of the possible transfers between trips. The aim is to build for each trip, during a preprocessing phase, a neighborhood of reachable trips in such way that for each value in the Pareto front, there exists an optimal path with this value using only elements of the resulting neighborhoods. A bicriteria earliest arrival time query then consists in a breadth-first search like exploration in a time-independent graph where trips are vertices and transfers are arcs. Figure 1 gives an outline of the algorithm.

The method proposed in [30] also covers profile queries, where all the optimal values must be found for a given starting time range, hence effectively optimizing latest departure time as a third criterion.

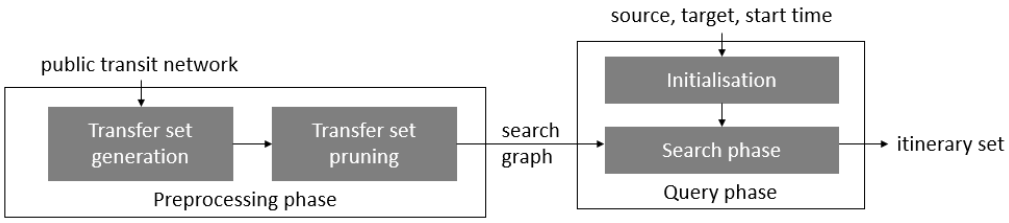


Figure 1 Phases of the TB algorithm.

Preprocessing. The transfer set size impacts the exploration time. As many transfers cannot appear in any optimal solution, it is advisable to prune the transfer set. For instance, if you consider the possible transfers between one trip and a different line, only the earliest trip that can be boarded is relevant for the above defined queries. The author hence suggests two pruning methods to reduce the set of possible transfers.

The first removes U-turn transfers for each trip, i.e. transfers that take you back to the previous stop in the trip (later than if you changed trip at the previous stop). The second aims at pruning the set of feasible transfers for each trip based on earliest arrival times at stops. Each transfer is considered, starting with the later ones. If taking later transfers (or remaining on the current trip) leads to identical or better arrival times or if all the trips reachable via the transfer can be reached via those later transfers, then the current transfer is removed from the set, as it cannot lead to other optimal values than the transfers already kept. Note that the transfer set obtained is not minimal in terms of number of transfers, and that it depends on the order of the transfers checked.

Earliest arrival time queries. In the context of the TB algorithm, earliest arrival time queries refer to bicriteria queries where a single departure time τ is provided as input, along with a source stop, denoted by p_{src}, and a target stop p_{tgt}. Minimum arrival time and the minimum number of transfers are optimized. Note that even if this case is not considered in [30], it is not necessary for the origin and destination of the queries to be stops. If they are placed anywhere on the road network, the algorithm is hardly modified but the footpaths to reach the closest stops in the network must be computed, for instance by classical shortest paths in the walking road network.

Earliest arrival time queries start with an initialization phase where the queue of the search phase is initialized and its target set is computed from the source and target stops. The target set is the set of lines $\mathcal{L}$ from which the destination can be reached:

$$\begin{aligned}\mathcal{L} \quad = \quad & \{(l,\, i,\, 0) \mid (l,\, i) \in L(p_{\text{tgt}})\} \quad \cup \\ & \{(l,\, i,\, \Delta\tau_{\text{fp}}(q,\, p_{\text{tgt}})) \mid (l,\, i) \in L(q) \text{ and } q \text{ is a neighbor of } p_{\text{tgt}}\}\end{aligned}$$

In the query phase, the author labels the trips by the index $R(t)$ of the first reached stop of t, initialized to ∞ for all trips. He defines for each number of transfers one queue Q_n of trip segments reached after n transfers. Q_0 is initialized from the lines that can be taken from the source stop. For any stop q reached by walking from p_{src}, the earliest trip of $(l, i) \in L(q)$ is added to the queue, starting at index i. It is the smallest trip t of line l such that

$$\tau_{\text{dep}}(t,\, i) \geq \begin{cases} \tau & \text{if } q = p_{\text{src}} \\ \tau + \Delta\tau_{\text{fp}}(p_{\text{src}},\, q) & \text{otherwise} \end{cases}$$

After initialization, a breadth-first like search is performed. At each iteration, the algorithm scans in turn the trip segments of the queue. If the current one belongs to a target line, the arrival time at destination is compared to that of the solution set. Then, the trip segments reached by transferring from the current trip segment are added to the queue of the next iteration if they improve the trips' labels. It is the case for a trip segment (t, i, k) if t is earlier than any trip of t_l taken so far at stop i. When a trip t is marked with $R(t) = i$, all the later trips of l_t are marked with the minimum of i and their current index.

Profile queries. In *profile queries*, the user provides an earliest departure time τ_{edt} and a latest departure time τ_{ldt}, i.e. an interval in which to depart. The result of the query is a complete set of solutions for minimum arrival time, minimum number of transfers and maximum departure time starting within the interval. The computation for profile queries is as follows: perform an earliest arrival time query starting at τ_{ldt} and add the solutions to the result set of the profile search. Then restart the search starting at the preceding instant without resetting the trip labels. By iterating the process, you obtain a complete set of solutions without performing unnecessary computations as the labels only let you improve on preceding arrival times.

4 Trip-Based algorithm with non-scheduled lines

As we mentioned above, to the best of our knowledge, lines without a schedule are not covered in the literature, although the special case of bike sharing appears in several articles (e.g. [11, 21, 28]). We explain here our method for the general case where the non-scheduled lines can have more than two stops in their sequence.

4.1 Defining a trip for a non-scheduled line

We consider that non-scheduled lines have predefined stop sequences and availability intervals for each stop of the line. In order to define the earliest trip segment of a trip t of a non-scheduled line l starting after a time τ, we need to evaluate the duration of this trip segment. One possibility is to use the same principle as in the General Transit Feed Specification (GTFS) format [19] for frequency-based trips: one trip with a complete schedule is defined and the others are translations of it with different start times. A more complex solution could consider time-dependent travel times between the line's consecutive stops and time-dependent

arrival and departure times of the trips at its stops. In that case, one must keep in mind that trips of a line cannot overtake one another and that those time-dependent travel times need to respect the FIFO property.

The algorithm proposed here is independent of the solution chosen as long as we can define a schedule for earliest(l, i, τ). Note that the trip segment's departure time is either time τ if $\tau \in I(l, i)$ or the earliest instant of $I(l, i)$ after τ. If such time doesn't exist in the current day, then we can consider taking the line the day after at $\min_{\theta \in I(l,i)} \theta$, if the service is available. If the intervals of $I(l, i)$ are sorted in increasing start time order, τ can be computed in logarithmic time of the number of intervals using binary search.

4.2 Transfers to and from a line without schedule

In the TB algorithm, the transfer generation phase starts by computing all the possible transfers for each trip. For each stop p_t^i of the current trip t, find all the stops q that can be reached by footpaths (i.e. $\Delta\tau_{\text{fp}}(p_t^i, q)$ is defined), and check if a transfer can take place for each element (l, j) of $L(q)$. Stop q is reached at time $\theta = \tau_{\text{arr}}(t, i) + \Delta\tau_{\text{fp}}(p_t^i, q)$ (or $\theta = \tau_{\text{arr}}(t, i) + \Delta\tau_{\text{ch}}(q)$ if $q = p_t^i$) and we can enforce a minimum boarding time to get the minimum time $\tau = \theta + \Delta\tau_{\text{bo}}(l)$ at which a trip of line l can be taken. If it is defined, only transfer to the earliest trip of each line passing after time τ is added to the neighborhood of t. We can proceed identically for admissible transfers from trips of scheduled lines to non-scheduled lines. The earliest trip passing at q after τ is defined as in Section 4.1 and we keep only the transfer to that trip.

Initially, the trips of non-scheduled lines are not instantiated: they are implicit within the non-scheduled line definition. It is however possible to precompute some trip segments and transfers to make the search faster. We extend the set of transfers to add the transfers from a trip of a scheduled line to non-scheduled lines. We then prune the resulting extended set of transfers as before. We denote with $\widehat{T}$ the set of transfers from a trip segment of a scheduled line to a trip segment of a non-scheduled line. $T \cup \widehat{T}$ is the extended set of transfers.

Note that for non-scheduled lines, we do not perform a preprocessing of the transfers to scheduled and non-scheduled lines: instead, the transfers from trip segments of non-scheduled lines are computed online during the query phase. It avoids explicitly creating all the non-scheduled trips.

4.3 Modifications in the query phase

The algorithm for the query phase and its initialization can be found in Algorithm 1. The auxiliary procedures of both are described in Algorithm 2.

In the initialization, the lines without schedule are scanned similarly to regular lines for determining the algorithm's targets. To build the initial queue, we consider the availability intervals of non-scheduled lines at the stops reached from the origin and the minimum boarding times to propose the earliest trip segment for reached lines.

A major difference with the initial version of the algorithm is the change in determining the next trip segments to add to the queue. For transfers from scheduled lines, the set T of transfers contains all the preprocessed transfers. For transfers from non-scheduled lines, the transfers are computed on the fly. For transfers to scheduled lines, the next trip to take is computed as in the initial algorithm and the trip segments added to the queue by the procedure ENQUEUE_TRIP. For transfers to non-scheduled lines, the more complicated process of ENQUEUE_LINE and UPDATE_R is required to avoid unnecessary computations.

Algorithm 1 Earliest arrival time query.

input Timetable data, transfer set $T \cup \widehat{T}$
input Source stop p_{src}, destination stop p_{tgt}, start time τ
output Result set J
$J \leftarrow \emptyset$, $\mathcal{L} \leftarrow \emptyset$
$Q_n \leftarrow \emptyset$ for $n = 0, 1, \ldots$
$R(t) \leftarrow \infty$ for each trip t
$\widehat{R}(l, j) \leftarrow \infty$ for each line l without schedule and each index $j = 0, 1, \ldots, |\overrightarrow{p}(l)|$
INITIALIZATION()
$\tau_{min} \leftarrow \infty$ ▷ The current minimum arrival time at target
$n \leftarrow 0$
while $Q_n \neq \emptyset$ **do**
  for each $p_t^b \rightarrow p_t^e \in Q_n$ **do**
    for each $(l_t, i, \Delta\tau) \in \mathcal{L}$ with $b < i$ and $\tau_{\text{arr}}(t, i) + \Delta\tau < \tau_{min}$ **do**
      $\tau_{min} \leftarrow \tau_{\text{arr}}(t, i) + \Delta\tau$ ▷ A target is reached and arrival time is improved
      $J \leftarrow J \cup \{(\tau_{min}, n)\}$, removing dominated entries

    if $\tau_{\text{arr}}(t, b+1) + \Delta\tau_{\text{al}}(l_t) < \tau_{min}$ **then** ▷ Filling the queue for the next round
      if $l_t \in \widehat{L}$ **then** ▷ Transfers must be computed
        for each stop p_t^i with $b < i \leq e$ **do**
          for each stop q such that $\Delta\tau_{\text{fp}}(p_t^i, q)$ is defined **do**
            $\tau \leftarrow \Delta\tau_{\text{fp}}(p_t^i, q) + \tau_{\text{arr}}(t, i) + \Delta\tau_{\text{al}}(l_t)$
            for each $(l, k) \in L(q)$ **do**
              $t' \leftarrow \text{earliest}(l, k, \tau + \Delta\tau_{\text{bo}}(l))$
              ENQUEUE_TRIP(t', k, $n+1$)
            for each $(l, k) \in \widehat{L}(q)$ **do**
              ENQUEUE_LINE(l, k, $\tau + \Delta\tau_{\text{bo}}(l), n+1$)
      else
        for each transfer $p_t^i \rightarrow p_u^j \in T$ with $b < i \leq e$ **do**
          ENQUEUE_TRIP(u, j, $n+1$)
        for each $(p_t^i \rightarrow p_l^j, \tau) \in \widehat{T}$ **do**
          ENQUEUE_LINE(l, j, τ, $n+1$)
  $n \leftarrow n + 1$

procedure INITIALIZATION
  for each stop q s.t. $\Delta\tau_{\text{fp}}(q, p_{\text{tgt}})$ is defined **do** ▷ Initialization of the target lines
    $\Delta\tau \leftarrow 0$ if $p_{\text{tgt}} = q$, else $\Delta\tau_{\text{fp}}(q, p_{\text{tgt}})$
    for each $(l, i) \in L(q) \cup \widehat{L}(q)$ **do**
      $\mathcal{L} \leftarrow \mathcal{L} \cup (l, i, \Delta\tau + \Delta\tau_{\text{al}}(l))$
  for each stop q s.t. $\Delta\tau_{\text{fp}}(p_{\text{src}}, q)$ is defined **do** ▷ Initialization of Q_0
    $\Delta\tau \leftarrow 0$ if $p_{\text{src}} = q$, else $\Delta\tau_{\text{fp}}(p_{\text{src}}, q)$
    for each $(l, i) \in L(q)$ **do**
      $t \leftarrow \text{earliest}(l, i, \tau + \Delta\tau_{\text{al}}(l))$
      ENQUEUE_TRIP(t, i, 0)
    for each $(l, i) \in \widehat{L}(q)$ **do**
      ENQUEUE_LINE(l, i, $\tau + \Delta\tau_{\text{al}}(l), 0$)

This process is as follows. For a non-scheduled line l, label $\widehat{R}(l)$ contains a set of pairs with the index of a stop and the earliest departure time at that stop. This set is such that an element (i, τ) of $\widehat{R}(l)$ is not dominated by any other element of $\widehat{R}(l)$. A pair (i, τ) is dominated by a pair (j, τ') if and only if

$$i \geq j \text{ and } \tau > \tau_{\text{dep}}(\text{earliest}(l, j, \tau'), i)$$

Indeed, if $i \geq j$, the trip is boarded later at i, missing some transfer opportunities compared to boarding it at j. And if $\tau > \tau_{\text{dep}}(\text{earliest}(l, j, \tau'), i)$, then the earliest trip that can be boarded at j after τ' brings you at the i^{th} stop earlier than τ. Hence, the set $\widehat{R}(l)$ contains at most $|\overrightarrow{p}(l)|$ elements and we can check the dominance of a new pair over the elements of the set in polynomial time. To maintain the elements of $\widehat{R}(l)$, one can save for each stop i of l the earliest departure time of a trip of l at that stop during the search. In that case, $\widehat{R}(l, i)$ represents the earliest departure time of l at its i^{th} stop in the current search. Another possibility is to sort the pairs of the set $\widehat{R}(l)$ by increasing stop index and to use the fact that the times are sorted in decreasing order to accelerate the dominance checks while needing less memory. Procedure UPDATE_R describes the update process of $\widehat{R}$ and computes the maximum index k for which $\widehat{R}(l, k)$ is modified, so as to determine the last element of the trip segment to add to the queue in the procedure ENQUEUE_LINE if $\widehat{R}$ is modified.

Note that since profile queries are an adaptation of earliest arrival time queries, it is possible to take them into account as proposed in [30] even after the modifications.

Algorithm 2 Earliest arrival query auxiliary procedures.

procedure ENQUEUE_TRIP(trip t, index i, number of transfers n)
  if $i < R(t)$ **then** ▷ Adding the given trip segment to the queue
    $Q_n \leftarrow Q_n \cup \{p_t^i \rightarrow p_t^{R(t)}\}$
    for each trip u with $t \leq u$ and $l_t = l_u$ **do**
      $R(u) \leftarrow \min(R(u), i)$

procedure ENQUEUE_LINE(line $l \in \widehat{L}$, index i, time τ, number of transfers n)
  $ind, t \leftarrow$ UPDATE_R(l, i, τ) ▷ Updating non-scheduled line labels
  if $ind \geq i$ **then** ▷ Adding the earliest trip segment to the queue
    $Q_n \leftarrow Q_n \cup \{p_t^i \rightarrow p_t^{ind}\}$

procedure UPDATE_R(line $l \in \widehat{L}$, index i, time τ)
  output Maximum index j s.t. $\widehat{R}(l, j)$ is modified, $i - 1$ if no modification
  $updated \leftarrow i$
  $t \leftarrow$ earliest(l, i, τ)
  while $updated \leq |\overrightarrow{p}(l)|$ and $\tau_{\text{dep}}(t, updated) < \widehat{R}(l, updated)$ **do**
    $\widehat{R}(l, updated) \leftarrow \tau_{\text{dep}}(t, updated)$
    $updated \leftarrow updated + 1$
    ▷ Trip is scanned to its end or stop is reached by an earlier trip
  return $updated - 1$, t

4.4 Complexity and correctness

Complexity. In [30], the complexity of the algorithm is not indicated. However, it can be shown that the algorithm performs a number of operations polynomial in the input size. We discuss here the worst case complexity for the non-scheduled line extension that we propose. An important difference is that only part of the instance is represented in the search graph.

The set of vertices V of the search graph contains the trips of the scheduled lines and the destination trips of the transfers of $\widehat{T}$. The number of elements in V is hence bounded by the number of trips of scheduled lines, denoted N_s, plus the size of $\widehat{T}$. Given an origin trip t, it would be possible to transfer from each stop of t (except the first one) to each stop (except the last one) of each non-scheduled line and to keep those transfers in $\widehat{T}$. We hence have $|\widehat{T}| = \mathcal{O}(N_s\,|\widehat{L}|\,|S|^2)$, if S is the set of stops, and $|V|$ is polynomial. Similarly, we can bound the number of elements of T by $|T| = \mathcal{O}(N_s^2\,|S|^2)$. The arcs A of the search graph represent the transfers of $T \cup \widehat{T}$. So $|A| = \mathcal{O}(N_s(N_s + |\widehat{L}|)|S|^2)$. Arcs from trips of non-scheduled lines are implicit. Computing the earliest trip of a line leaving a given stop after a time τ can be done efficiently using binary search in $\mathcal{O}(\log n)$ for scheduled lines, if n is the number of trips of the line and $\mathcal{O}(\log |S|)$ for non-scheduled lines. The set of earliest feasible transfers can hence be obtained in $\mathcal{O}(N_s(\log(N_s)N_s + \log(|S|)|\widehat{L}|)|S|^2)$. The pruning phase is also polynomial, as for each transfer, at most all the stops' labels must be updated.

For the initialization of the query phase, at most all the stops can be reached from p_{src} and all the lines taken, which takes $O(|S|\,|L \cup \widehat{L}|)$ 'ENQUEUE' operations. Then, at each iteration, we loop over the queue's content. For each trip segment in the queue, we first iterate over the targets (those number is bounded by $|S|\,|L \cup \widehat{L}|$). Then if it is from a scheduled line, we scan its outgoing arcs. At most, $|T \cup \widehat{T}|$ arcs are processed and elements are added to the queue. Otherwise, the worst case corresponds to the existence of a transfer from each stop of the non-scheduled line to each stop of any other line. It is hence bounded by $\mathcal{O}(|L \cup \widehat{L}|\,|S|^2)$. It results in a polynomial number of 'ENQUEUE' operations.

The ENQUEUE_TRIP procedure updates in the worst case the labels of all the trips of the line of its input trip t. It hence has complexity $\mathcal{O}(N_s)$. ENQUEUE_LINE updates at most $|\overrightarrow{p}(l)|$ labels of the set of labels of its input line l. It is hence bounded by $|S|$.

Overall, each step of the search phase is hence polynomial in the instance size.

To bound the number of iterations, first note that it is not possible to take twice the same trip in an optimal solution. A solution that alights a trip to board it again has at least one more transfer than the solution remaining on the current trip. It hence cannot be built by the algorithm and, for the original algorithm, the number of iterations is bounded by N_s. Taking twice the same line would be possible, for instance if the line is passing twice by the same stop, but not taking an earlier trip at a stop already passed by a preceding trip of the same line. The number of non-scheduled line trip segments in a solution built by the algorithm is hence bounded by $\mathcal{O}(|S|)$. Hence, the number of iterations is in $\mathcal{O}(N_s + |S|\,|\widehat{L}|)$.

Correctness. To prove that the algorithm is correct, we need to show that for any optimal solution in the Pareto set, there exists an optimal solution with the same value whose transfers are either in the pruned transfer set $T \cup \widehat{T}$ or are transfers from non-scheduled lines. Let s be an optimal solution with at least one transfer described by the trip segments that compose it: $s = \left\langle p_{t_1}^{j_1} \to p_{t_1}^{i_1}, p_{t_2}^{j_2} \to p_{t_2}^{i_2}, \ldots, p_{t_{K+1}}^{j_{K+1}} \to p_{t_{K+1}}^{i_{K+1}} \right\rangle$. If all its transfers are either in the transfer set $T \cup \widehat{T}$ or are transfers from non-scheduled lines, we are done. If not, from this solution, we build another optimal solution s' whose transfers are either in $T \cup \widehat{T}$ or are transfers from non-scheduled lines. First, iterating from $p_{t_2}^{j_2} \to p_{t_2}^{i_2}$, we replace the trip segments $p_{t_k}^{j_k} \to p_{t_k}^{i_k}$ that are not the earliest for which the transfer to l_{t_k} is possible from $p_{t_{k-1}}^{i_{k-1}}$. Since s is optimal, it is not possible to arrive sooner at stop $p_{t_{K+1}}^{i_{K+1}}$ and trip segment $p_{t_{K+1}}^{j_{K+1}} \to p_{t_{K+1}}^{i_{K+1}}$ is not modified. To simplify, we keep the same notation for the modified trip segments of s if any.

Table 1 Netherlands and IDF datasets.

Netherlands

# stops	# lines	# foot paths
47313	2773	429.4K

	# trips	# connections
Baseline	364.2K	6.527M
Non-sch.	317.7K	5.938M

IDF

# stops	# lines	# foot paths
42302	1869	752K

	# trips	# connections
Baseline	373.3K	7.867M
Non-sch.	318.0K	7.014M

Consider the last transfer $p_{t_K}^{i_K} \to p_{t_{K+1}}^{j_{K+1}}$ of s. If trip t_K is from a non-scheduled line, we keep it in s'. Otherwise, suppose that this transfer is not in $T \cup \widehat{T}$. In that case, there exists a transfer $p_{t_K}^{i'_K} \to p_{t'_{K+1}}^{j'_{K+1}}$ in $T \cup \widehat{T}$ from t_K such that $i'_K \geq i_k$ and t'_{K+1} arrives at $p_{t_{K+1}}^{i_{K+1}}$ at time $\tau_{\text{arr}}(t_{K+1}, i_{K+1})$ or before by definition of the pruning phase. Note that as the solution is optimal, it is exactly at time $\tau_{\text{arr}}(t_{K+1}, i_{K+1})$. If we denote with i'_{K+1} the smallest index in the stop sequence of t'_{K+1} such that $i'_{K+1} \geq j'_{K+1}$ and $p_{t_{K+1}}^{i_{K+1}} = p_{t'_{K+1}}^{i'_{K+1}}$, we can hence replace the two last trip segments by $p_{t_K}^{j_K} \to p_{t_K}^{i'_K}, p_{t'_{K+1}}^{j'_{K+1}} \to p_{t'_{K+1}}^{i'_{K+1}}$ in s'.

Proceeding likewise for the other transfers going backward in the solution, we can obtain a solution s' with the same value as s those transfers are all in $T \cup \widehat{T}$ or are transfers from a non-scheduled line.

5 Experiments

To the best of our knowledge, there is no open transit dataset with non-scheduled lines. One of the reason is that the most widespread data format, the GTFS format [19], does not provide specifications for defining non-scheduled lines. A recent proposal [23] extends it to some on-demand transports [29], but it doesn't cover the general case of non-scheduled lines, where the stop sequences of the lines are defined. Due to lack of standards, service providers usually develop their own methods for specification and integration of non-scheduled lines in their trip planners if they wish to propose them.

For our experiments, we modified public datasets for Netherlands [8] and Île-De-France [9] (IDF). The Netherlands dataset contains on-demand lines, but with predefined schedules, which require phone activation. From the perspective of the TB algorithm, this type of lines are handled as standard lines as they have predefined schedules and are not appropriate for our need. We hence slightly change the original dataset by converting 253 on-demand lines with predefined schedules to lines without schedule. For the IDF dataset, we obtain 201 non-scheduled lines. For each line, we set the availability period to 7.30 am to 7 pm for the first stop and translate the interval for each later stop of the line according to a fixed travel time between the origin stop and that stop. We denote by *Non-sch.* those datasets and we use the proposed algorithm to compute itineraries in those networks.

We also generate another variation of those datasets: instead of non-scheduled lines, we instantiated all the possible trips for the non-scheduled lines by generating one trip every two minutes in the interval. Those datasets are our baseline, as they allow to take into account non-scheduled lines without modification of the base algorithm.

Table 1 summarizes the datasets. Non sch. and baseline have the same number of stops, lines and foot paths, but the number of trips and connections (transfer between two consecutive stops taking a trip) differ. The experiments are run on a 2.7 GHz CPU Intel(R) Xeon(R) CPU E5-4650 server with 64 cores, 20M of L3 cache and 504 GB of RAM by using a solver developed in the Rust programming language.

Table 2 Preprocessing phase.

	Netherlands			IDF		
	Time (s)	# Transfers to scheduled lines	# Transfers to non-sch. lines	Time (s)	# Transfers to scheduled lines	# Transfers to non-sch. lines
Baseline	49	62.087M	0	75	90.183M	0
Non-sch.	46	60.306M	0.121M	73	84.212M	1.191M

Table 3 Query times for Baseline and Non-sch.

	Query	Netherlands			IDF		
		Mean time (ms)	Min time (ms)	Max time (ms)	Mean time (ms)	Min time (ms)	Max time (ms)
Baseline	Earliest arrival	60	23	173	56	21	153
	Profile 8.30	147	47	319	118	42	199
	Profile 14.30	134	44	244	131	41	238
Non-sch	Earliest arrival	45	18	147	43	8	106
	Profile 8.30	122	33	289	103	15	170
	Profile 14.30	99	25	268	102	13	183

Preprocessing times of the two settings are similar, although the setting using non-scheduled lines is slightly faster, as it has less trips to process (see Table 2). Remark that the preprocessing time is low enough to allow for real-time updates of the network every couple of minutes. It implies that in the case when a non-scheduled trip is booked, it is possible to update the network to include it as a scheduled trip. The availability intervals of some stops-line pairs can also be modified to take into account the fact that this booked vehicle is no longer available.

To compare query times, we selected randomly 300 origin and destination pairs over each network. We generated 3 queries per origin-destination pair: an earliest arrival time query and two profile queries. For each query, a complete set of solutions is computed. We fixed the departure times of the earliest arrival time queries at 8.30 (am), a time at which trips are usually more frequent (which makes the exploration more time consuming) and for profile queries, we considered time intervals of length one hour, starting at 8.30 and 14.30 respectively. Profile queries starting at 14.30, a time where trip frequencies are less high, result in fewer solutions and are hence expected to run faster than the ones starting at 8.30. Results of the experiments can be found in Table 3.

For our experiments, we turned about 10% of the lines into non-scheduled lines, and hence cannot expect a huge difference in query times. However, the difference is significant enough for the method to be interesting from a performance point of view, as mean query times are between 13% and 27% faster than that of the baseline version (see Table 4).

Table 4 Non-sch. query times divided by baseline query times.

	Netherlands			IDF		
Query	Mean time	Min time	Max time	Mean time	Min time	Max time
Earliest arrival	0.75	0.78	0.85	0.77	0.38	0.74
Profile 8.30	0.83	0.70	0.91	0.87	0.36	0.85
Profile 14.30	0.73	0.57	1.1	0.78	0.32	0.77

6 Conclusion and future work

In this article, we proposed a method for computing itineraries in public transit or multimodal networks with scheduled and non-scheduled lines. It extends the Trip-Based Public Transit Routing algorithm to on-demand lines with a predefined stop sequence and availability intervals but no associated schedules. Experimental results over two large datasets show that the proposed approach performs better than the baseline consisting in discretizing the availability interval to generate all the possible trips for the non-scheduled lines.

This model has car and bike sharing as a special case. A perspective of our work could hence be to test our method with multimodal networks including those modes, against classical modeling as a transfer, and not as a trip. Another line of work could be concerned with applying classical acceleration techniques, such as Transfer Patterns [4, 6], to the proposed algorithm. Transfer patterns have been adapted to Trip-Based Public Transit Routing in [31] and could be extended to take into account non-scheduled lines.

References

1 Transports montalbanais. Access date: 2021/03/29. URL: `https://www.montm.com/transport-a-la-demande-et-pmr/`.

2 Chris Barrett, Riko Jacob, and Madhav Marathe. Formal-language-constrained path problems. *SIAM J. Comput.*, 30(3):809–837, May 2000. `doi:10.1137/S0097539798337716`.

3 Hannah Bast. Car or Public Transport – Two Worlds. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, pages 355–367. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. `doi:10.1007/978-3-642-03456-5_24`.

4 Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *Proceedings of the 18th Annual European Conference on Algorithms: Part I*, ESA'10, pages 290–301, Berlin, Heidelberg, 2010. Springer-Verlag.

5 Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In *Kliemann L., Sanders P. (eds) Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, Cham, 2016. `doi:10.1007/978-3-319-49487-6_2`.

6 Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable Transfer Patterns. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 15–29, January 2016. `doi:10.1137/1.9781611974317.2`.

7 Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ESA.2019.14`.

8 datahub. Timetables for transit in netherlands. Access date: 2019/07/29. URL: `https://old.datahub.io/dataset/gtfs-nl`.

9 Île de France Mobilités. Open data portal. Access date: 2020/05/04. URL: `https://data.iledefrance-mobilites.fr/pages/home/`.

10 Daniel Delling, Julian Dibbelt, and Thomas Pajor. Fast and Exact Public Transit Routing with Restricted Pareto Sets. In *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 54–65, San Diego, California, USA, 2019. Editor(s): Stephen Kobourov and Henning Meyerhenke. `doi:10.1137/1.9781611975499.5`.

11 Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing Multimodal Journeys in Practice. In *Experimental Algorithms - Proceedings of the 12th International Symposium, SEA 2013*, volume 7933 of *Lecture Notes in Computer Science*, pages 260–271. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38527-8_24.

12 Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. Public Transit Labeling. In Evripidis Bampis, editor, *Experimental Algorithms - Proceedings of the 14th International Symposium (SEA 2015)*, volume 9125 of *Lecture Notes in Computer Science*, pages 273–285. Springer International Publishing, 2015. doi:10.1007/978-3-319-20086-6_21.

13 Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. In Society for Industrial and Applied Mathematics, editors, *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 130–140, 2012. doi:10.1287/trsc.2014.0534.

14 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly Simple and Fast Transit Routing. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms. International Symposium on Experimental Algorithms, SEA 2013*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-38527-8_6.

15 Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. User-Constrained Multi-Modal Route Planning. In *Proceedings of the 14th Workshop on Algorithm Engineering and Experiments (ALENEX'12)*, pages 118–129. SIAM, 2012. Editors David A. Bader and Petra Mutzel. doi:10.1137/1.9781611972924.12.

16 Agglo du Pays de Dreux. Linéad. Access date: 2021/03/29. URL: https://www.linead.fr/8-Transport-a-la-demande.html.

17 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In C.C. McGeoch, editor, *Experimental Algorithms. 7th Workshop on Experimental Algorithms (WEA 2008)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-68552-4_24.

18 Andrew 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.

19 General transit feed specification. Access date: 2021/03/29. URL: https://gtfs.org/.

20 Pierre Hansen. Bicriterion Path Problems. In Günter Fandel and Tomas Gal, editors, *Multiple Criteria Decision Making Theory and Application*, volume 177 of *Lecture Notes in Economics and Mathematical Systems*, pages 109–127. Springer Berlin Heidelberg, 1980. doi:10.1007/978-3-642-48782-8_9.

21 Dominik Kirchler, Leo Liberti, and Roberto Wolfler Calvo. Efficient Computation of Shortest Paths in Time-Dependent Multi-Modal Networks. *ACM Journal of Experimental Algorithmics (JEA)*, 19, January 2015. doi:10.1145/2670126.

22 Vassilissa Lehoux and Darko Drakulic. Mode Personalization in Trip-Based Transit Routing. In Valentina Cacchiani and Alberto Marchetti-Spaccamela, editors, *19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019)*, volume 75 of *OpenAccess Series in Informatics (OASIcs)*, pages 13:1–13:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.ATMOS.2019.13.

23 Ross MacDonald. Mobility on demand (mod) sandbox: Vermont agency of transportation (vtrans) flexible trip planner. Technical Report 0150, Federal Transit Administration (FTA) Research, 2020.

24 Transports publics de Flers Agglo. Némus. Access date: 2021/03/29. URL: https://nemus.flers-agglo.fr/se-deplacer/transport-a-la-demande.

25 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics (JEA)*, 12(2.4):1–39, 2008. doi:10.1145/1227161.1227166.

26 Andrea Raith, Marie Schmidt, Anita Schöbel, and Lisa Thom. Extensions of labeling algorithms for multi-objective uncertain shortest path problems. *Networks*, 72(1):84–127, 2018. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.21815. doi:10.1002/net.21815.

27 Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. Faster Multi-Modal Route Planning With Bike Sharing Using ULTRA. In Simone Faro and Domenico Cantone, editors, *18th International Symposium on Experimental Algorithms (SEA 2020)*, volume 160 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SEA.2020.16.

28 Luis Ulloa, Vassilissa Lehoux, and Frédéric Roulland. Trip Planning Within a Multimodal Urban Mobility. *IET Intelligent Transport Systems*, 12(2):87–92, 2018. doi:10.1049/iet-its.2016.0265.

29 GTFS-Flex v2. Flexible public transit services in gtfs. URL: https://github.com/MobilityData/gtfs-flex/blob/master/spec/reference.md.

30 Sascha Witt. Trip-Based Public Transit Routing. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, volume 9294 of *Lecture Notes in Computer Science*, pages 1025–1036, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. Editors: Nikhil Bansal and Irene Finocchi. doi:10.1007/978-3-662-48350-3_85.

31 Sascha Witt. Trip-Based Public Transit Routing Using Condensed Search Trees. In Marc Goerigk and Renato Werneck, editors, *Proceedings of the 16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2016)*, volume 54 of *OpenAccess Series in Informatics (OASIcs)*, pages 10:1–12, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Editors: Marc Goerigk and Renato Werneck. doi:10.4230/OASIcs.ATMOS.2016.10.

Relating Real and Synthetic Social Networks Through Centrality Measures

Maria J. Blesa
Computer Science Department, Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

Mihail Eduard Popa
Barcelona School of Informatics, Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

Maria Serna
Computer Science Department, Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
Institute of Mathematics (IMTech), Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

Abstract

We perform here a comparative study on the behaviour of real and synthetic social networks with respect to a selection of nine centrality measures. Some of them are topology based (degree, closeness, betweenness), while others consider the relevance of the actors within the network (Katz, PageRank) or their ability to spread influence through it (Independent Cascade rank, Linear Threshold Rank). We run different experiments on synthetic social networks, with 1K, 10K, and 100K nodes, generated according to the Gaussian Random partition model, the stochastic block model, the LFR benchmark graph model and hyperbolic geometric graphs model. Some real social networks are also considered, with the aim of discovering how do they relate to the synthetic models in terms of centrality. Apart from usual statistical measures, we perform a correlation analysis between all the nine measures. Our results indicate that, in general, the correlation matrices of the different models scale nicely with size. Moreover, the correlation plots distinguish four categories that classify most of the real networks studied here. Those categories have a clear correspondence with particular configurations of the models for synthetic networks.

2012 ACM Subject Classification Networks → Network algorithms; Networks → Network dynamics

Keywords and phrases centrality measures, influence spread models, synthetic social networks

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.7

Supplementary Material *Other (Experimental Results)*: `https://www.cs.upc.edu/ mjblesa/centrality/syntheticGraphs/`

Funding M. Blesa and M. Serna acknowledge support by MICIN/AEI/10.13039/501100011033 under grant PID2020-112581GB-C21 (MOTION). M.E. Popa was funded by the Ministry of Education and Vocational Training (MEFP) with a student grant (Beca de colaboración, call 2020-2021).

1 Introduction

Nowadays, social media are more and more integrated in our daily lives leading to the emergence of varied and complex social networks. One of the main research questions is to understand the relevant characteristics of those huge networks. In network analysis, indicators of centrality identify the most important vertices within a graph with respect to some particular characteristic. Centrality concepts were first developed in social network analysis, and many of the terms used to measure them reflect that sociological origin [17]. They should not be confused with node influence metrics, which seek to quantify the influence of every node in the network. Traditional measures are *degree*, *closeness* and *betweenness* which are topology dependant. Other well-known centrality measures are the Katz Rank [8] and the PageRank [19]. Two new measures have been introduced in an attempt to measure centrality with respect to influence spreading, the Independent Cascade (ICR) [9] and the Linear Threshold Rank (LTR) [4].

Many efforts have been devoted to understand the relationship among different centrality measures on real networks (see for example [26, 22, 18] and references herein). [18] show that network topology determines the correlation pattern among several measures and the combination of several centrality measures can help in the interpretation of the roles that a node or a group of nodes play in the network. In all these studies the evaluation has been performed in a selection of real networks. Our objective is to understand whether the parameters and correlation patterns among a small number of centrality measure can identify a family of synthetic social networks. Eventually, we would like to use such patterns to associate a correct model to real networks. Recall that, after a surge in interest in network structure among mathematicians and physicists, a body of research has been devoted to modeling networks either analytically or numerically. We focus our attention on random models generating synthetic networks from a priori knowledge on its communities. In particular, we generate graphs according to the *Gaussian Random partition* model [25], the *stochastic block* model [20], and the *Lancichinetti–Fortunato–Radicchi (LFR) benchmark* [13]. These models are accepted to be good generators for benchmarks in community detection [3]. We complement the study analyzing *hyperbolic geometric graphs* [12], known as graphs showing properties expected in large real network graphs, and the behaviour of the centrality on some real social networks.

We experimentally evaluate the behaviour of the considered centrality measures on the selected models of synthetic networks and perform a stochastic comparison among them. To compare the different centrality measures, we have extracted three statistics measuring diversity: the standard deviation, the number of different values, and the Gini coefficient. The second component is a correlation analysis. We use the Kendall [11] and Spearman [24] coefficients. Our results show, as expected, that the considered models of synthetic social networks perform differently with respect to the centrality measures. Besides this, our results indicate that, in general, the correlation matrices on the different models scale nicely with size. Furthermore, we observe that the correlation plots distinguish different graph families. From the correlation plots, we have been able to obtain four categories classifying most of the real networks studied here. Furthermore, such categories are identified with submodels of the synthetic networks. The unclassified networks are very small or a particular structure which hints to another type of generator will be needed for those networks.

2 Centrality Measures

We outline the centrality measures used in the paper. Some of them are based on the topological properties of the nodes, some others take into account the relevance of the actors, while other centrality measures quantify the influence exerted by the actors on the network in terms of diffusion power. Given a directed graph $G = (V, E)$, where $|V| = n$, we consider:

2.1 Topology based

Degree. This is one of the simplest centrality measures and simply consists on assigning the centrality based of the degree of the node. For every node $i \in V$, the degree centrality of i is the degree itself of i (δ_i), normalized by the number of nodes minus one, i.e.,

$$\mathsf{Deg}(i) = \frac{\delta_i}{n-1}.$$

Closeness. For the closeness measure, the distance to other nodes is considered. Intuitively, the more central a node is, the closer it is to all other nodes. For every node $i \in V$, the closeness of i is calculated as the normalized reciprocal of the sum of the length of the shortest path distances between i and all other nodes $j \in V \setminus \{i\}$ in the graph.

$$\mathsf{Clsn}(i) = \frac{1/\sum_j d(j,i)}{(n-1)} = \frac{n-1}{\sum_j d(j,i)}.$$

In order to be able to compute the closeness measure also for the nodes of networks which are not strongly connected, we will consider the adaptation in [27], defined as follows:

$$\mathsf{Clsn}(i) = \frac{J_i/(n-1)}{\sum_j d(j,i)/J_i}.$$

where J_i is the number of actors in the influence range of actor i, i.e., the number of actor who are reachable from i.

Betweenness. In the betweenness centrality, a node is more important if it belongs to the shortest path between any pair of nodes in the graph [5]. Given $G = (V, E)$, for every node $i \in V$, we define the betweenness as the sum $\forall s \forall t \in V$ of the proportion of $\sigma_{st}(i)$ (the shortest paths between s, t that go through i) with respect to σ_{st} (all the shortest path between s and t), i.e.,

$$\mathsf{Btwn}(i) = \sum_{s \neq i \neq t} \frac{\sigma_{st}(i)}{\sigma_{st}}.$$

2.2 Relevance based

Katz. The Katz centrality [8] is a generalization of degree centrality and it can also be viewed as a variant of eigenvector centrality. While degree centrality measures the number of direct neighbors, the Katz centrality measures the number of all nodes that can be connected through a path, while the contributions of distant nodes are penalized. It is based on the idea that an actor is important if it is linked to other important actors or if it is highly linked. It overcomes the limitations of the eigenvector centrality when the graph has nodes that reach strongly connected components, but those connected components do not reach the node, which may occur in social networks.

Let A be the adjacency matrix of the directed graph $G = (V, E)$ (i.e., $a_{ij} = 1$ if there is an edge between i and j, and a $a_{ij} = 0$ otherwise). Let β be a constant independent from the structure of the social network and $\alpha \in [0, \ldots, {\lambda_{max}}^{-1}]$ is the damping factor, being λ_{max} the highest eigenvalue in A. Then the $Katz(i)$ is defined as

$$\mathsf{Ktz}(i) = \alpha \sum_{j \in V} a_{ji}\, \mathsf{Ktz}(j) + \beta.$$

PageRank. One of the most popular centrality measures is the PageRank [19], that Google uses to assign importance to web pages. A web page is important if other important web pages point to it. It uses a parameter $\alpha \in (0, 1]$, that represents the probability that a user keeps jumping from a web page to another through the links that are between them (and thus, $1 - \alpha$ represents the probability that the user goes to a random web page). Let A be

the adjacency matrix of a directed graph $G = (V, E)$ (i.e., $a_{ij} = 1$ if there is an edge between i y j, and a $a_{ij} = 0$ otherwise), and let $\delta^+(i)$ be the out degree of $i \in V$. The PageRank (PR) of i is given by

$$\mathsf{PgR}(i) = (1 - \alpha) + \alpha \sum_{j \in V} \frac{a_{ji}\, \mathsf{PgR}(j)}{\delta^+(j)}.$$

2.3 Influence based

Perhaps the two most prevalent diffusion models in computer science are the *Independent Cascade* model [7] and the *Linear Threshold* model [9, 23]. Based on them, the corresponding influence-based centrality measures are defined:

Independent Cascade Rank. The Independent Cascade Rank [10] is an influence-based centrality measure based on the Independent Cascade Model (ICM) [7], which is a stochastic model that was initially proposed in the context of marketing. It is based on the assumption that whenever a node is activated, it will (stochastically) do attempt to activate each actor he targets. Given an activated node $i \in V$, any neighbor j such that $(i, j) \in E$ will be activated with a probability p_{ij}. When a new actor is activated, the process is repeated for this actor. The whole process ends when there are no active nodes with a new chance to spread its influence.

Given an initial node $u \in V$ and a probability $p \in [0, 1]$ (where $\forall (i, j) \in E : p_{ij} = p$), the expected influence spread of u is denoted by $F'(u, p)$ and comprises the set of activated nodes under the ICM influence model, starting from the initial node u. Then, the Independent Cascade Rank of a node $u \in V$ is then defined as

$$\mathsf{ICR}(u, p) = \frac{|F'(u, p)|}{\max_{v \in V}\{|F'(v, p)|\}}.$$

Linear Threshold Rank. The Linear Threshold Rank [4] is also an influence-based centrality measure, based on the Linear Threshold Model (LTM) [9]. Every node has an influence threshold, which represents the resistance of this node to be influenced by others. Every edge (u, v) also has a weight representing the influence that node u has over node v.

The influence algorithm starts with an initial predefined set of activated nodes. At every iteration, the active nodes will influence their neighbors. When the total influence that a node i receives exceeds its influence threshold θ_i, then this node will become active and join the set of active nodes. As long as new nodes join the set of active nodes, the spread of influence is still on progress. The algorithm stops when the set of active nodes converges, i.e., when no new nodes are influenced. In order to formally define the Linear Threshold Rank, we need to introduce the following concepts:

▶ **Definition 1.** *An influence graph is a tuple (G, w, θ), where $G = (V, E)$ is a directed graph made by a set of actors V and a set of relations $E, w : E \to \mathbb{Z}$ is a weight function that assigns a weight to each edge, representing the influence of one node to the other, and $\theta : V \to \mathbb{N}$ is a labeling function that quantifies how resistant to influence every node is.*

▶ **Definition 2.** *Given an influence graph (G, w, θ) and an initial active set $X \subseteq V$, $F_t(X) \subseteq V$ denotes the set of activated nodes at the t-th iteration starting with X as kernel.*

At the first step, $t = 0$ only the nodes in X are active, which means that $F_0(X) = X$. At the $t + 1$ iteration, a node i will be activated if, and only if, the sum of all the weights of the active nodes incident to i is higher than the resistance (or influence) threshold of i, i.e.,

$$\sum_{j \subseteq F_t(X)} w_{ij} \geq \theta_i,$$

Observe that the process is monotonic, therefore it stops after at most $n = |V|$ steps.

▶ **Definition 3.** *Let $k = min\ \{t \in \mathbb{N} \mid F_t(X) = 0\}$, where $k \leq n$. The expansion of $X \subseteq V$ on an influence graph $(G = (E, V), w, \theta)$, is defined as $F(X) = \bigcup_{t=0}^{k} F_t(X)$.*

Given an influence graph $(G = (V, E), w, \theta)$, the Linear Threshold Rank of a node $i \in V$ is given by

$$\mathsf{LTR}(i) = \frac{|F(\{i\} \cup \mathcal{N}(i))|}{|V|}, \text{ where } \mathcal{N}(u) = \{v \mid (u, v) \in E \vee (v, u) \in E\}.$$

Forward and Backward Linear Threshold Rank. The Forward and the Backward Linear Threshold Ranks [1] are centrality measures similar to the LTR, but with a different initial set of activated nodes. Given an influence graph $(G = (V, E), w, f)$, the Forward Linear Threshold Rank and the Backward Linear Threshold Rank of a node $i \in V$ is given, respectively, by

$$\mathsf{FwLTR}(i) = \frac{|F(\{i\} \cup \mathcal{N}^+(i))|}{|V|} \quad \text{and} \quad \mathsf{BwLTR}(i) = \frac{|F(\{i\} \cup \mathcal{N}^-(i))|}{|V|}$$

where $\mathcal{N}^+(i) = \{j \in V \mid (i, j) \in E\}$ and $\mathcal{N}^-(i) = \{j \in V \mid (j, i) \in E\}$.

3 Social Networks

We describe the characteristics of the synthetic networks considered in this work, as well as the real social networks chosen for our experiments. The structural characteristics of the networks are described by seven common attributes: the number of vertices, the number of edges, whether the graph is weighted, whether the graph is directed, the average clustering coefficient, and the size of the main core.

The *average clustering coefficient* (ACC) is the average of the local clustering coefficients in the graph. The local clustering coefficient C_i of a node i is the number of triangles T_i in which the node participates normalized by the maximum number of triangles that the node could participate in.

$$\text{ACC} = \frac{1}{n} \sum_{i=1}^{n} C_i, \quad \text{where } C_i = \frac{T_i}{\delta_i(\delta_i - 1)}$$

where δ_i is the degree of the node i, and $n = |V|$. Given a graph G and $k \in \mathbb{Z}^+$, a *k-core* is the maximal induced subgraph of G where every node has at least degree k. The *main core* is a k-core of G with the highest k.

3.1 Synthetic Social Networks

Concerning the models for synthetic social networks, we have considered four different models: Gaussian Random Partition Graphs, the Stochastic Block Model, LFR Benchmark Graphs and Hyperbolic Geometric Graphs.

3.1.1 Gaussian Random Partition Graph (GRP)

The process to create a Gaussian Random Partition Graph [25] starts by creating k partitions of different size. Those sizes will be taken from a normal distribution $\mathcal{N}(\mu, \sigma^2)$. Two nodes from the same partition are connected with probability p_{in}, while two nodes from two different partitions will be connected with probability p_{out}. To generate this type of graph we will be using the implementation from NetworkX [14], which lets us assign values for the following parameters:

- n: the number of nodes of the network,
- μ: mean of the sizes of the partition in the graph,
- σ: variance of the sizes of the partition in the graph,
- p_{in}: probability of generating a intracluster edge,
- p_{out}: probability of generating a intercluster edge,
- dir: whether or not the graph is directed.

We created five different types of graphs, that we denote as GRPa, GRPb, GRPc, GRPd and GRPe (see Table 1). They consider different size and variance of partitions, and also different probabilities for creating intracluster and intercluster edges. All categories are directed, except for GRPd. The choice for these parameters is based on the ones proposed in [25], but conveniently adapted to represent bigger meaningful social networks.

3.1.2 Stochastic Block Model (SBM)

The construction of a Stochastic Block Model Graph [20] starts by partitioning the nodes of the network into blocks of arbitrary sizes. Secondly, edges are placed between pairs of nodes independently, with a probability that depends on the blocks, i.e., the probability to create an edge (u, v) depends on the probability of connection defined between the cluster of u and the cluster of v.

To generate this type of graph we will be using the implementation from NetworkX [16], which lets us assign values for the following parameters:

- n: the approximate number of nodes of the network,
- k: the number of blocks within the network,
- $S = \{s_1, \ldots, s_k\}$: the list of block sizes, where s_i denotes the number of nodes in the block i.
- $P \in k^2$: a probability matrix, where p_{ij} is the probability of creating an (intercluster) edge between a node in cluster i and a node in cluster j. Observe that p_{ii} is then the probability of an intracluster edge within block i.

We created five different types of graphs, that we denote as SBMa, SBMb, SBMc, SBMd and SBMe (see Table 2). The sizes of the blocks are created according to different statistical distributions. We use exponential distributions for all the types of graphs, except for the SMBb, where a normal distribution is used. All categories are directed.

We can observe that all the networks have the order of $\theta(\sqrt{n})$ number of clusters, except for SBMd, which has less clusters but of bigger size. In general, we wanted to work with very different cluster sizes. For that reason, in most of the cases we used the exponential distribution to generate S. For the SBMb we used the normal distribution instead. We want to see whether big differences on the size of the blocks affect the final result.

Table 1 Parameters for the Gaussian Random Partition Graphs. For each type, three different graph sizes are considered: $n = 1000$, $n = 10000$ and $n = 100000$. For the probabilities, $f(n) = \frac{\log(n)}{\mu}$ and $g(n) = \frac{\log(n)}{n-\mu}$.

Name	μ	σ	p_{in}	p_{out}	dir
GRPa	$n/10\sqrt{n}$	4	$3/4\ f(n)$	$1/4\ g(n)$	True
GRPb	$n/10\sqrt{n}$	2	$3/4\ f(n)$	$1/4\ g(n)$	True
GRPc	$n/10\sqrt{n}$	4	$1/2\ f(n)$	$1/2\ g(n)$	True
GRPd	$n/10\sqrt{n}$	4	$3/4\ f(n)$	$1/4\ g(n)$	False
GRPe	$n/\sqrt{n}$	4	$3/4\ f(n)$	$1/4\ g(n)$	True

Table 2 Parameters for the Stochastic Block Model Graphs. For each type, three different graph sizes are considered: $n = 1000$, $n = 10000$ and $n = 100000$. $S \sim Exp(\lambda)$ is a sample from an exponential distribution with rate λ, and $S \sim \mathcal{N}(\mu, \sigma^2)$ is a normal distribution with mean μ and standard deviation σ^2. For the probabilities, $f(n) = \log(n)/\overline{S}$ and $g(n) = \log(n)/(n - \overline{S})$, where $\overline{S}$ represents the mean size of the blocks.

Name	k	S	p_{ii}	p_{ij}
SBMa	$10\sqrt{n}$	$Exp(10/\sqrt{n})$	$3/4\ f(n)$	$1/4\ g(n)$
SBMb	$10\sqrt{n}$	$\mathcal{N}(k/100, k/1000)$	$3/4\ f(n)$	$1/4\ g(n)$
SBMc	$10\sqrt{n}$	$Exp(10/\sqrt{n})$	$1/2\ f(n)$	$1/2\ g(n)$
SBMd	$\sqrt{n}$	$Exp(1/\sqrt{n})$	$3/4\ f(n)$	$1/4\ g(n)$
SBMe	$10\sqrt{n}$	$Exp(10/\sqrt{n})$	$3/4\ \frac{\log(n)}{s_i}$	$1/4\frac{\log(n)}{n-s_i}$

3.1.3 LFR Benchmark Graph

The LFR Benchmark [13] is a model for graph generation more complex than GRP and SBM are. Consequently, it allows to create artificial networks that are significantly more similar to real ones. In a very summarized way, the algorithm starts finding a power law distribution for the degree of the nodes. Every node will have a proportion μ of its connections to nodes belonging to other communities (intercluster), whereas the remaining $(1 - \mu)$ proportion of its edges will be attached to nodes in same the community (intracluster). This leads to the emergence of communities of different sizes (following a power law distribution as well). Each node will be randomly assigned to one community following the constraint imposed by μ.

To generate this type of graph we will be using the implementation from NetworkX [15], which lets us assign values for a lot of different parameters. After a deep study of them, where we detected some incompatibilities, the following parameters where identified as the most relevant and worth to play with:

- n: the number of nodes that our social network will have,
- τ_1: exponent of the power law distribution for the node degree distribution,
- τ_2: exponent of the power law distribution for the community size distribution,
- μ: proportion of intracluster edges for each node,
- max_c: maximum community size in the graph,
- min_c: minimum community size in the graph,
- max_d: maximum node degree in the graph,
- avg_d: mean node degree in the graph.

We create five different types of graphs, that we denote as LFRa, LFRb, LFRc, LFRd and LFRe (see Table 3). We have fixed the parameters τ_1 and τ_2 to the values proposed in [13]: the adequate values to represent social networks oscillate between $2 \leq \tau_1 \leq 3$ and $1 \leq \tau_2 \leq 2$. We do not work with $\tau_2 = 1$ because the generator implemented in NetworkX demands that $\tau_2 > 1$. LFRc builds bigger communities than the rest of the graphs, and in LFRe the proportion of intracluster edges is greater than the usual 0.2.

Table 3 Parameters for the LFR Benchmark Graphs. For each type, three different graph sizes are considered: $n = 1000$, $n = 10000$ and $n = 100000$.

Name	τ_1	τ_2	μ	max_c	min_c	max_d	avg_d
LFRa	2	1.1	0.2	$0.05\ n$	$max_c/100$	$0.05\ n$	$5/4\ \log(n)$
LFRb	2	2	0.2	$0.05\ n$	$max_c/100$	$0.05\ n$	$5/4\ \log(n)$
LFRc	2	1.1	0.2	$0.05\ n$	$max_c/10$	$0.05\ n$	$5/4\ \log(n)$
LFRd	3	2	0.2	$0.05\ n$	$max_c/100$	$0.05\ n$	$5/4\ \log(n)$
LFRe	2	1.1	0.35	$0.05\ n$	$max_c/100$	$0.05\ n$	$5/4\ \log(n)$

Table 4 Parameters for the Hyperbolic Geometric Graphs. For each type, three different graph sizes are considered: $n = 1000$, $n = 10000$ and $n = 100000$.

Name	k	γ	t	z
HYPa	$\log(n)$	2	0	1
HYPb	$\log(n)$	2	2	1
HYPc	$\log(n)$	3	0	1
HYPd	$\log(n)$	3	0.5	1
HYPe	$\log(n)$	∞	0.5	1

Table 5 Real data sets considered in this work. Shadowed rows state for directed networks. ACC = Average Clustering Coefficient, MC = size of the main core. The diameter is ∞ when the graph is not connected (or not strongly connected in the case of digraphs); in these cases, the diameter of the biggest connected component is provided. The subscript (w) indicates edge weighted networks.

Networks	n	m	**ACC**	**Diameter**	**MC**
Dining Table $_{(w)}$	26	52	0.1178	∞ (6)	20
Dolphins	62	159	0.2590	8	36
Human Brain $_{(w)}$	480	1000	0.3004	∞ (20)	11
ArXiv	5242	14496	0.5296	∞ (17)	44
Wikipedia	7115	103689	0.1409	7	336
Caida $_{(w)}$	26475	106762	0.2082	17	50
ENRON	36692	183831	0.4970	11	275
Gnutella	62586	147892	0.0055	11	1004
Epinions	75879	508837	0.1378	14	422
Higgs $_{(w)}$	256491	328132	0.0156	19	10
Amazon	334863	925872	0.3967	44	497
Texas	1379917	1921660	0.0470	1054	1579

The parameters max_c and max_d are both fixed to $0.05\ n$ for all the networks. For lower values, the resulting graphs are not good representations for social networks. For bigger values, the generator takes an enormous amount of time to converge (even with a very small number of vertices) or even does not converge at all. In the same sense, another problematic parameter was avg_d. When fixed to $avg_d = \log(n)$, the generator was no always converging and thus, we had to slightly increase that value. In our case, that increase implies increasing the average degree by one.

3.1.4 Hyperbolic Geometric Graph

Recent studies in graph geometry showed that many networks appearing in nature or representing societies can be modeled as geometric graphs in hyperbolic spaces [21]. Based on this fact, hyperbolic geometric graphs have started to be used as models for synthetic social networks. In our study we will be using the generator proposed in [21], which allows us to configure the following parameters:

- n: the number of nodes that our social network will have,
- k: mean node degree in the graph,
- γ: exponent of the power law distribution for the node degree distribution,
- t: temperature,
- z: square root of the curvature of the hyperbolic space.

Using these parameters, we create five different types of graphs that we denote as HYPa, HYPb, HYPc, HYPd and HYPe (see Table 4). Most of the values we work with are those suggested in [21] for generating hyperbolic geometric graphs representing social networks.

3.2 Real Social Networks

We will also be using real social networks in our experiments in order to have a point of view of what happens in reality and to see how our artificial models really compare. We consider twelve well-known real social networks, which are mostly available at the `snap.stanford.edu` and the `networkrepository.com` repositories. The main characteristics of these networks are summarized in Table 5.

4 Statistical Metrics

For analysing the results of a centrality measure on its own, three statistical metrics will be used: the number of different ranks, the standard deviations and the Gini coefficient of the distribution. The Gini coefficient [6, 2] comes originally from sociology as a measure of the inequality of populations with respect to different criteria (e.g., wealth spread), but it is lately being used as a measure for quantifying the fairness of distributions in other areas.

▶ **Definition 4.** *Given a list of values $\mathcal{X}$ of size n, the Gini coefficient of $\mathcal{X}$ is calculated as:*

$$Gini(\mathcal{X}) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}|x_i - x_j|}{2n\sum_{i=1}^{n} x_i}$$

As observed in other works, we will also use two well-known correlation measures for comparing the centrality ranks among themselves: the Spearman's correlation coefficient [24] and the Kendall correlation coefficient [11].

▶ **Definition 5.** *Given two lists of elements $\mathcal{X}$, $\mathcal{Y}$ both with n elements, the Spearman's rank correlation coefficient (ρ) is equal to:*

$$\rho(\mathcal{X},\mathcal{Y}) = 1 - \frac{6\sum_{i=1}^{n}(x_i - y_i)^2}{n(n^2-1)}$$

▶ **Definition 6.** *Let $\mathcal{X}$ and $\mathcal{Y}$ be two lists of elements, then the Kendall's rank correlation coefficient (τ) is defined as:*

$$\tau(\mathcal{X},\mathcal{Y}) = \frac{n_c - n_d}{0.5n(n-1)}$$

where n_c is the number of concordant pairs between $\mathcal{X}$ and $\mathcal{Y}$, and n_d is the number of discordant pairs. A pair (i,j) is concordant if either $x_i > x_j$ and $y_i > y_j$, or $x_i < x_j$ and $y_i < y_j$. A discordant pair is one that is not concordant.

5 Experiments and results

We have generated five different configurations for each of the four synthetic models. Then we have calculated the centrality measures and statistically analyzed the results obtained on those twenty configurations under different perspectives. In particular, the correlation analysis between the different centrality measures gives us an insight about which metrics seem to behave similarly in specific types of networks. The studies carried out can help us understand the similarities of the centrality measures under diffusion processes in different types of networks, but we do not believe that at the moment they can provide any information for their inverse use, that is, on how to generate these networks.

Undirected networks are transformed into directed ones by replacing edges into bidirectional arcs. For unweighted networks, edge weights are all fixed to 1. As in [4], we fix the threshold function θ_i of every actor i to the simple majority rule. ICR works with probability $p = 0.1$. Here we can only summarize the most relevant observations. For a detailed vision of the results and exact data on them, we point the reader to `www.cs.upc.edu/~mjblesa/centrality/syntheticGraphs/`.

5.1 Statistics results

We can observe some very clear trends when looking at the complete results of our experiments. Starting with the GRP models (see Figure 1), the deviation and the number of different ranks are the highest in GRPb and lowest in GRPd. The Gini coefficient is still high in GRPb, but it is also high in GRPc, the latter is still the lowest for some metrics such as Closeness and Pagerank. The rest of the models are quite similar to each other. Remember that GRPb is the one of the networks with the highest deviation on the sizes of the clusters, which implies more differences between clusters and thus more differences between nodes. On the other hand, GRPd is the only undirected model which could explain the low values for the standard deviation and the number of different ranks.

In the LFR models (see Figure 2), there is a clearly one model with the lowest values in every scenario and for the majority of measures, specially the influence-based ones: LFRd. These low results are caused mainly by the exponents used during the generation, being in this models the highest, especially the exponent for the degree distribution. There does not seem to be a model with clearly higher results, although LFRb does get higher values in some cases.

For the stochastic block model generator (see Figure 3), STOa, STOc, STOd have the highest values for every metric except Betweenness and Pagerank, for the standard deviation, the number of different ranks and the Gini coefficient. The models with the lowest values are STOb and STOe. Similarly to GRP, these differences occur due to the sizes of the clusters, in STOb the sizes follow a normal distribution instead of a exponential distribution, this will result in more similar clusters, so more similar nodes. In the case of STOe, although the sizes of the clusters follow an exponential distribution, the probability of creating edges inside the cluster depends on the size of the clusters, which means a node who belongs to a big cluster will have a small probability, this will imply that the number of edges will be close to a node that belongs to a small cluster but has a large probability of creating edges inside of the cluster. This phenomena balances the degree distribution of the nodes to some extent. This events can be observed specially well in the models with 100K nodes.

Finally, the hyperbolic models (see Figure 4) do not seem easy to analyse. The standard deviation is low for HYPc, HYPd and HYPe for every metric except Betweeness but there are more different ranks in these three models than in HYPa or HYPb, again with some

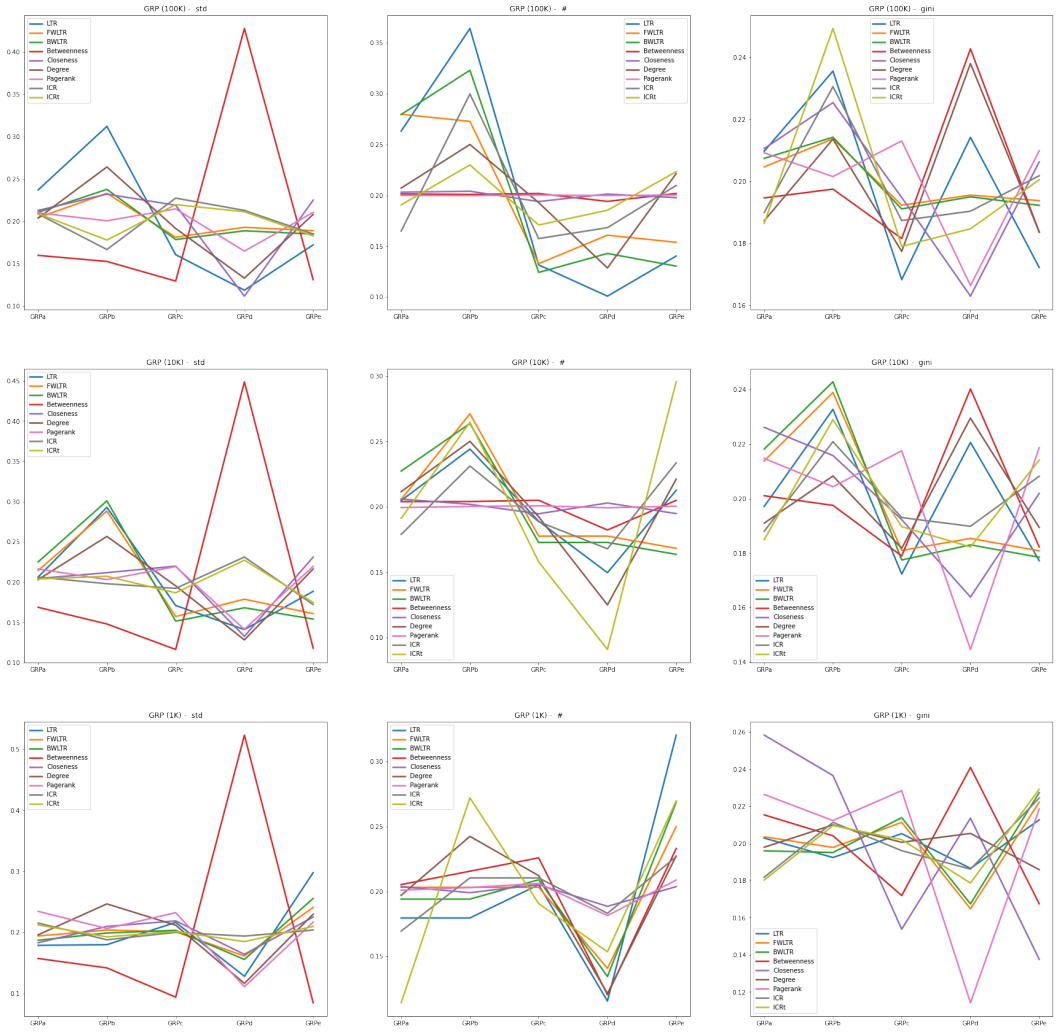


Figure 1 Statistics results for the GRP models.

exceptions like the Degree Centrality and the three centralities based on the LTM. The Gini coefficient is high in HYPa and HYPb for some metrics, but for other metrics the results in HYPc, HYPd and HYPe are higher. The only irrefutable conclusion from this network generator is that there is a clear distinction between the results for HYPa, HYPb and those for HYPc, HYPd, HYPe.

5.2 Correlation analysis

Figure 5 collects the correlation plots for the set of biggest networks (i.e., those with 100K nodes) for each of the four synthetic models under study. Figure 6 shows the correlation plots for four real social networks that represent each of the four behavioural tendencies observed. The correlation plots do not include the results of the Ktz centrality because most of the time the algorithm did not converge.

For the Gaussian graphs there is one network very different than the others, GRPd, but this happens because it is the only undirected social network, which creates the difference in the correlation patterns, having a maximum direct correlation between the three LTM

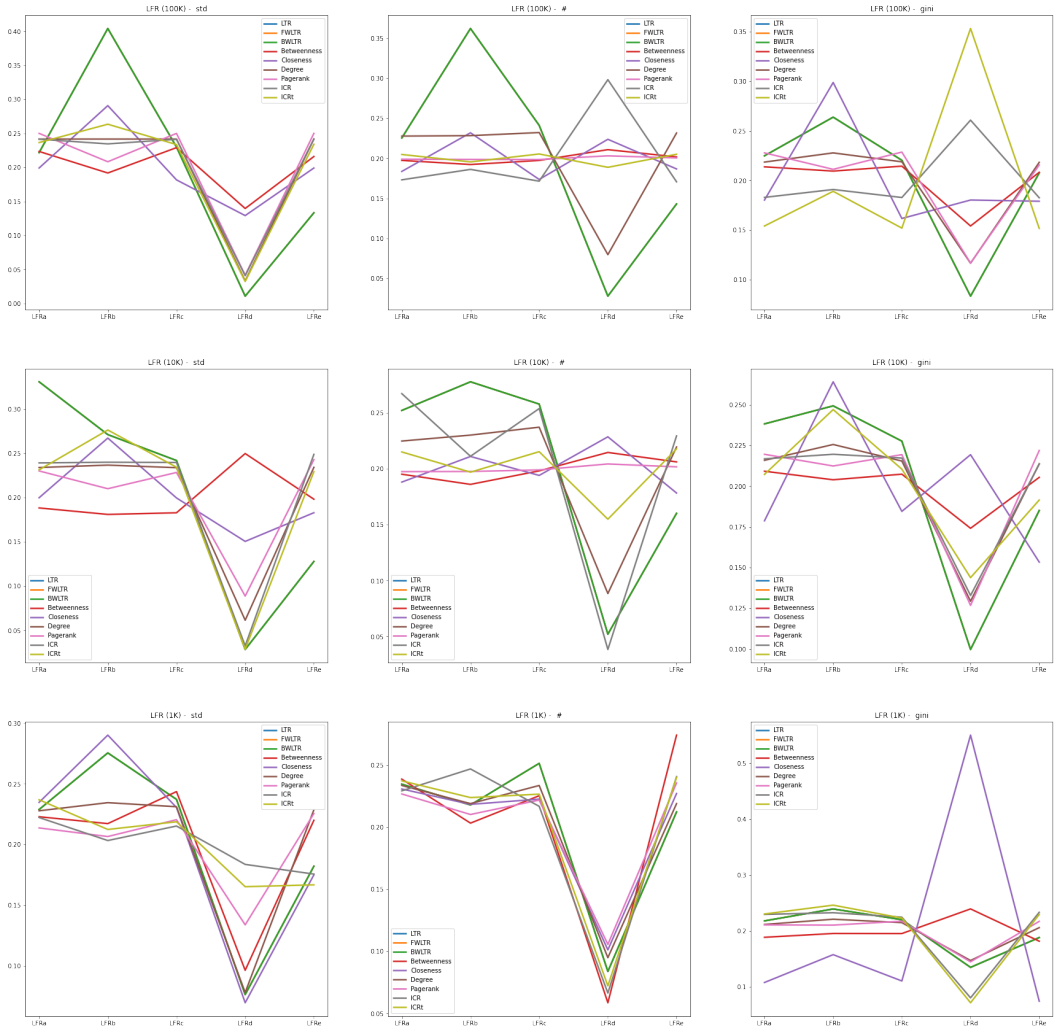


Figure 2 Statistics results for the LFR models.

based centralities. Comparing the rest of the networks we observe very similar patterns in the correlation plots, but they still can be distinguished, having more similarities between GRPa and GRPb, and between GRPc and GRPe. This differences can be seen more clearly as we decrease the number of nodes in the graph.

In the case of the LFR benchmark generator, we take into consideration that all the networks are undirected which implies that the results from LTR, FWLTR and BWLTR will be the same. The first three networks (LTRa, LTRb, LTRc) display similar patters, with low correlation between Betweenness and the LTM based metrics and high correlations between Pagerank and Degrees. One the other hand, LTRd differentiates in some aspects from these three mentioned networks, such as the high correlation between Pagerank and the LTM metrics and between Degree and the LTM metrics. Another difference is that in LFRd the correlation between Closeness and the LTM metrics is lower than the correlation between Betweenness and the LTM metrics. Finally, the LFRe network is pretty similar to the first three networks (LTRa, LTRb, LTRc) but with very subtle differences.

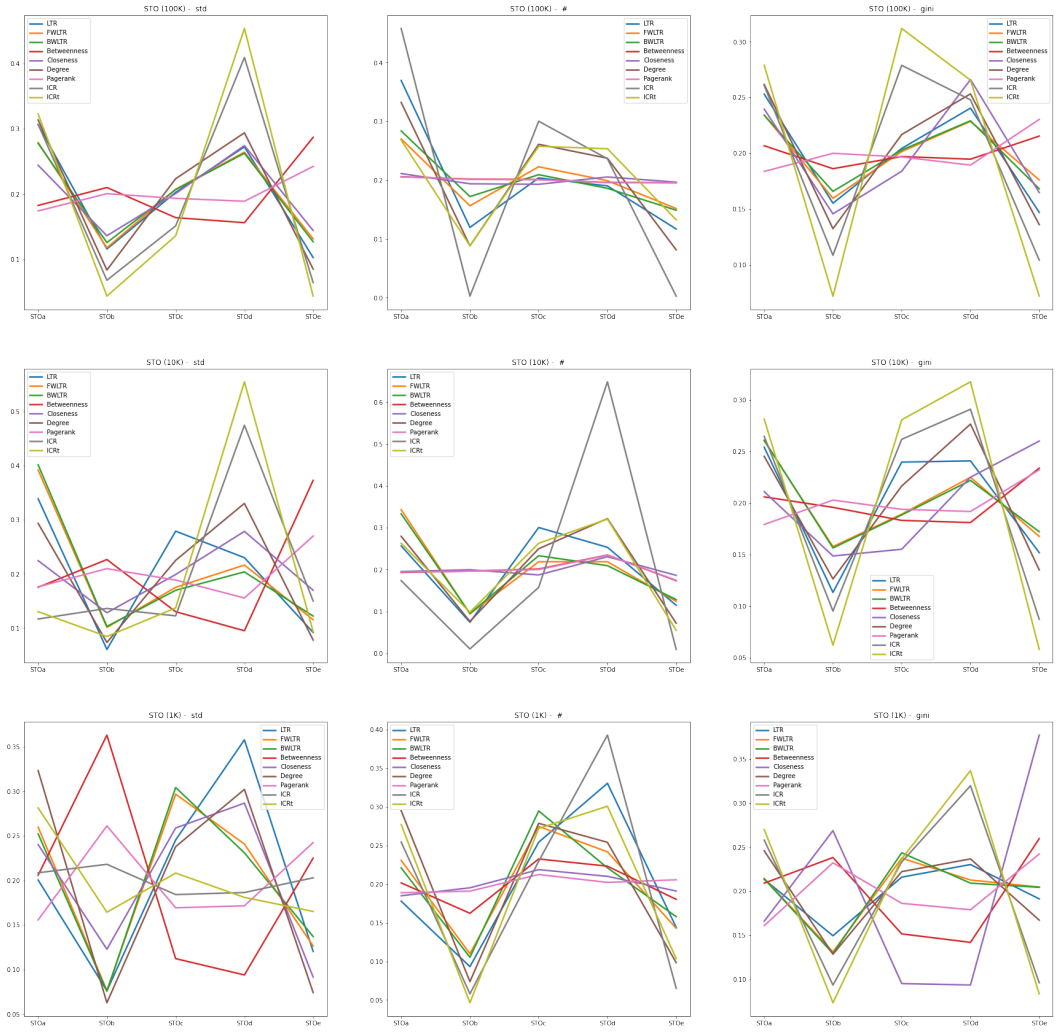


Figure 3 Statistics results for the STO models.

We can observe more differences when comparing the results from the Stochastic Block Model generator. The STOb and STOe are nearly identical, STOc has a similar pattern as the these last two, but with higher correlations between all metrics. For the other two graphs, STOa and STOd, we find most of the similarities when for big graphs with a large number of nodes, however when we compare both models with only one thousand nodes, the similarities in the patterns in the correlation plots seem to disappear, for example, STOa has very high correlations between all the LTM based metrics (LTR, FWLTR, BWLTR) but STOd does not, with very low correlations between FWLTR and BWLTR.

Finally, the networks generated in a hyperbolic space show two patterns. The first type, present in HYPa and HYPb, with high correlation between Closeness and the LTM metrics, and low correlation between the LTM metrics and almost any other metric where Betweenness and Pagerank take the lowest values, this also implies a low correlation between Closeness and, Betweenness and Pagerank. However, the other type, including HYPc, HYPd and HYPe, Closeness takes the lowest correlation with the LTM metrics. HYPc is a little different than HYPd and HYPe but the general distribution of the correlation is still the same.

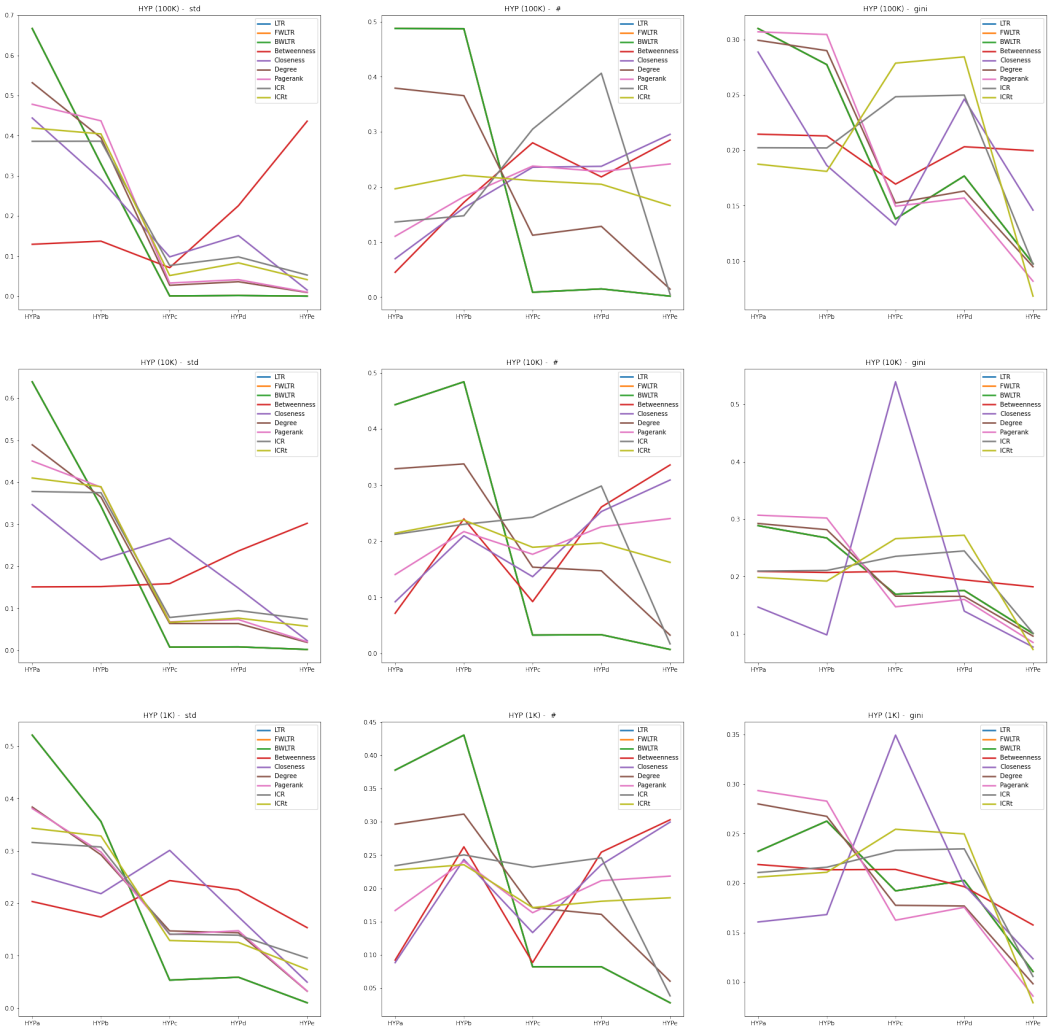


Figure 4 Statistics results for the HYP models.

5.3 Comparison with real networks

In order to extend the study of centrality measures on synthetic networks, we decided to focus on real social networks. Our aim with that was to check whether the synthetic models do really approximate real networks when centrality is concerned. We also wanted to check whether the different behavioural patterns observed in the synthetic networks would help us to classify the real networks.

Some correlation plots from real graphs look almost identical to correlation plots of synthetic networks (e.g., the Texas graph and the HYPe, the Caida graph and the LFRa). The Human Brain network is similar to most of the LFR benchmark models, but it is with a hyperbolic graph where more similarities can be found (specifically with HYPa). There are also examples of directed graphs where this also occurs, e.g., Epinions, which is pretty similar to GRPa. In most of the real social networks considered, we can observe some kind of similarity to some type of artificial network. Based on those similarities, we organize our results in four categories (two for directed graphs and two for undirected graphs). These categories are qualitative and based merely on correlations, thus describing distinguishable color patterns in the plot of the Figure 6.

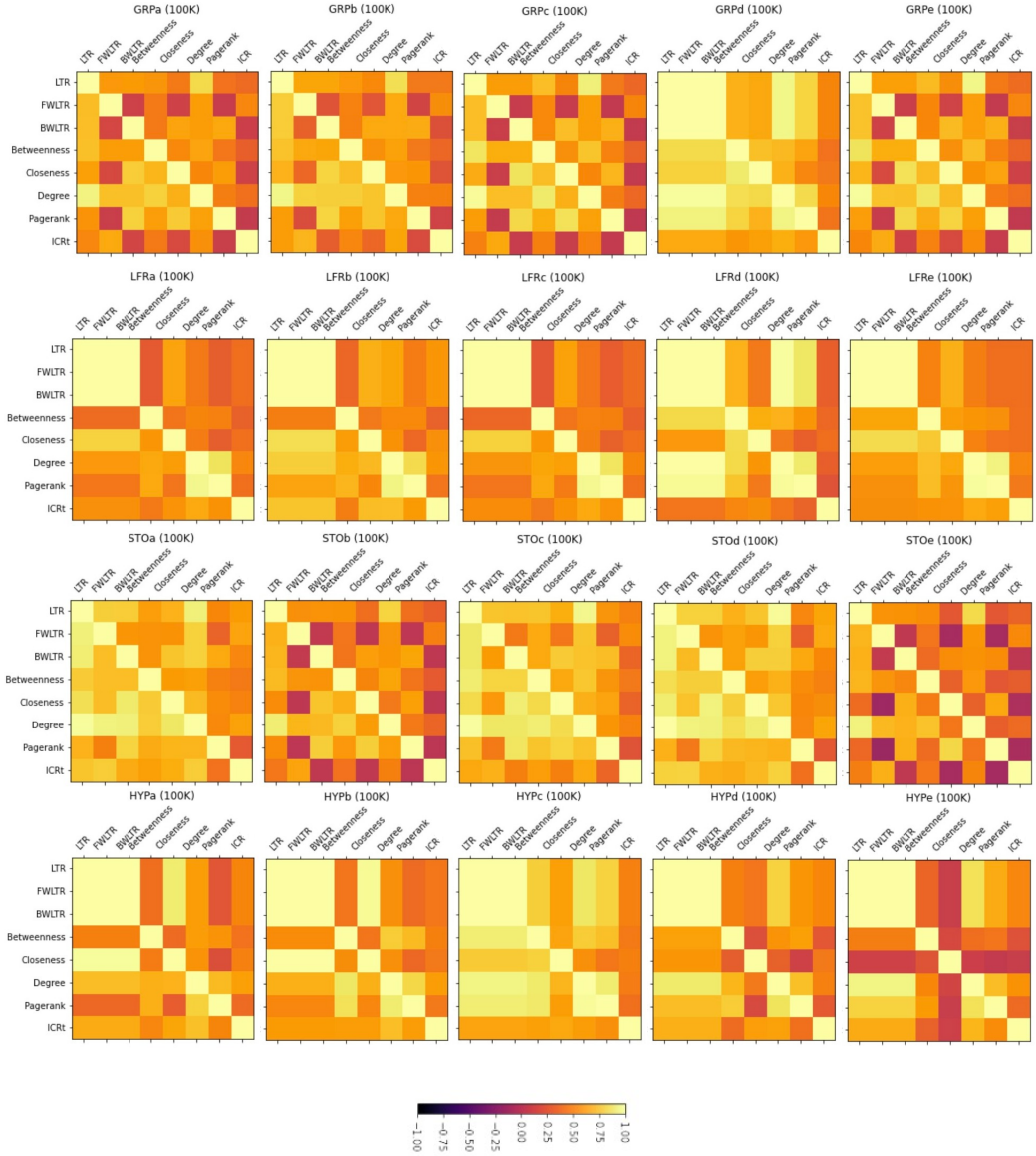


Figure 5 Heatmaps for the correlation between measures for synthetic networks with 100K nodes. Kendall coefficients are represented in the upper triangular part and Spearman in the lower one.

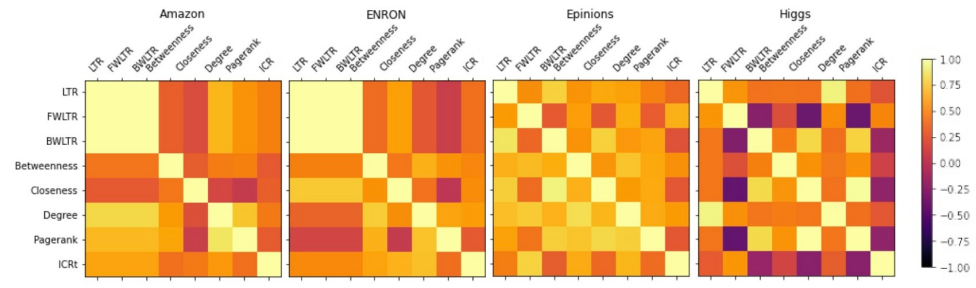


Figure 6 Heatmaps for the correlation between measures for four real social networks (Amazon, ENRON, Epinions, Higgs), which represent the four different behaviours observed experimentally. Kendall coefficients are represented in the upper triangular part and Spearman in the lower one.

The first type is for undirected graphs and includes the networks Amazon, Dolphins, Texas, and ArXiv. The synthetic networks that represent this first category are GRPd, LFRd, HYPc, HYPd, HYPe. They have low correlation between Betweenness and the LTM metrics and a very low correlation between Closeness and the LTM metrics. We find higher correlations between Degree and the LTM metrics, Pagerank and the LTM metrics, and Degree and Pagerank.

The second category is also for undirected graphs and it includes the real networks Caida, ENRON and Human Brain, and the synthetic networks LFRa, LFRb, LFRc, LFRe, HYPa and HYPb. One of the main differences between this and the previous category is the change in the correlations of Closeness and Degree: here the correlation between Degree and the LTM is high, and the correlation between Closeness and the LTM metrics is one of the highest, opposite to the first category.

We observe a third category for directed graphs, which includes the Epinions real graph and GRPa, GRPb and STOc. We found the lowest correlations when looking at FWLTR and ICRt. These two metrics have a low correlation with BWLTR, Closeness and Pagerank. The rest of the correlations are neither high nor low.

The last category that we can distinguish is also for directed graphs. In this case we have the Higgs and Wikipedia as real networks representatives, and GRPc, GRPe as synthetic graphs. The main different with the third category is that this time the low correlations of FWLTR and ICRt are much lower, with values very close to zero. In the rest of the correlations we can find low and medium correlations unlike in the last category where most of them were medium correlations.

The synthetic graphs STOb and STOe are halfway between the third and fourth category, having similarities and differences with both of them.

There are two real social networks who do not seem to belong to any of the categories mentioned, which means that they are not very similar with the synthetic networks generated in terms of the centrality measures correlation. The first example, the Dining Table network, can be easily explained. Most of the correlations given by this network have a p-value higher than 0.05 which makes most of the results not statistically significant. However, in the remaining network Gnutella, this phenomena does not occur which means than the results are valid and significant. In this case, the problem could be that the number of generators and models used is limited and does not cover all the possible networks. Another cause could be that the structure of this network is very particular and it is hard to replicate artificially with algorithms.

All the comparisons between correlations are qualitative in this work. We plan to introduce quantitative measures to be able to weight those relation, e.g. by means of similarity measures applied to the correlation matrices.

References

1 M.J. Blesa, P. García-Rodríguez, and M.Serna. Forward and backward linear threshold ranks. In *International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 265–269. ACM, 2021. doi:10.1145/3487351.3488355.

2 L. Ceriani and P. Verme. The origins of the Gini index: extracts from Variabilità e Mutabilità (1912) by Corrado Gini. *The Journal of Economic Inequality*, 10(3):421–443, 2012.

3 Hocine Cherifi, Gergely Palla, Boleslaw K. Szymanski, and Xiaoyan Lu. On community structure in complex networks: challenges and opportunities. *Applied Network Science*, 4(117):2364–8228, 2019. doi:10.1007/s41109-019-0238-9.

4 X. Molinero F. Riquelme, P. Gonzalez-Cantergiani and M. Serna. Centrality measures in social networks based on linear threshold model. *Knowledge-Based Systems*, 40:92–102, 2017. doi:10.1016/j.knosys.2017.10.029.

5 L.C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

6 C. Gini. *Variabilitàe Mutuabilità. Contributo allo Studio delle Distribuzioni e delle Relazioni Statistiche*. C. Cuppini, 1912.

7 J. Goldenberg, B. Libai, and E. Muller. Using complex systems analysis to advance marketing theory development. *Technical Report, Academy of Marketing Science Review*, 2001.

8 L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18:39–43, 1953.

9 D. Kempe, J.M. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. *Proceedings of the 9th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 137–146, 2003.

10 D. Kempe, J.M. Kleinberg, and É. Tardos. Influential nodes in a diffusion model for social networks. In *Intl. Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580, pages 1127–1138. Lecture Notes in Computer Science, 2005. doi:10.1007/11523468_91.

11 M. Kendall. A new measure of rank correlation. *Biometrika*, 30:81–93, 1938.

12 D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguñá. Hyperbolic geometry of complex networks. *Phys. Rev. E Stat. Nonlin. Soft Matter Phys.*, 82((3 Pt 2):036106), 2010. doi:10.1103/PhysRevE.82.036106.

13 A. Lancichinetti, S. Fortunato, and F. Radicchi. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E*, 78:046110, October 2008. doi:10.1103/PhysRevE.78.046110.

14 NetworkX. Gaussian Random Partition Graph. Accessed: 2022-02. URL: https://networkx.org/documentation/stable/reference/generated/networkx.generators.community.gaussian_random_partition_graph.html.

15 NetworkX. LFR Benchmark Graph. Accessed: 2022-02. URL: https://networkx.org/documentation/stable/reference/generated/networkx.generators.community.LFR_benchmark_graph.html.

16 NetworkX. Stochastic Block Model. Accessed: 2022-02. URL: https://networkx.org/documentation/stable/reference/generated/networkx.generators.community.stochastic_block_model.html.

17 Mark E.J. Newman. *Networks: An introduction*. Oxford University Press, 2010. doi:10.1093/acprof:oso/9780199206650.001.0001.

18 S. Oldham, B. Fulcher, L. Parkes, A. Arnatkevičiūté, C. Suo, and A. Fornito. Consistencies and differences between centrality measures across distinct classes of networks. *PLoS ONE*, 14(7):0220061, 2019. doi:10.1371/journal.pone.0220061.

19 L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, technical report. *Stanford Digital Library*, 1999.

20 K. Blackmond Laskey P.W. Holland and S. Leinhardt. Stochastic blockmodels: First steps. *Social Networks*, 5(2):109–137, 1983. doi:10.1016/0378-8733(83)90021-7.

21 C. Orsini R. Aldecoa and D. Krioukov. Hyperbolic graph generator. *Computer Physics Communications*, 196:492–496, 2015. doi:10.1016/j.cpc.2015.05.028.

22 C. Sciarra, G. Chiarotti, F. Laio, and L. Ridolfi. A change of perspective in network centrality. *Scientific Reports*, 2018. doi:10.1038/s41598-018-33336-8.

23 P. Shakarian, A. Bhatnagar, A. Aleali, E. Shaabani, and R. Guo. *The Independent Cascade and Linear Threshold Models*, chapter 4, pages 35–48. Briefs in Computer Science. Springer, 2015. doi:10.1007/978-3-319-23105-1_4.

24 C. Spearman. The proof and measurement of association between two things. *AM. J. Psychol*, 15:88–103, 1904.

25 D. Wagner U. Brandes, M. Gaertler. Experiments on graph clustering algorithms. In *Algorithms - ESA 2003*, pages 568–579. Springer Berlin Heidelberg, 2003.

26 T.W. Valente, K. Corognes, C. Lakon, and E. Costenbader. How correlated are network centrality measures? *Connections*, 28(1):16–26, 2008.

27 S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

A Details on the correlation analysis (Section 5.2, Fig. 5)

We detail the correlation coefficients for the centrality measures summarized in Figure 5. In all the forthcoming tables, the Kendall coefficients (τ) are shown in the upper triangular part and the Spearman coefficients (ρ) in the lower triangular part. For the `Katz` measure, a $-$ indicates non-convergence.

Table 6 Correlation for the different centrality measures on the synthetic GRP networks of size 100K from the data set.

GRPa									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	0,5504	0,5569	0,5385	0,4199	0,8184	0,5200	0,3807	0,3361
FWLTR	0,6971	1	0,0987	0,4204	0,1049	0,5684	0,1379	0,0556	0,4694
BWLTR	0,7051	0,1335	1	0,4185	0,5962	0,5672	0,7634	0,6151	0,0882
Betweenness	0,7125	0,5655	0,5642	1	0,4899	0,6015	0,4401	0,4045	0,2871
Closeness	0,5782	0,1490	0,7608	0,6734	1	0,4775	0,7082	0,5674	0,1046
Degree	0,9228	0,7139	0,7135	0,7757	0,6442	1	0,5910	0,4314	0,3685
Katz	0,6943	0,1954	0,8995	0,6153	0,8868	0,7653	1	0,6170	0,1337
Pagerank	0,5286	0,0791	0,7757	0,5707	0,7573	0,5882	0,8040	1	0,0510
ICRt	0,4714	0,6248	0,1257	0,4174	0,1564	0,5106	0,1992	0,0764	1

GRPb									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	0,6088	0,6057	0,5473	0,4894	0,8402	-	0,4116	0,4075
FWLTR	0,7631	1	0,2149	0,4548	0,2271	0,6248	-	0,1240	0,5059
BWLTR	0,7609	0,2917	1	0,4505	0,6248	0,6176	-	0,6326	0,1957
Betweenness	0,7251	0,6105	0,6059	1	0,5109	0,5889	-	0,4138	0,3272
Closeness	0,6633	0,3213	0,7914	0,6988	1	0,5478	-	0,5411	0,2207
Degree	0,9398	0,7767	0,7707	0,7663	0,7266	1	-	0,4473	0,4453
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,5687	0,1774	0,7945	0,5826	0,7291	0,6107	-	1	0,1118
ICRt	0,5646	0,6691	0,2775	0,4717	0,3255	0,6078	-	0,1669	1

GRPc									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1,0000	0,5664	0,5656	0,7066	0,4633	0,9123	0,5180	0,4118	0,3212
FWLTR	0,7090	1,0000	0,0584	0,4872	0,0572	0,5639	0,0658	0,0371	0,4628
BWLTR	0,7081	0,0785	1,0000	0,4852	0,7039	0,5630	0,8367	0,6444	0,0423
Betweenness	0,8661	0,6397	0,6379	1,0000	0,5120	0,7322	0,5052	0,4609	0,3069
Closeness	0,6273	0,0812	0,8566	0,6954	1,0000	0,4805	0,8177	0,6689	0,0447
Degree	0,9706	0,7061	0,7051	0,8851	0,6464	1,0000	0,5410	0,4334	0,3310
Katz	0,6886	0,0934	0,9457	0,6885	0,9546	0,7125	1,0000	0,6833	0,0505
Pagerank	0,5647	0,0528	0,8022	0,6382	0,8515	0,5901	0,8621	1,0000	0,0272
ICRt	0,4487	0,6154	0,0600	0,4437	0,0669	0,4609	0,0755	0,0408	1,0000

GRPd									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	1	1	0,6530	0,6144	0,9391	-	0,7964	0,4652
FWLTR	1	1	1	0,6530	0,6144	0,9391	-	0,7964	0,4652
BWLTR	1	1	1	0,6530	0,6144	0,9391	-	0,7964	0,4652
Betweenness	0,8131	0,8131	0,8131	1	0,7164	0,6812	-	0,5996	0,3806
Closeness	0,7791	0,7791	0,7791	0,8904	1	0,6507	-	0,4734	0,4089
Degree	0,9767	0,9767	0,9767	0,8355	0,8112	1	-	0,8248	0,4816
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,9204	0,9204	0,9204	0,7949	0,6578	0,9356	-	1	0,3856
ICRt	0,6204	0,6204	0,6204	0,5423	0,5805	0,6368	-	0,5502	1

Table 7 Correlation for the different centrality measures on the synthetic LFR networks of size 100K from the data set.

LFRa									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	1	1	0,2514	0,5987	0,4142	-	0,2761	0,3573
FWLTR	1	1	1	0,2514	0,5987	0,4142	-	0,2761	0,3573
BWLTR	1	1	1	0,2514	0,5987	0,4142	-	0,2761	0,3573
Betweenness	0,3475	0,3475	0,3475	1	0,3881	0,4660	-	0,4483	0,2909
Closeness	0,7887	0,7887	0,7887	0,5414	1	0,4231	-	0,2739	0,3695
Degree	0,5541	0,5541	0,5541	0,6162	0,5650	1	-	0,8558	0,4711
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,4006	0,4006	0,4006	0,6211	0,3969	0,9513	-	1	0,3673
ICRt	0,5103	0,5103	0,5103	0,4200	0,5250	0,6245	-	0,5223	1

LFRb									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	1	1	0,3212	0,6461	0,6059	-	0,4276	0,5392
FWLTR	1	1	1	0,3212	0,6461	0,6059	-	0,4276	0,5392
BWLTR	1	1	1	0,3212	0,6461	0,6059	-	0,4276	0,5392
Betweenness	0,4371	0,4371	0,4371	1	0,4125	0,4852	-	0,4911	0,2999
Closeness	0,8310	0,8310	0,8310	0,5639	1	0,5194	-	0,3302	0,5080
Degree	0,7708	0,7708	0,7708	0,6360	0,6823	1	-	0,8205	0,5399
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,6007	0,6007	0,6007	0,6708	0,4781	0,9346	-	1	0,3934
ICRt	0,7313	0,7313	0,7313	0,4299	0,6970	0,7072	-	0,5576	1

LFRc									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	1	1	0,2362	0,5899	0,4007	-	0,2704	0,3440
FWLTR	1	1	1	0,2362	0,5899	0,4007	-	0,2704	0,3440
BWLTR	1	1	1	0,2362	0,5899	0,4007	-	0,2704	0,3440
Betweenness	0,3244	0,3244	0,3244	1	0,3893	0,4570	-	0,4397	0,2852
Closeness	0,7793	0,7793	0,7793	0,5465	1	0,4347	-	0,2962	0,3624
Degree	0,5389	0,5389	0,5389	0,6060	0,5790	1	-	0,8644	0,4682
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,3937	0,3937	0,3937	0,6115	0,4271	0,9571	-	1	0,3732
ICRt	0,4934	0,4934	0,4934	0,4126	0,5162	0,6212	-	0,5304	1

LFRd									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	1	1	0,6590	0,4128	0,9887	-	0,8734	0,2870
FWLTR	1	1	1	0,6590	0,4128	0,9887	-	0,8734	0,2870
BWLTR	1	1	1	0,6590	0,4128	0,9887	-	0,8734	0,2870
Betweenness	0,8076	0,8076	0,8076	1	0,6365	0,6616	-	0,5454	0,3300
Closeness	0,5506	0,5506	0,5506	0,8257	1	0,4094	-	0,2862	0,3536
Degree	0,9954	0,9954	0,9954	0,8098	0,5459	1	-	0,8875	0,2836
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,9616	0,9616	0,9616	0,7289	0,4135	0,9700	-	1	0,2139
ICRt	0,3940	0,3940	0,3940	0,4751	0,5029	0,3891	-	0,3120	1

Table 8 Correlation for the different centrality measures on the synthetic STO networks of size 100K from the data set.

STOa									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	0,7657	0,7628	0,5716	0,6591	0,9080	-	0,4816	0,5613
FWLTR	0,9074	1	0,5423	0,5310	0,5206	0,7807	-	0,3081	0,5954
BWLTR	0,9055	0,7070	1	0,5295	0,7270	0,7743	-	0,6175	0,4697
Betweenness	0,7554	0,7074	0,7057	1	0,5592	0,5797	-	0,4508	0,4155
Closeness	0,8374	0,6982	0,8849	0,7522	1	0,6940	-	0,4939	0,4592
Degree	0,9769	0,9171	0,9127	0,7624	0,8673	1	-	0,4874	0,5897
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,6572	0,4380	0,7894	0,6291	0,6795	0,6643	-	1	0,2558
ICRt	0,7332	0,7609	0,6326	0,5848	0,6333	0,7600	-	0,3751	1

STOb									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	0,5155	0,5248	0,5183	0,3539	0,7955	0,4616	0,3545	0,2871
FWLTR	0,6558	1	0,0324	0,3959	0,0163	0,5361	0,0325	0,0082	0,4472
BWLTR	0,6667	0,0437	1	0,3876	0,5706	0,5397	0,7924	0,6075	0,0219
Betweenness	0,6888	0,5341	0,5251	1	0,4637	0,6024	0,4124	0,3861	0,2573
Closeness	0,4937	0,0232	0,7332	0,6425	1	0,4131	0,6761	0,5815	0,0204
Degree	0,9048	0,6758	0,6805	0,7739	0,5645	1	0,5297	0,4191	0,3162
Katz	0,6261	0,0463	0,9181	0,5809	0,8604	0,6989	1	0,6660	0,0316
Pagerank	0,4942	0,0116	0,7682	0,5477	0,7719	0,5710	0,8475	1	0,0049
ICRt	0,4049	0,5974	0,0311	0,3756	0,0306	0,4403	0,0473	0,0074	1

STOc									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	0,7269	0,7263	0,7334	0,6384	0,9499	-	0,5323	0,4527
FWLTR	0,8733	1	0,4191	0,6181	0,4000	0,7270	-	0,2926	0,4975
BWLTR	0,8731	0,5586	1	0,6185	0,7524	0,7264	-	0,6999	0,3161
Betweenness	0,8962	0,7923	0,7926	1	0,6139	0,7385	-	0,5372	0,4079
Closeness	0,8163	0,5507	0,8997	0,8063	1	0,6472	-	0,6002	0,3081
Degree	0,9906	0,8734	0,8732	0,8995	0,8241	1	-	0,5406	0,4585
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,7153	0,4151	0,8604	0,7283	0,7927	0,7246	-	1	0,2152
ICRt	0,6103	0,6536	0,4390	0,5736	0,4408	0,6166	-	0,3174	1

STOd									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	0,7804	0,7778	0,6526	0,7271	0,9695	-	0,4735	0,6019
FWLTR	0,9169	1	0,5273	0,5893	0,5454	0,7764	-	0,2877	0,6185
BWLTR	0,9149	0,6908	1	0,5873	0,7723	0,7734	-	0,6139	0,4801
Betweenness	0,8342	0,7697	0,7673	1	0,5720	0,6519	-	0,4984	0,4514
Closeness	0,8909	0,7250	0,9162	0,7682	1	0,7355	-	0,4813	0,5051
Degree	0,9955	0,9141	0,9119	0,8331	0,8968	1	-	0,4755	0,6071
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,6491	0,4102	0,7859	0,6831	0,6661	0,6511	-	1	0,2500
ICRt	0,7751	0,7846	0,6471	0,6310	0,6864	0,7797	-	0,3674	1

Table 9 Correlation for the different centrality measures on the synthetic HYP networks of size 100K from the data set.

HYPa									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	1	1	0,3516	0,8978	0,5677	-	0,2327	0,4664
FWLTR	1	1	1	0,3516	0,8978	0,5677	-	0,2327	0,4664
BWLTR	1	1	1	0,3516	0,8978	0,5677	-	0,2327	0,4664
Betweenness	0,4394	0,4394	0,4394	1	0,3377	0,5672	-	0,5245	0,3816
Closeness	0,9806	0,9806	0,9806	0,4222	1	0,5300	-	0,1856	0,4476
Degree	0,7015	0,7015	0,7015	0,6464	0,6621	1	-	0,6918	0,5833
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,3285	0,3285	0,3285	0,6465	0,2690	0,7802	-	1	0,3402
ICRt	0,6298	0,6298	0,6298	0,4741	0,6093	0,7160	-	0,4651	1

HYPb									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	1	1	0,3957	0,9618	0,5102	-	0,3386	0,4049
FWLTR	1	1	1	0,3957	0,9618	0,5102	-	0,3386	0,4049
BWLTR	1	1	1	0,3957	0,9618	0,5102	-	0,3386	0,4049
Betweenness	0,4999	0,4999	0,4999	1	0,3925	0,7631	-	0,6952	0,4380
Closeness	0,9963	0,9963	0,9963	0,5004	1	0,5113	-	0,3290	0,4099
Degree	0,6424	0,6424	0,6424	0,8707	0,6483	1	-	0,8014	0,5697
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,4734	0,4734	0,4734	0,8456	0,4689	0,8943	-	1	0,4135
ICRt	0,5575	0,5575	0,5575	0,5696	0,5671	0,7036	-	0,5675	1

HYPc									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	1	1	0,7696	0,5854	0,8678	-	0,7839	0,4477
FWLTR	1	1	1	0,7696	0,5854	0,8678	-	0,7839	0,4477
BWLTR	1	1	1	0,7696	0,5854	0,8678	-	0,7839	0,4477
Betweenness	0,9005	0,9005	0,9005	1	0,5989	0,7588	-	0,7111	0,4204
Closeness	0,7523	0,7523	0,7523	0,7789	1	0,5138	-	0,3994	0,4409
Degree	0,9358	0,9358	0,9358	0,8895	0,6668	1	-	0,8953	0,4276
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,9043	0,9043	0,9043	0,8817	0,5648	0,9704	-	1	0,3605
ICRt	0,5988	0,5988	0,5988	0,5853	0,6128	0,5692	-	0,5090	1

HYPd									
p \ t	LTR	FWLTR	BWLTR	Betweenness	Closeness	Degree	Katz	Pagerank	ICRt
LTR	1	1	1	0,4444	0,3961	0,7839	-	0,5402	0,4800
FWLTR	1	1	1	0,4444	0,3961	0,7839	-	0,5402	0,4800
BWLTR	1	1	1	0,4444	0,3961	0,7839	-	0,5402	0,4800
Betweenness	0,5921	0,5921	0,5921	1	0,1939	0,5260	-	0,5842	0,2417
Closeness	0,5447	0,5447	0,5447	0,2787	1	0,2962	-	0,1065	0,4101
Degree	0,8890	0,8890	0,8890	0,6754	0,4082	1	-	0,7186	0,4474
Katz	-	-	-	-	-	-	-	-	-
Pagerank	0,6992	0,6992	0,6992	0,7652	0,1587	0,8549	-	1	0,2560
ICRt	0,6415	0,6415	0,6415	0,3468	0,5713	0,5931	-	0,3673	1

Efficient and Accurate Group Testing via Belief Propagation: An Empirical Study

Amin Coja-Oghlan
Faculty of Computer Science, TU Dortmund, Germany

Max Hahn-Klimroth
Faculty of Computer Science, TU Dortmund, Germany

Philipp Loick
Institute for Mathematics, Goethe Universität, Frankfurt am Main, Germany

Manuel Penschuck
Faculty of Computer Science, Goethe Universität, Frankfurt am Main, Germany

Abstract

The group testing problem asks for efficient pooling schemes and inference algorithms that allow to screen moderately large numbers of samples for rare infections. The goal is to accurately identify the infected individuals while minimizing the number of tests.

We propose the novel adaptive pooling scheme *adaptive Belief Propagation* (ABP) that acknowledges practical limitations such as limited pooling sizes and noisy tests that may give imperfect answers. We demonstrate that the accuracy of ABP surpasses that of individual testing despite using few overall tests. The new design comes with Belief Propagation as an efficient inference algorithm. While the development of ABP is guided by mathematical analyses and asymptotic insights, we conduct an experimental study to obtain results on practical population sizes.

2012 ACM Subject Classification Mathematics of computing → Probabilistic inference problems; Mathematics of computing → Random graphs; Mathematics of computing → Coding theory

Keywords and phrases Group testing, Probabilistic Construction, Belief Propagation, Simulation

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.8

Related Version *Previous Version*: `https://arxiv.org/abs/2105.07882`

Supplementary Material *Software (Source Code)*: `https://github.com/manpen/group-testing`

Funding This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG) under DFG CO 646/3, DFG CO 646/5, and DFG ME 2088/5-1 (FOR 2975 – Algorithms, Dynamics, and Information Flow in Networks).

1 Introduction

Every day medical laboratories around the globe screen moderately large numbers of samples for rare pathogens. The vast majority of samples, anywhere between 90% and 99.9%, are actually uninfected [9, 25, 28, 40, 32, 37, 38, 39, 42]. Labs therefore test pools of samples rather than individual samples. The *group testing problem* asks for pooling strategies that minimise the total number of tests required while maximising the accuracy of the results. The latter is crucial because test results are generally not perfectly accurate.

Practical solutions are challenging precisely because the number of samples in a real-world scenario is in the hundreds or thousands. While the group testing problem has inspired a body of mathematical work for the asymptotical scenario [5, 13, 12], these results, where the number of samples grows to infinity, do not directly apply to practical problem sizes. They

also tend to construct excessively large test pools or distribute samples in very many tests [5, 13, 12]. Yet practical problem sizes are too large to exhaustively search for an optimal test. Thus, pooling schemes from the 1940s remain in practical use [9, 25, 28, 40, 32, 37, 38, 39, 42].

The aim of this paper is to investigate better test designs for practical problem sizes. We focus on improving the *accuracy* of the results, i.e., avoiding false positives and/or negatives while keeping the number of tests as small as possible. Indeed, group testing, originally invented to reduce the number of tests, actually excels at improving the accuracy of the results. This may seem surprising at first glance because one might deem individual testing optimal in terms of accuracy. It is not. Group testing does better in much the same way as error-correcting codes gain power from encoding entire blocks of data simultaneously.

1.1 Our contributions and outline

Given the moderate number of samples in real-world scenarios, we obtain practically meaningful results by conducting an extensive experimental study based on theoretical work on group testing as well as recent ideas from information theory and statistical physics. Our novel test design ABP improves the accuracy of the overall results while keeping the number of tests conducted low. Furthermore, the new test design requires only relatively small test pools and only assigns each sample to a small number of tests. Finally, the design comes with an efficient, easy-to-implement algorithm to infer the status of the individual samples from the test results, namely the Belief Propagation (BP) message passing algorithm.

We proceed to discuss the mathematical model we work with in Sec. 1.2. In Sec. 2, we discuss designs and algorithms that are in practical use or have been studied in the mathematical literature on group testing. In Sec. 3, we present the details behind our novel test design ABP and relate ABP to the theoretical work on group testing and asymptotic considerations in Sec. 5. Finally, in Sec. 6 we discuss the potential impact of the new results and future directions for both empirical and theoretical work.

1.2 The model

We work with a simple but standard model of group testing that allows for inaccurate test results [5]. Let $x_1, \dots, x_n$ be the samples to be tested and let $\lambda \in [0, 1]$ be the prior probability that any one sample is infected. The true infection status of each sample is indicated by $\boldsymbol{\sigma}(x_j) \in \{0, 1\}$, with 1 representing "infected". The $\boldsymbol{\sigma}(x_j)$ are assumed to be independent Bernoulli variables with mean λ. We refer to the vector $\boldsymbol{\sigma} = (\boldsymbol{\sigma}(x_j))_{j=1,\dots,n}$ as the *ground truth*. Let $k = \sum_{j=1}^{n} \mathbf{1}\{\boldsymbol{\sigma}(x_j) = 1\}$ signify the actual number of infected samples.

A *test design* is a bipartite graph G with one class $\mathcal{X} = \{x_1, \dots, x_n\}$ representing the n samples and the other class $\mathcal{A} = \{a_1, \dots, a_m\}$ representing the test pools. An edge between $\{x_j, a_i\}$ indicates that x_j is included in test pool a_i. For each x_j we let $\partial x_j = \partial_G x_j$ be the set of test pools that include x_j; analogously, ∂a_i denotes the samples x_j in pool a_i.

Let $\hat{\boldsymbol{\sigma}} = (\hat{\boldsymbol{\sigma}}(a_i))_{i=1,\dots,m}$ denote the test results. Ideally, test a_i should report positive iff at least one sample $x_j \in \partial a_i$ is infected. But the actual result $\hat{\boldsymbol{\sigma}}(a_i)$ may include independent noise controlled via the *specificity* p and *sensitivity* q as follows:

$$\hat{\boldsymbol{\sigma}}(a_i) = \begin{cases} 0 & \text{with probability } p \\ 1 & \text{with probability } 1-p \end{cases} \qquad \text{if } \boldsymbol{\sigma}(x_j) = 0 \text{ for all } x_j \in \partial a_i \tag{1}$$

$$\hat{\boldsymbol{\sigma}}(a_i) = \begin{cases} 0 & \text{with probability } 1-q \\ 1 & \text{with probability } q \end{cases} \qquad \text{if } \boldsymbol{\sigma}(x_j) = 1 \text{ for some } x_j \in \partial a_i \tag{2}$$

Unless $p = q = 1$ and every x_j is tested separately, the ground truth $\boldsymbol{\sigma}$ cannot be inferred perfectly form the test results $\hat{\boldsymbol{\sigma}}$ of a single "one-shot" test design [2]. Indeed, under the noise model in Eqs. (1) and (2) the posterior of the ground truth given the test results reads

$$\mu_G(\sigma) = \mathbb{P}\left[\boldsymbol{\sigma} = \sigma \mid \hat{\boldsymbol{\sigma}}\right] \propto \prod_{i=1}^{n} \lambda^{\sigma(x_i)}(1-\lambda)^{1-\sigma(x_i)} \prod_{i=1}^{m} \psi_{\hat{\boldsymbol{\sigma}}(a_i)}\left((\sigma(y))_{y\in\partial a_i}\right) \tag{3}$$

$$\text{where} \quad \psi_0(\sigma_1,\dots,\sigma_\ell) = p^{1-\bigvee_{i=1}^{\ell}\sigma_i}(1-q)^{\bigvee_{i=1}^{\ell}\sigma_i}, \quad \text{and}$$
$$\psi_1(\sigma_1,\dots,\sigma_\ell) = (1-p)^{1-\bigvee_{i=1}^{\ell}\sigma_i} q^{\bigvee_{i=1}^{\ell}\sigma_i},$$

and where the $\propto$-notation hides the normalisaton required to turn μ_G into a probability distribution. Hence, the information-theoretically optimal inference algorithm just draws a random sample from the distribution μ_G. In effect, the design's accuracy is governed by the entropy of the posterior μ_G: the smaller the entropy the better the results. Furthermore, depending on the specific design G there may or may not exist an *efficient* algorithm for sampling from μ_G.

In contrast, *adaptive group testing* uses multiple stages; an *ℓ-stage test design* is a sequence $G^{(0)}, G^{(1)}, \dots, G^{(\ell)}$ of test designs such that $G^{(i+1)}$ is obtained from $G^{(i)}$ by adding tests and edges based on the results from previous stages. The results of the new tests are assumed to be distributed independently according to Eqs. (1) and (2). The aim, of course, is to diligently add tests so as to maximally reduce the entropy of the posterior.

In summary, the group testing problem poses the following, partially conflicting, challenges:

- **(i)** We require an adaptive test design with high accuracy and a small number of tests.
- **(ii)** We require an *efficient* algorithm that infers the $\boldsymbol{\sigma}(x_j)$ from the observed $\hat{\boldsymbol{\sigma}}(a_i)$.
- **(iii)** Practical limitations require a small number of samples in a test and tests per sample.
- **(iv)** We aim for a small number of test stages to ensure a timely reporting of test outcomes – or at least ensure that most samples can be diagnosed after the first or second stage.

2 Established designs and algorithms

2.1 Individual testing

The most straightforward test strategy, of course, is to conduct $m = n$ individual tests for each of the n samples. Naturally, in the case $p = q = 1$, individual testing will register the status of each sample correctly. However, realistic values for p and q range between 0.95 and 0.99 [9, 10, 31, 40, 44]. Then individual testing will produce numbers of false positives/negatives distributed as $\mathrm{Bin}(n-k, 1-p)$ and $\mathrm{Bin}(k, 1-q)$, respectively.

The accuracy of the results could obviously be boosted by conducting two or three individual tests per sample. Indeed, if we test each x_j twice and report x_j as infected only if both tests come back positive, then we could reduce the expected number of false positives to $(n-k)(1-p)^2$. But we would now expect a slightly larger number of $2k(1-q)$ false negatives. To reduce the number of false positives and negatives simultaneously we could test each x_j thrice and report the majority of the three test results.

2.2 Dorfman

The test designs that appear to be currently most widely adapted in practice date back to the 1940s. Indeed, the idea of group testing was first brought up by Dorfman in 1943 [20]. He suggested a two-stage test procedure, we denote as Dorfman. In the first stage, every sample gets placed in precisely one pool. All pools are the same size, which depends on the prior λ only. Pools with a positive test result get tested separately in the second stage.

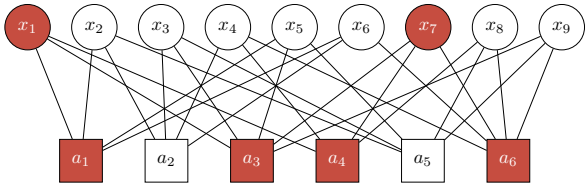


Figure 1 Illustration of a random biregular test design with $\Delta = 3$ and $\Gamma = 4$.

Depending on the prior, this scheme can significantly reduce the number of tests required. For example, with $\lambda = 0.05$ this scheme uses pools of size five and the expected overall number of tests conducted in both stages comes to about $0.426n$. At the same time, DORFMAN's two-stage procedure reduces the number of false positives because a sample is ultimately reported as positive only if both the tests are positive. But for the same reason, the expected number of false negatives increases. For instance, with $n = 10^4$ and $k = \lambda n = 500$, we expect 18.2 false positives and 9.95 false negatives.

A natural extension of the DORFMAN procedure employs three stages. In the first stage, relatively large pools are formed. The second stage then splits the positive pools into smaller sub-pools and the third stage resorts to individual testing. In effect, as with the two-stage procedure, the expected number of false positives decreases while the expected number of false negatives increases. For $n = 10^4$ and $k = \lambda n = 500$ the expected numbers of false positives/negatives work out to be 11.76 and 14.8, respectively.

2.3 Probabilistic constructions

More sophisticated test designs have been proposed in the mathematical theory of group testing. The currently best, and in certain asymptotic settings provably optimal, test designs harness randomisation [5, 13]. For instance, in the *random biregular test design* illustrated in Fig. 1 every test pool has an equal size Γ and every individual sample joins an equal number Δ of pools. In other words, the test design $G = G_{n,m}(\Gamma, \Delta)$ is chosen uniformly at random from the set of all (Δ, Γ)-regular bipartite graphs (e.g., see [41]).[1] To maximize information gained, the parameters Γ and Δ need to be chosen as to maximise the conditional entropy of the vector $\hat{\boldsymbol{\sigma}}$ of test results, i.e., so that about half the tests will be positive:[2]

$$\Delta = m\log(2)/(n\lambda) \qquad \Gamma = \log(2)/\lambda \tag{4}$$

Intuitively, the randomness of the test design minimizes dependencies between the different test results $\hat{\boldsymbol{\sigma}}(a_i)$. Thus, with the parameters as in Eq. (4) and for a number m of tests up to a threshold, we can hope to squeeze up to one bit of information from each test. Similar randomised constructions are used in coding theory and compressed sensing [17, 18, 26, 35].

Unlike previous discussions, the random biregular design has no obvious inference algorithm. For $p = q = 1$, a posteriori inference implies a minimum hypergraph vertex cover [12], which is an NP-hard problem and even on random instances no efficient algorithm is known.

[1] G is typically drawn from the pairing model [8, 34]. Then, in rare cases the same individual joins a test pool twice. In practice, such double occurrence could, of course, be reduced to single occurrences.

[2] Due to rounding issues, we cannot ensure that the expected number of positive tests is precisely $m/2$.

Definite defectives (DD) [4] is a blunt but efficient algorithm. The algorithm classifies every sample that is only included in positive test as infected under the condition that it appears in at least one positive test pool where all other samples appear in a negative test. All other samples are classified as uninfected. In symbols,

$$\boldsymbol{\sigma}_{\mathtt{DD}}(x_j) = \bigwedge_{a\in\partial x_j} \hat{\boldsymbol{\sigma}}(a) \quad \wedge \quad \bigvee_{a\in\partial x_j} \bigwedge_{y\in\partial a} \bigvee_{b\in\partial y} (1-\hat{\boldsymbol{\sigma}}(b)).$$

For $p = q = 1$ this algorithm will never produce false positives but may render false negatives. Several similarly-flavoured algorithms have been analysed mathematically. Aldridge analysed an adaptive test design whose different stages employ random biregular test designs with suitably chosen degrees [3]. This adaptive test design carried out over an unbounded number of stages (which may take too long in practice) achieves rates in excess of 0.95 bits per tests.

2.4 Glauber dynamics

While DD merely extracts binary information about each sample, we want a more fine-grained picture of the posterior distribution Eq. (3) of the random test design. Glauber dynamics (Glauber) is a Markov Chain Monte Carlo algorithm and starts at a random initial configuration $\sigma^{(0)} = (\sigma^{(0)}(x_i))_{i=1,\dots,n}$ drawn from the prior. Thus, the individual $\sigma^{(0)}(x_i)$ are independent $\mathrm{Be}(\lambda)$ variables. Glauber then proceeds to generate a random sequence $(\sigma^{(t)})_{t=0,\dots,T}$ of configurations by updating the status of a random sample at each time step according to Eq. (3); see [27] for details of the update rule. The hope is that for moderate T the empirical means of the sequence approximate the actual posteriors well, i.e.,

$$\mu_G(\{\boldsymbol{\sigma}(x_j) = s\}) \approx \frac{1}{T}\sum_{i=0}^{T} \mathbf{1}\left\{\sigma^{(t)}(x_j) = s\right\} \qquad (j = 1,\dots,n;\ s\in\{0,1\}). \tag{5}$$

We are unaware of a rigorous analysis of Glauber. Further, an exact empirical assessment appears difficult as the marginals of the posterior in Eq. (3) cannot be computed by exhaustive enumeration even for moderate values of n. Still, [16] studies Glauber experimentally.

2.5 Informative Dorfman

Informative Dorfman [29] is a multi-stage test design that uses the posterior marginals of a first stage (e.g., as approximated by Glauber) to determine the group sizes of a subsequent Dorfman test design. More precisely, it sorts the samples increasingly by their marginals and groups them in this order. The pools containing samples with small marginals are relatively large, while samples with marginals above 0.3 get tested individually. In the empirical study [16] of a combination of Glauber and InfDorfman, Cuturi et al. find that this procedure works decently well for a given number of tests but is still outperformed by quite a margin by more complicated multi-stage test designs and algorithms.

3 Adaptive Belief Propagation (aBP)

In this section we discuss our novel design aBP and its inference algorithm. The first stage employs the random biregular test design (Sec. 2.3). Given the results of the first stage, in the second and third stage we use a blend of the random biregular design and InfDorfman. For the inference algorithm we seize upon the BP message passing paradigm [33].

3.1 Belief Propagation

In recent years the Belief Propagation (BP) message passing paradigm has been applied in combination with randomised constructions with stunning success. Prominent examples include coding theory and other signal processing tasks such as compressed sensing [18, 26, 35]. The development of BP with randomised constructions has been inspired by ideas from the statistical mechanics of disordered systems [30]. More recently, substantial mathematical research has been devoted to BP (e.g., [6, 14, 21, 43]). Although most of this theoretical work is asymptotical, we let these ideas guide our quest for a practical group testing design.

BP is a generic message passing technique to approximate the marginals of Boltzmann distributions on factor graphs (e.g., Eq. (3)). The basic intuition behind BP is that under certain assumptions the posterior distribution admits a succinct representation in terms of *messages* [14, 15, 30, 45]. These assumptions are provably met in many Bayes-optimal inference problems on random factor graphs including the group testing problem as modelled in Sec. 1.2; at least asymptotically as the problem size tends to infinity [7, 11].

At first glance the posterior distribution Eq. (3) appears to be quite a difficult object to study; e.g., to estimate its entropy, we might have to inspect all 2^n possible vectors $\sigma \in \{0,1\}^n$. But according to the BP paradigm we can get a handle on the posterior distribution in terms of messages associated with the edges of the test design $G = G_{n,m}(\Gamma, \Delta)$. Formally, the *message space* of $\mathcal{M}(G)$ consists of vectors $(\mu_{x_j \to a_i}(s),\ \mu_{a_i \to x_j}(s))_{j=1,\ldots,n;\, i=1,\ldots,m;\, x_j \in \partial a_i;\, s \in \{0,1\}}$.

The idea is that there are two messages $\mu_{x_j \to a_i}(\,\cdot\,)$ and $\mu_{a_i \to x_j}(\,\cdot\,)$ associated with every edge of G, one directed from the sample x_j to the test a_i and one in the opposite direction. The messages themselves are probability distributions on $\{0,1\}$. Thus, we have $\mu_{x_j \to a_i}(0),\ \mu_{x_j \to a_i}(1) \in [0,1]$ and $\mu_{x_j \to a_i}(0) + \mu_{x_j \to a_i}(1) = 1$, and analogously for $\mu_{a_i \to x_j}(\,\cdot\,)$.

Roughly speaking, $\mu_{a_i \to x_j}(\,\cdot\,)$ represents the impact that a_i has on x_j in the absence of all other tests $b \in \partial x_j$. Moreover, $\mu_{x_j \to a_i}(\,\cdot\,)$ represents the status of x_j in the absence of test a_i. More formally, we define the *standard message* $\mu_{G, x_j \to a_i}(s)$ as the posterior probability that $\boldsymbol{\sigma}(x_j) = s$ given the test design $G - a_i$ obtained from G by omitting test a_i and given the test results $(\hat{\boldsymbol{\sigma}}(a_h))_{h \neq i}$. With the notation of Eq. (3), we can write this probability out as

$$\mu_{G, x_j \to a_i}(s) \propto \sum_{\sigma \in \{0,1\}^{\mathcal{X}},\ \sigma(x_j) = s} \prod_{i=1}^{n} \lambda^{\sigma(x_i)} (1-\lambda)^{1-\sigma(x_i)} \prod_{i=1}^{m} \psi_{\hat{\boldsymbol{\sigma}}(a_i)}\left((\sigma_y)_{y \in \partial a_i}\right)$$

with the $\propto$-sign hiding the normalisation to ensure that $\mu_{G, x_j \to a_i}(0) + \mu_{G, x_j \to a_i}(1) = 1$. Similarly, the standard message $\mu_{G, a_i \to x_j}(s)$ is defined as the posterior probability that $\boldsymbol{\sigma}(x_j) = s$ given the test design $G - (\partial x_j \setminus \{a_i\})$ obtained by removing all tests that x_j takes part in except for a_i and given the test results $\hat{\boldsymbol{\sigma}}(a_h)$ of all tests $a_h \notin \partial x_j \setminus \{a_i\}$.

Conceived wisdom, vindicated mathematically for a broad family of inference problems, predicts that asymptotically these messages satisfy the following BP equations [7, 11, 14, 45]:

$$\mu_{G, x \to a}(s) \propto \lambda^s (1-\lambda)^{1-s} \prod_{b \in \partial x \setminus \{a\}} \mu_{G, b \to x}(s), \tag{6}$$

$$\mu_{G, a \to x}(0) \propto 1 - q + (p + q - 1) \prod_{y \in \partial a \setminus \{x\}} \mu_{G, y \to a}(0), \quad \mu_{G, a \to x}(1) \propto 1 - q \qquad \text{if } \hat{\boldsymbol{\sigma}}(a) = 0, \tag{7}$$

$$\mu_{G, a \to x}(0) \propto q + (1 - p - q) \prod_{y \in \partial a \setminus \{x\}} \mu_{G, y \to a}(0), \qquad \mu_{G, a \to x}(1) \propto q \qquad \text{if } \hat{\boldsymbol{\sigma}}(a) = 1 \tag{8}$$

These equations express the notion that the random biregular design $G_{n,m}(\Gamma, \Delta)$ minimises dependencies between the test results. Furthermore, we expect that the marginals of the posterior distribution can be well approximated in terms of the messages:

$$\mu_G(\{\boldsymbol{\sigma}(x_i) = s\}) \propto \lambda^s (1-\lambda)^{1-s} \prod_{b \in \partial x_i} \mu_{G, b \to x_i}(s) \tag{9}$$

Apart from the marginals, asymptotic results also suggest that the entropy of the posterior distribution can be approximated in terms of the messages [11, 14, 30]. This approximation comes in terms of a functional called the *Bethe free energy*, defined as

$$\mathcal{B}_G = \sum_{x\in\mathcal{X}} \mathcal{B}_{G,x} + \sum_{a\in\mathcal{A}} \mathcal{B}_{G,a} - \sum_{x\in\mathcal{X},a\in\partial x} \mathcal{B}_{G,x,a} \qquad \text{with} \tag{10}$$

$$\mathcal{B}_{G,x} = \log \sum_{s\in\{0,1\}} \prod_{a\in\partial x} \mu_{G,a\to x}(s) \tag{11}$$

$$\mathcal{B}_{G,a} = \begin{cases} \log\left(1-q+(p+q-1)\prod_{x\in\partial a}\mu_{G,x\to a}(0)\right) & \text{if } \hat{\boldsymbol{\sigma}}(a)=0 \\ \log\left(q+(1-p-q)\prod_{x\in\partial a}\mu_{G,x\to a}(0)\right) & \text{if } \hat{\boldsymbol{\sigma}}(a)=1 \end{cases} \tag{12}$$

$$\mathcal{B}_{G,x,a} = \log \sum_{s\in\{0,1\}} \mu_{G,x\to a}(s)\mu_{G,a\to x}(s). \tag{13}$$

The resulting approximation of the entropy reads

$$\begin{aligned}\mathcal{H}_G = \mathcal{B}_G - n\log\lambda + \sum_{i=1}^{n}\mu_G(\{\boldsymbol{\sigma}_x=0\})\log\frac{\lambda}{1-\lambda} \\ - \sum_{\substack{i=1\\ \hat{\boldsymbol{\sigma}}(a_i)=0}}^{m}\left[\frac{p\log(p)\prod_{x\in\partial a_i}\mu_{G,x\to a_i}(0)}{1-q+(p+q-1)\prod_{x\in\partial a_i}\mu_{G,x\to a_i}(0)} + \frac{(1-q)\log(1-q)(1-\prod_{x\in\partial a_i}\mu_{G,x\to a_i}(0))}{1-q+(p+q-1)\prod_{x\in\partial a_i}\mu_{G,x\to a_i}(0)}\right] \\ - \sum_{\substack{i=1\\ \hat{\boldsymbol{\sigma}}(a_i)=1}}^{m}\left[\frac{(1-p)\log(1-p)\prod_{x\in\partial a_i}\mu_{G,x\to a_i}(0)}{q+(1-p-q)\prod_{x\in\partial a_i}\mu_{G,x\to a_i}(0)} + \frac{q\log(q)(1-\prod_{x\in\partial a_i}\mu_{G,x\to a_i}(0))}{q+(1-p-q)\prod_{x\in\partial a_i}\mu_{G,x\to a_i}(0)}\right].\end{aligned} \tag{14}$$

Hence, in order to estimate the marginals and the entropy of the posterior we need to calculate the BP messages. A natural idea is to perform a fixed point iteration using the BP Eqs. (6)–(8). These equation usually possess several solutions [11, 45]. Whether or not the fixed point iteration homes in on the correct solution then depends on the initialisation.

While there is no generic recipe for choosing an appropriate initialisation $\mu^{(0)} \in \mathcal{M}(G)$, two choices suggest themselves. First, we can initialise the messages based to the prior λ, i.e.,

$$\mu^{(0)}_{x_j\to a_i}(s) = \lambda^s(1-\lambda)^{1-s}. \tag{15}$$

We can also initialise the messages according to the ground truth, i.e. $\mu^{(0)}_{x_j\to a_i}(s) = \boldsymbol{\sigma}(x_j)$. While the latter is not practically useful for the obvious reason, the analogy with other inference problems suggests that *if* the fixed point iteration converges to the same solution from both initialisations, then this solution actually is a good approximation to the correct messages. Luckily, this can be tested using empirical simulations.

3.1.1 Preventing oscillations

The textbook method to perform the fixed point iteration is to update all messages in parallel. This means that, starting from the initialisation $(\mu^{(0)}_{x_j\to a_i})_{i,j}$, we compute all test-to-sample approximations $\mu^{(0)}_{a_i\to x_j}$ via Eqs. (6)–(8). Then we use these together with Eq. (6) to compute the next approximation $(\mu^{(1)}_{x_j\to a_i}(\,\cdot\,))_{i,j}$ to all sample-to-test messages, and so forth.

Cuturi et al. [16] demonstrated experimentally that such parallel updates do not converge. Instead, the messages oscillate between odd and even rounds. A similar observation was already made by Sejdinovic and Johnson [36]. Similar oscillations emerge in other applications

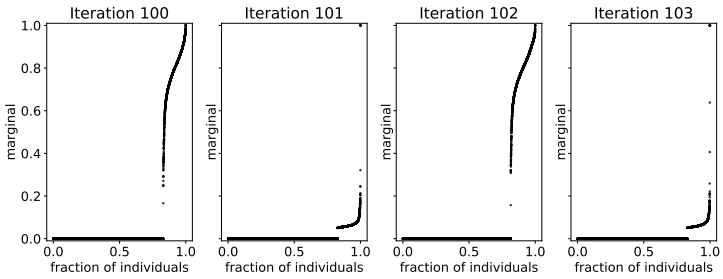


Figure 2 Oscillations in BP with parallel updates for $\lambda = 0.05$, 0.2 tests/n, and $p = q = 1$.

of BP and can also be observed in our simulations as illustrated in Fig. 2. They may result from an instability of the empirical mean of the messages. If in some particular iteration t the deviation from the prior

$$\sum_{j=1}^{n}\sum_{i=1}^{m} \mathbf{1}\{a_i \in \partial x_j\}\,(\mu^{(t)}_{x_j\to a_i}(1) - \lambda) \tag{16}$$

is positive, then we should expect a negative deviation in the next round. This is because due to Eq. (16) in the next iteration many tests will receive a relatively large indication that one of their samples may be infected. The test will therefore send out "less urgent" messages to the other samples. Conversely, if Eq. (16) is negative, then in iteration $t+1$ we expect to see a positive deviation. Due to the analytic nature of the update rules Eq. (6)–Eq. (8) these oscillations do not dampen down but actually amplify. This observation led the authors of [16] to turn to the computationally more intensive Glauber.

But actually oscillations of this type have been observed in other problems and several mitigations are known. We resort to a natural solution, namely to update the messages in a randomised order rather than in parallel to break the cycle of oscillations. Starting from the initialisation $(\mu^{(0)}_{x_j\to a_i}(\,\cdot\,))_{i,j}$, we apply Eq. (7)–Eq. (8) once to initialise the test-to-sample messages $\mu_{a_i\to x_j}(\,\cdot\,)$. Then at each time $t \geq 1$ we choose an edge $a_i x_j$ of G randomly and then randomly either process $\mu^{(t)}_{x_j\to a_i}(\,\cdot\,)$ or $\mu^{(t)}_{a_i\to x_j}(\,\cdot\,)$.

We stop the fixed point iteration after a fixed number T of steps. The precise choice of T is guided by experiments but T should be large enough so that every message will likely get updated several times. We note that this update scheme does not impede practical matters from using our algorithm in a laboratory setting since it purely pertains to the computations behind the scene and does not impact how samples are split and combined.

Beyond relying on asymptotic ideas and comparing the messages that result from the two aforementioned initialisations we take two additional steps to corroborate the results of BP. First, we compared the marginals obtained by BP with the empirical marginals of Glauber on a number of samples. They match. Second, we compared the marginals obtained via BP on moderately sized biregular test designs with the marginal distributions obtained via *population dynamics*, a heuristic intended to approximate the limiting distribution of the marginals as $n \to \infty$ [30]. They, too, align very well. Figure 3 displays the typical outcome of the BP along with the estimate Eq. (14) of the remaining entropy.

3.2 The first stage

As the first stage we use the random biregular design $G = G_{n,m}(\Delta, \Gamma)$ with the optimal parameters from Eq. (4). Thus, the only free parameter is the total number m of tests conducted in the first stage. Its choice is informed by BP. Specifically, we choose the

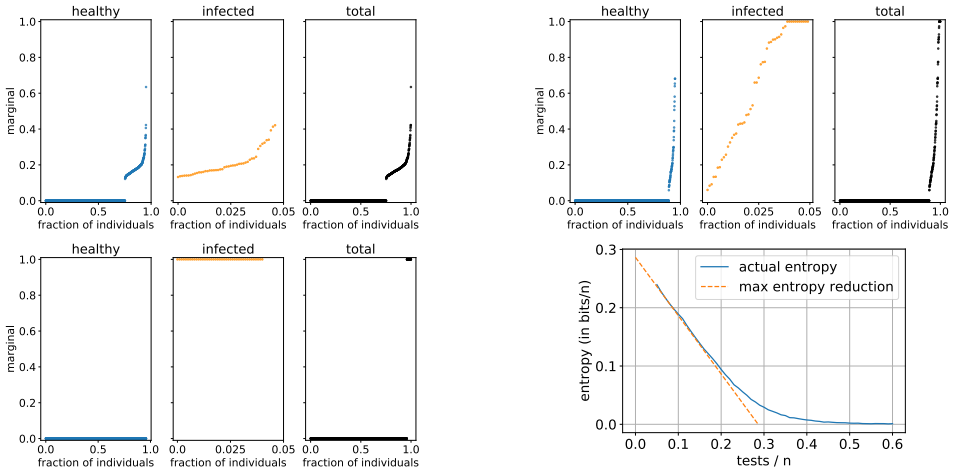


Figure 3 Posteriors of BP on a random biregular design with 0.15 (top left), 0.25 (top right) and 0.6 (bottom left) tests/n and remaining entropy (bottom right) for $\lambda = 0.05$ and $p = q = 1$.

largest number m of tests up to which each test yields the optimal entropy reduction of $\ln 2$. Practically, this means that we choose m to match the point at which the entropy plot for the corresponding parameter values flattens. The fourth graphic in Fig. 3 shows the approximation of the entropy as a function of the number of tests for $n = 1000$ and $\lambda = 0.05$ in the noiseless setting. For other priors and noise levels, the story turns out to be analogous.

3.3 The second and third stage

Given the approximation of the marginals from the first stage, how should we proceed? As we saw in Sections 2.3 and 2.5, two ideas for the subsequent stages proposed in the literature include InfDorfman as well as individual testing of all samples whose marginals are not entirely polarised after the first round. The former suffers from the same problem as the original Dorfman scheme, namely a potentially fairly large number of false positives and negatives. The latter strategy, known as DD, seems wasteful as it completely disregards any non-trivial information about the marginals resulting from the BP computation.

To remedy these issues, we propose a new design that blends the random biregular design with the InfDorfman scheme from Sec. 2.5. First, we directly report samples with marginals obtained from the first stage marginals less than 0.1% as healthy and those with marginals beyond 99.9% as infected. As illustrated in Fig. 4, the remaining samples are split into two groups, one comprising samples with marginals below 12.4% (*low risk*) and one with marginals above (*high risk*). The choice of 12.4% marks precisely the threshold beyond which the expressions Eq. (4) suggest that any sample should be placed in one test only.

For the high risk group, we set up an InfDorfman design G''. If such a pooled test turns out to be negative, we classify all samples in this pool as healthy. Otherwise, we conduct individual tests and classify samples solely based on these individual test results.

For the low risk group, we set up another random biregular test design on which we run BP where the priors are given by the posteriors of the first stage. The resulting marginals are again thresholded at 0.1% and 99.9%. Those samples whose marginals fall in between are subsequently retested individually with their classification being solely determined by the outcome of the individual test. To be more precise, let $\mathcal{X}'$ be the samples in the low risk

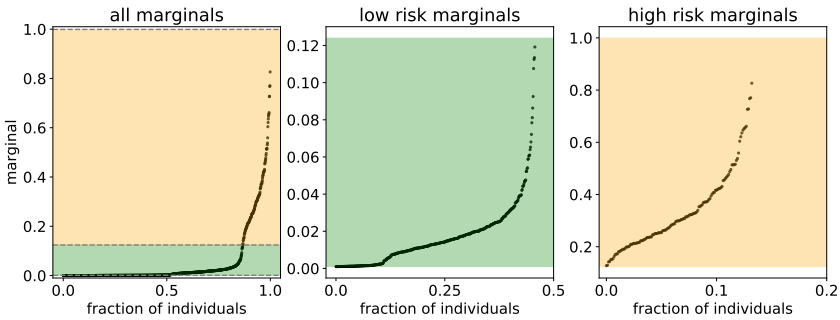


Figure 4 Low and high risk marginals for $\lambda = 0.05$ in the high noise setting with $m/n = 0.25$.

group, let $n' = |\mathcal{X}'|$ and let m' be the number of tests dedicated to this group. Based on the first round's BP results we approximate the average marginal $\lambda' = \frac{1}{n'} \sum_{x \in \mathcal{X}'} \mu_G(\{\boldsymbol{\sigma}(x) = 1\}$. Mimicking Eq. (4) we then choose the degrees

$$\Delta' = m' \log(2)/(n'\lambda') \qquad \Gamma' = \log(2)/\lambda' \tag{17}$$

and set up a random biregular test design $G' = G_{n',m'}(\Delta', \Gamma')$ on $\mathcal{X}'$. Furthermore, we modify the BP equations on this random biregular design to accommodate the marginals computed in the first stage. Hence, instead of using the universal prior λ' for all the samples, we substitute the separate marginals computed in the first stage:

$$\mu_{G',x\to a}(s) \propto \mu_G(\{\hat{\boldsymbol{\sigma}}(x) = 1\})^s (1 - \mu_G(\{\hat{\boldsymbol{\sigma}}(x) = 1\})^{1-s} \prod_{b \in \partial x \setminus \{a\}} \mu_{b\to x}(s) \tag{18}$$

3.4 Enhanced accuracy

In the following, we discuss a trade-off between accuracy and number of tests. The construction discussed so far is denoted as aBP-1, and the more accurate variants are aBP-2 and aBP-3. In aBP-1, almost all false results originate from the InfDorfman procedure in the second stage and neither the thresholding nor the second-stage random biregular design tend to produce a notable number of mistakes. Therefore, in aBP-2 and aBP-3 we perform the InfDorfman procedure twice or thrice independently in parallel.[3]

If we perform InfDorfman twice (aBP-2), we need to choose whether to reduce false negatives or false positives. Accordingly, we classify a sample as healthy (infected) if both InfDorfman procedures classify it as healthy (infected). In aBP-3, we classify according to the majority vote of the three InfDorfman schemes. We refer to Appendix A for a listing of the number of tests to be performed in the first and second stage.

4 Empirical investigation

In this section, we consider instance of $n = 1000$ samples and omit extensive simulations for $n = 100$ and $n = 10\,000$ since the results presented here, particularly the power of aBP carry over to those sizes. For smaller instance, rounding issues and few samples in the second stage necessitate slightly more tests; for larger n, we obtain a better performance.

[3] In case of aBP-3, we opted to keep the number of stages small. Instead, we may also run aBP-2 and only carry out a third round on samples where both runs yield different results.

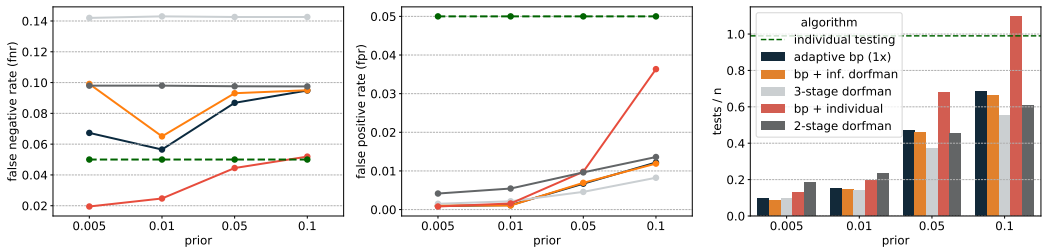


Figure 5 High noise scenario (sensitivity and specificity of $p = q = 95\%$).

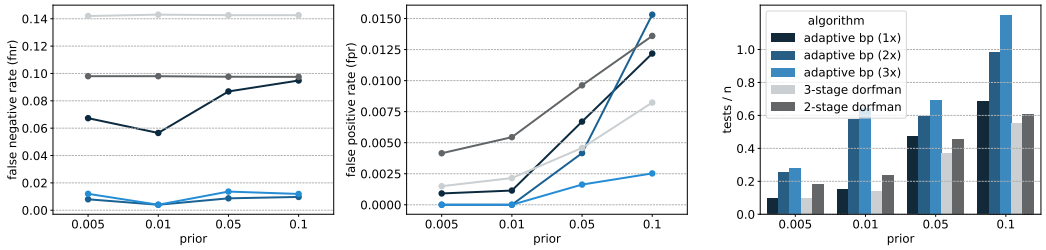


Figure 6 Reliability-enhanced ABP for high noise scenario (sensitivity/specificity of $p = q = 95\%$).

We study infection rates $\lambda \in \{0.5\%, 1\%, 5\%, 10\%\}$ and the following specificity/sensitivity scenarios (details on the parametrizations of the test designs for these settings are reported in Appendices A and B):

(a) perfectly reliable tests, i.e. $p = q = 1$,

(b) moderately values $p = 0.99$ and $q = 0.98$ (e.g., certain Covid-19 tests [9, 10, 31, 40, 44])

(c) a noisy scenario with $p = q = 0.95$.

Our experiments show that ABP improves the accuracy by an order of magnitude compared to known test designs while keeping the number of tests at a reasonable level. In the following, let the *false positive rate (fpr)* be the number of healthy samples falsely classified as infected over all healthy samples; define the *false negative rate (fnr)* analogously.

In the **High-noise scenario** with $p = q = 0.95$, ABP reaps the greatest gains. Figure 5 displays the results of ABP-1 in comparison to several previously known approaches. These include the widely used two- and three-stage DORFMAN designs (Sec. 2.2), the INFDORFMAN design (Sec. 2.5) as well as BP followed by individual testing advocated in the theoretical literature[4]. The figure shows that with about the same number of tests as 2-stage DORFMAN, ABP achieves up to 78% reduction in the number of false positives and an up to 42% reduction in the number of false negatives. The gains are particularly high for small priors.

Still, the absolute error rates in Fig. 5, particularly for large priors, may still be prohibitive for many real-world applications. Here our two designs ABP-2 and ABP-3 (Sec. 3.4) come to the rescue. As Fig. 6 shows, these designs, particularly ABP-3, dramatically reduce the number of false positives and negatives. Of course, these improvements come at the expense of a larger number of tests. But for priors $\lambda \leq 0.05$ the number of extra tests is moderate, and for the largest prior $\lambda = 0.1$ ABP-2 and ABP-3 require not many more tests than individual testing while being the only designs that deliver decent accuracy.

[4] With perfectly reliable tests, this approach is equivalent to DD algorithm followed by individual testing.

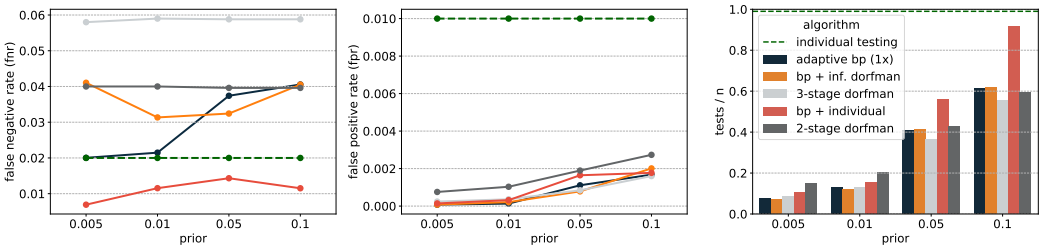


Figure 7 Sensitivity for moderate noise scenario ($p = 98\%$, $q = 99\%$).

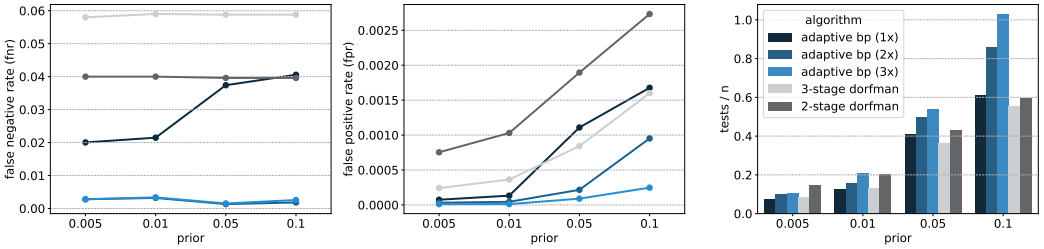


Figure 8 Reliability-enhanced ABP for moderate noise ($p = 98\%$, $q = 99\%$).

Figure 7 indicates a similar behaviour for **moderately high noise** with $p = 0.99$, $q = 0.98$. In comparison to the classical two- and three-stage DORFMAN, ABP requires at most 11% more tests for high priors of $\lambda = 0.1$ and even fewer for small priors. The benefit is that ABP boosts accuracy compared to all the previously known designs, particularly so for low priors. We point out that the gains vis-a-vis INFDORFMAN for moderately high priors are modest. The key benefit in ABP lies in its versatility to meaningfully enhance the accuracy at the expense of somewhat more tests as shown in Fig. 8. A similar extension of INFDORFMAN would yield a similar accuracy but require significantly more tests than ABP.

Even with **perfectly reliable** tests, the conventional DD approach (Sec. 2.3) is improved upon by ABP or the INFDORFMAN approach. Both schemes are able to reduce the number of tests compared to the former by up to 18% and comes within 19% to 32% of the information-theoretic lower bound. The gains vis-a-vis two-stage DORFMAN with up to 57% and individual testing with up to 94% are even more pronounced. We do not need to consider the accuracy in the noiseless case since all test designs recover the entire ground truth by construction.

In Fig. 10, we consider the fraction of samples that are identified by in each stage. It highlights that despite a total of three stages needed for ABP the majority of samples are identified already in the first and second stage, depending on the prior and noise level.

All examined algorithms require reasonable pool sizes and splits of the individual sample that are in line with common pooling procedures [22, 24, 29]. The maximum pool size is between 8 and 170 depending on noise level and prior, while the splits of the individual sample range between 3 and 19. It should be noted that the proposed algorithms and test designs can readily be adjusted to accommodate smaller pool sizes or individual sample splits – at the expense of somewhat more tests.

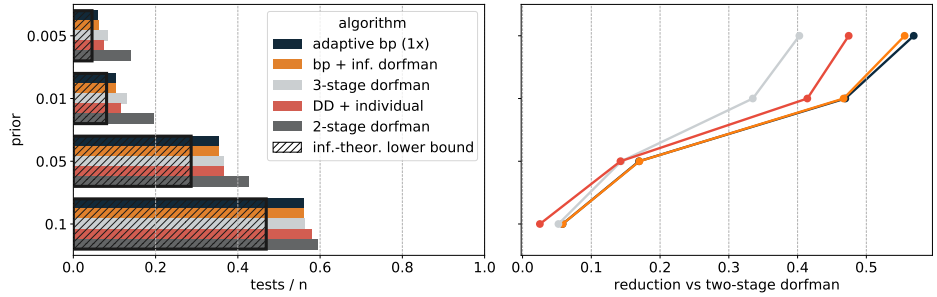


Figure 9 Simulation results for the noiseless setting. The black area represents a plausible information-theoretic lower bound for the number of tests. The left plot displays the numbers of tests required by the different designs; the right plot shows the reduction achieved by comparison to the 2-stage DORFMAN procedure, a classical and widely used test design.

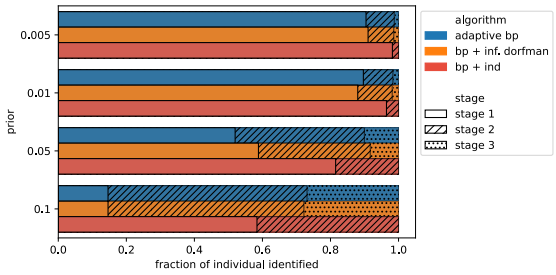


Figure 10 Fraction of samples identified in each stage by (i) ABP, (ii) BP followed by individual testing, and (iii) BP followed by INFDORFMAN.

5 Asymptotic considerations

Clearly, ABP relies on heuristics and is not asymptotically optimal. This begs the question of how we would adapt the design and algorithm if we decide to live unburdened by practical considerations and consider the case $n \to \infty$?

5.1 Variations on aBP

The optimal drop in entropy in Fig. 3 encourages running BP on a random biregular test design in the first. The discrete partition into three groups in the second stage, however, gives something away. Indeed, in the asymptotic regime infinitesimal intervals of posterior marginals contain an unbounded number of samples.[5] Thus, it seems information-theoretically optimal to construct a random biregular design for every single small marginal interval and repeat this procedure over a few stages. However, such an approach is impractical as, for moderate n, each random biregular design would only contain very few samples.

A simpler alternative that we considered is to still include all samples in one single second-stage test design, in which we choose the number of tests in which each sample takes part according to the posterior marginal from the first stage. Specifically, we chose these numbers so that in expectation half the tests should be positive. However, this design turned out to be unstable for small values of n because of random fluctuations.

[5] Of course, depending on the prior and the noise setting the distribution of the posterior marginals need not be supported on the entire unit interval.

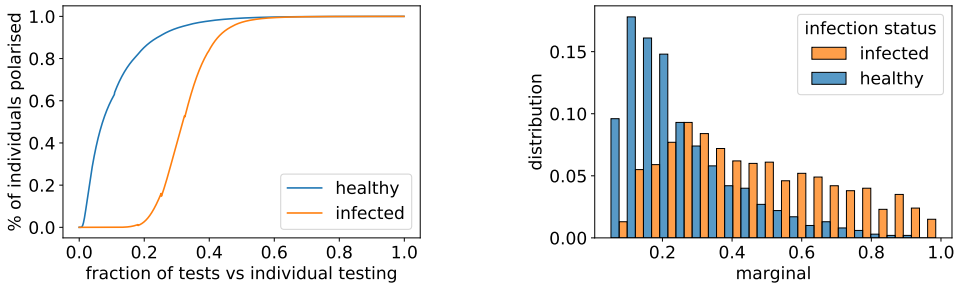


Figure 11 Asymptotic fraction of polarised marginals and the posteriors for non-polarised samples obtained with population dynamics on the distribution by [23] for $\lambda = 0.05$ and $p = q = 1$.

5.2 Plain Belief Propagation

Thus far we disregarded what might seem at first glance the most straightforward scheme: just run BP on a random biregular design and then simply threshold the marginals at, say, 50%. An obvious advantage of this approach is that it requires one stage only. Indeed, when we simulated this scheme for large group testing instances such as $n = 10\,000$, this approach turned out to work extremely well. Particularly for small priors such as 0.5% and 1% the plain BP plus thresholding design is on par or even outperforms ABP in terms of both efficiency and reliability. However, for smaller values of n plain BP plus threshold is extremely vulnerable to fluctuations of the number k of infected samples. This is because such fluctuations might cause the fraction of positive tests to significantly deviate from half.

5.3 Population dynamics

As already discussed, the *population dynamics* heuristics allows us to get a glimpse of the marginal distribution resulting from running BP as $n \to \infty$ [30]. To this end, we require as input the distribution of infected and healthy samples in the local neighbourhood of a sample which is provided in [23]. Subsequently, we iteratively sample the local neighbourhood for infected and healthy samples and perform one-step BP updates to model the marginal distribution of those samples whose marginal is not completely polarised. As shown in Fig. 11, the resulting distribution closely resembles the marginal distribution that we observe from running BP in our simulation in the first stage. As a side product, we obtain the proportion of polarised healthy and infected samples which lines up nicely with our simulation results. It should be noted that the population dynamics heuristic is nowhere near a complete analysis of BP on random biregular graphs. Given the gains in efficiency and reliability that we observe in this empirical work for moderately-sized instances, a formal analysis of BP seems to be an important next step in group testing research.

6 Discussion

Group testing is a powerful method to efficiently and accurately detect infected samples. Since the mathematical work on group testing deals with the asymptotic $n \to \infty$ scenario, practical adoption of methods proposed in this literature has been limited. Instead practitioners tend to apply very simple test designs dating back to the 1940s. In this paper we therefore conducted an experimental study that shows how a mildly more sophisticated test design can significantly improve the accuracy of the overall test results by comparison to classical methods without asking for many more tests. The new test design comes with an efficient,

easy-to-run and easy-to-implement algorithm that determines the status of each sample from the test results. Since the new design employs randomisation, its adoption is probably feasible only in a practical setting that employs a degree of automation in preparing test pools. But on the plus side the new ABP design keeps the pool sizes and the number of pools that each sample has to be placed in fairly low.

Apart from the group testing model studied in the present paper, there are complicated models; e.g., in *quantitative group testing* each test returns the *number* of infected samples rather than a binary positive or negative result. Further variants include the pooled data problem, the generalised coin weighing problem or the compressed sensing problem [1, 19].

What are the loose ends of the present work? On the one hand, it seems worthwhile to consider alternative noise models. A candidate might be one where the specificity decreases in the test size. Both the fixed noise model considered in this work and this diluted model have value from a practical perspective and it would be interesting to see whether our results carry over. On the other hand, the success of BP in practical group testing leaves us wondering whether it is guaranteed to converge to a fixpoint reminiscent of the ground truth. Hence, a mathematical analysis of BP remains as an outstanding open problem.

References

1 A. El Alaoui, A. Ramdas, F. Krzakala, L. Zdeborová, and M. I. Jordan. Decoding from pooled data: Phase transitions of message passing. *IEEE Transactions on Information Theory*, 65:572–585, 2019.

2 M. Aldridge. Individual testing is optimal for nonadaptive group testing in the linear regime. *IEEE Transactions on Information Theory*, 65:2058–2061, 2018.

3 M. Aldridge. Conservative two-stage group testing. *arXiv*, 2020. `arXiv:2005.06617`.

4 M. Aldridge, L. Baldassini, and O. Johnson. Group testing algorithms: Bounds and simulations. *IEEE Transactions on Information Theory*, 60:3671–6687, 2014.

5 M. Aldridge, O. Johnson, and J. Scarlett. *Group testing: an information theory perspective*. Foundations and Trends in Communications and Information Theory, 2019.

6 V. Bapst and A. Coja-Oghlan. Harnessing the bethe free energy. *Random Structures and Algorithms*, 49:694–741, 2016.

7 J. Barbier and D. Panchenko. Strong replica symmetry in high-dimensional optimal bayesian inference. *arXiv*, 2020. `arXiv:2005.03115`.

8 E. A. Bender and E. R. Canfield. The asymptotic number of labeled graphs with given degree sequences. *Journal of Combinatorial Theory and Series A*, 24, 1978.

9 V. Brault, B. Mallein, and JF Rupprecht. Group testing as a strategy for covid-19 epidemiological monitoring and community surveillance. *PLOS Computational Biology*, 17:e1008726, 2021.

10 A. Cohen and B. Kessel. False positives in reverse transcription pcr testing for sars-cov-2. *medRxiv*, page 10.1101/2020.04.26.20080911, 2020.

11 A. Coja-Oghlan, C. Efthymiou, N. Jaafari, M. Kang, and T. Kapetanopoulos. Charting the replica symmetric phase. *Communications in Mathematical Physics*, 359:603–698, 2018.

12 A. Coja-Oghlan, O. Gebhard, M. Hahn-Klimroth, and P. Loick. Information-theoretic and algorithmic thresholds for group testing. *Proc. 46th ICALP*, page #43, 2019.

13 A. Coja-Oghlan, O. Gebhard, M. Hahn-Klimroth, and P. Loick. Optimal group testing. *Proc. 33rd COLT*, pages 1374–1388, 2020.

14 A. Coja-Oghlan and W. Perkins. Belief propagation on replica symmetric random factor graph models. *Annales de l'institut Henri Poincare D*, 5:211–249, 2018.

15 A. Coja-Oghlan and W. Perkins. Bethe states of random factor graphs. *Communications in Mathematical Physics*, 366:273–201, 2019.

16 M. Cuturi, O. Teboul, O. Berthet, A. Doucet, and J. Vert. Noisy adaptive group testing using bayesian sequential experimental design. *arXiv*, 2020. arXiv:2004.12508.

17 D. Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52:1289–1306, 2006.

18 D. Donoho, A. Javanmard, and A. Montanari. Information-theoretically optimal compressed sensing via spatial coupling and approximate message passing. *IEEE Transactions on Information Theory*, 59:7434–7464, 2013.

19 D. Donoho, A. Maleki, and A. Montanari. Message-passing algorithms for compressed sensing. *Proceedings of the National Academy of Sciences*, 106:18914–18919, 2009.

20 R. Dorfman. The detection of defective members of large populations. *Annals of Mathematical Statistics*, 14:436–440, 1943.

21 C. Efthymiou, T. Hayes, D. Stefankovic, E. Vigoda, and Y. Yin. Convergence of mcmc and loopy bp in the tree uniqueness region for the hard-core model. *SIAM J. Comput.*, 48:581–643, 2019.

22 L. Garrison, J. Babigumira, A. Masaquel, B. Wang, D. Lalla, and M. Brammer. The lifetime economic burden of inaccurate her2 testing: Estimating the costs of false-positive and false-negative her2 test results in us patients with early-stage breast cancer. *Journal of the International Society for Pharmacoeconomics and Outcomes Research*, 18:541–546, 2015.

23 O. Gebhard and P. Loick. Note on the offspring distribution for group testing in the linear regime. *arXiv*, 2021. arXiv:2103.13039.

24 E. Joly and B. Mallein. Group testing and pcr: a tale of charge value. *arXiv*, 2020. arXiv:2012.09096.

25 S. Kleinman, D. Strong, G. Tegtmeier, P. Holland, J. Gorlin, C. Cousins, R. Chiacchierini, and L. Pietrelli. Hepatitis b virus (hbv) dna screening of blood donations in minipools with the cobas ampliscreen hbv test. *Transfusion*, 45:1247–1257, 2005.

26 F. Krzakala, M. Mézard, F. Sausset, Y. Sun, and L. Zdeborová. Statistical-physics-based reconstruction in compressed sensing. *Physical Review X*, 2:021005, 2012.

27 D. Levin, Y. Peres, and E. Wilmer. *Markov chains and mixing times*. AMS, 2 edition, 2017.

28 S. Mallapaty. The mathematical strategy that could transform coronavirus testing. *Nature*, 583:504–505, 2020.

29 C. McMahan, J. Tebbs, and C. Bilder. Informative dorfman screening. *Biometrics*, 68:287–296, 2012.

30 M. Mézard and A. Montanari. *Information and physics and computation*. Oxford University Pres, 2009.

31 M. Mueller, P. Derlet, C. Mudry, and G. Aeppli. Testing of asymptomatic individuals for fast feedback-control of covid-19 pandemic. *Physical biology*, 17:065007, 2020.

32 Y. Ohhashi, A. Pai, H. Halait, and R. Ziermann. Analytical and clinical performance evaluation of the cobas taqscreen mpx test for use on the cobas s201 system. *Journal of Virological Methods*, 165:246–253, 2010.

33 J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

34 M. Penschuck, U. Brandes, M. Hamann, S. Lamm, U. Meyer, I. Safro, P. Sanders, and C. Schulz. Recent advances in scalable network generation. *arXiv*, 2020. arXiv:2003.00736.

35 T. Richardson and R. Urbanke. *Modern coding theory*. Cambridge University Press, 2008.

36 D. Sejdinovic and O. Johnson. Note on noisy group testing: Asymptotic bounds and belief propagation reconstruction. *48th Annual Allerton Conference on Communication and Control and and Computing*, pages 998–1003, 2010.

37 Noam Shental, Shlomia Levy, Vered Wuvshet, Shosh Skorniakov, Bar Shalem, Aner Ottolenghi, Yariv Greenshpan, Rachel Steinberg, Avishay Edri, Roni Gillis, Michal Goldhirsh, Khen Moscovici, Sinai Sachren, Lilach M. Friedman, Lior Nesher, Yonat Shemer-Avni, Angel Porgador, and Tomer Hertz. Efficient high-throughput sars-cov-2 testing to detect asymptomatic carriers. *Science Advances*, 6:eabc5961, 2020.

38 M. Sherlock, N. Zelota, and J. Klausner. Routine detection of acute hiv infection through rna pooling: Survey of current practice in the united states. *Sexually Transmitted Diseases*, 34:314–316, 2007.

39 J. Tebbs, C. McMahan, and C. Bilder. Two-stage hierarchical group testing for multiple infections with application to the infertility prevention project. *Biometrics*, 69:1064–1073, 2013.

40 L. Theagarajan. Group testing for covid-19: how to stop worrying and test more. *arXiv*, 2020. `arXiv:2004.06306`.

41 R. van der Hofstad. *Random Graphs and Complex Networks*. Cambridge Series in Statistical and Probabilistic Mathematics, 2016.

42 G. van Zyl, W. Preiser, S. Potschka, A. Lundershausen, R. Haubrich, and D. Smith. Pooling strategies to reduce the cost of hiv-1 rna load monitoring in a resource-limited setting. *Clinical Infectious Diseases*, 52:264–270, 2011.

43 P. Vontobel. Counting in graph covers: a combinatorial characterization of the bethe entropy function. *IEEE Transactions on Information Theory*, 59:6018–6048, 2013.

44 J. Watson, P. Whiting, and J. Brush. Interpreting a covid-19 test result. *BMJ*, page 369, 2020.

45 L. Zdeborová and F. Krzakala. Statistical physics of inference: thresholds and algorithms. *Advances in Physics*, 65:453–552, 2016.

A Number of tests in first and second stage

Number of tests for the first and second stage found via optimization for various algorithms, priors and noise levels. The number of tests in the second stage in terms of the stated parameter c can be obtained as $c\lambda' n' \log(n')$ with λ' and n' defined as the average marginal and size of the low risk group, respectively.

		noiseless		moderate noise		high noise	
algorithm	prior	m1/n	c	m1/n	c	m1/n	c
BP + individual testing	0.5%	0.05	n/a	0.09	n/a	0.11	n/a
	1%	0.08	n/a	0.12	n/a	0.16	n/a
	5%	0.23	n/a	0.37	n/a	0.45	n/a
	10%	0.3	n/a	0.7	n/a	0.34	n/a
BP + InfDorfman	0.5%	0.045	n/a	0.05	n/a	0.045	n/a
	1%	0.075	n/a	0.075	n/a	0.1	n/a
	5%	0.28	n/a	0.24	n/a	0.16	n/a
	10%	0.125	n/a	0.1	n/a	0.1	n/a
aBP-1	0.5%	0.035	1.0	0.05	2.0	0.05	2.0
	1%	0.075	1.0	0.085	2.0	0.1	2.0
	5%	0.28	1.0	0.18	2.0	0.16	2.0
	10%	0.125	0.25	0.15	4.0	0.1	2.0
aBP-2	0.5%	n/a	n/a	0.075	8.0	0.02	8.0
	1%	n/a	n/a	0.12	8.0	0.03	8.0
	5%	n/a	n/a	0.4	2.0	0.36	2.0
	10%	n/a	n/a	0.5	2.0	0.325	2.0
aBP-3	0.5%	n/a	n/a	0.075	8.0	0.02	8.0
	1%	n/a	n/a	0.085	8.0	0.03	8.0
	5%	n/a	n/a	0.4	2.0	0.4	2.0
	10%	n/a	n/a	0.55	2.0	0.5	2.0

B Sample splits and test degree

The algorithms required the following number of maximum test degree and the following maximum and average split of samples. The algorithms can be readily adjusted to work with smaller test degrees or sample splits at the expense of slightly more tests.

		noiseless			moderate noise			high noise		
algorithm	prior	$\Gamma_{\max}$	$\Delta_{\max}$	Δ_{mean}	$\Gamma_{\max}$	$\Delta_{\max}$	Δ_{mean}	$\Gamma_{\max}$	$\Delta_{\max}$	Δ_{mean}
BP + individual testing	0.5%	140	8	7.0	134	13	12.0	137	16	15.0
	1%	75	7	6.0	67	9	8.0	69	12	11.0
	5%	14	4	3.1	14	6	5.2	14	7	6.2
	10%	7	3	2.3	8	6	5.2	6	3	2.8
BP + InfDorfman	0.5%	134	8	6.0	140	9	7.1	134	8	6.2
	1%	67	7	5.1	67	7	5.2	70	9	7.2
	5%	15	6	4.1	20	5	3.5	13	4	2.9
	10%	8	3	1.8	14	3	2.3	10	3	2.3
aBP	0.5%	143	8	5.2	140	11	7.6	140	13	8.0
	1%	67	8	5.1	71	12	6.7	70	13	8.0
	5%	15	7	4.1	147	12	6.2	66	12	6.5
	10%	8	3	1.8	172	19	10.3	50	10	4.9

Efficient Exact Learning Algorithms for Road Networks and Other Graphs with Bounded Clustering Degrees

Ramtin Afshar
University of California, Irvine, CA, USA

Michael T. Goodrich
University of California, Irvine, CA, USA

Evrim Ozel
University of California, Irvine, CA, USA

Abstract

The completeness of road network data is significant in the quality of various routing services and applications. We introduce an efficient randomized algorithm for exact learning of road networks using simple distance queries, which can find missing roads and improve the quality of routing services. The efficiency of our algorithm depends on a *cluster degree* parameter, $d_{\max}$, which is an upper bound on the degrees of vertex clusters defined during our algorithm. Unfortunately, we leave open the problem of theoretically bounding $d_{\max}$, although we conjecture that $d_{\max}$ is small for road networks and other similar types of graphs. We support this conjecture by experimentally evaluating our algorithm on road network data for the U.S. and 5 European countries of various sizes. This analysis provides experimental evidence that our algorithm issues a quasilinear number of queries in expectation for road networks and similar graphs.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Random network models; Theory of computation → Query learning

Keywords and phrases Road Networks, Exact Learning, Graph Reconstruction, Randomized Algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.9

Supplementary Material *Software (Source Code)*: `https://github.com/UC-Irvine-Theory/RoadNetworkReconstruction`; archived at `swh:1:dir:337dc5ac6d78bd68031d6f94b24332b935797a67`

1 Introduction

We study the problem of reconstructing an undirected, unweighted and connected graph $G = (V, E)$, by taking as input its set of vertices V and issuing queries to a *distance oracle*, which takes as input a pair of vertices $u, v \in V$ and returns the number of edges on the shortest path between them. The goal is to learn the edges in E by using the results that are returned from these queries. In particular, we are concerned with reconstructing *road networks*, which have been characterized in numerous ways, e.g., see [18, 21, 22, 25]. As a starting point, we can view road networks as undirected, unweighted, and connected graphs with a constant maximum degree, where each vertex corresponds to a road junction or terminus, and each edge corresponds to road segments that connect two vertices. In this paper, we present a randomized incremental algorithm for *exact learning* of road networks, where we assume the existence of a distance oracle that responds to distance queries.

Even though our algorithm only works with unweighted graphs, it is possible to use weighted graphs as input by subdividing each edge, replacing each edge e with $\lceil w(e) \rceil$ edges, where $w(e)$ is the weight of e. Since the average edge weight in road networks is typically small (e.g., as observed in [25]), this will only increase the number of vertices and edges in the

graph by a constant factor that is independent of the size of the graph. This preprocessing step is important for applications of road network reconstruction in routing services, where the completeness of road network data has great importance. For example, machine learning techniques have been utilized in the past to find the missing roads in incomplete road network data [24]. Though our experiments focus on unweighted road networks, we include some experimental results for weighted road networks with subdivided edges as well.

Another application relevant to this work is the use of structured encryption [16] in the context of cloud computing, where a data owner encrypts structured data, such as a graph, stores it in a database managed by a third-party cloud provider, and wishes to query it privately (e.g., using single-pair shortest path queries [26]). In the scenario where an adversary server is able to generate valid queries of its own, it would be able to use a graph reconstruction algorithm to learn the edges in the graph, resulting in a breach of privacy.

A graph reconstruction algorithm A is evaluated based on the number of queries it issues, which we call the *query complexity* of A, following nomenclature from learning theory (e.g., see [2, 3, 17, 20, 35]) and complexity theory (where this is also known as "decision-tree complexity," e.g., see [38, 15]). For instance, Kannan, Mathieu and Zhou [29] present exact learning algorithms for connected, undirected graphs that have bounded degree, including a randomized algorithm that has expected query complexity $O(\Delta^3 n^{3/2} \operatorname{polylog}(n))$, where Δ is the maximum degree of the graph, using distance queries. This bound simplifies to $O(n^{3/2} \operatorname{polylog}(n))$ for graphs with maximum degree $O(\operatorname{polylog}(n))$.

We note that a bound on the maximum degree is necessary for subquadratic exact learning algorithms, as there is a simple $\Omega(n^2)$ lower bound for the query complexity of graphs with unbounded degrees, e.g., see [29]. Likewise, a trivial upper bound for the task of reconstructing a general graph G is $O(n^2)$ distance queries, as one can issue a distance query for every pair of vertices in the graph and return all pairs of vertices that have distance 1 between them as edges. We refer to this as an *exhaustive search* on G.

1.1 Related Prior Results

The problem of reconstructing graphs by issuing queries has been studied extensively, e.g., see [1, 5, 8, 13, 32, 37, 4, 6, 7, 9, 10, 11, 17, 19, 27, 28, 30, 33]. These works differ in terms of their assumptions about the hidden graph (e.g., whether the hidden graph is a tree, a general graph, or something else) or the types of queries that they issue.

In terms of the most relevant prior work, Kannan, Mathieu and Zhou [29] showed how to reconstruct a connected, unweighted graph G using $O(\Delta^3 n^{3/2} \operatorname{polylog}(n))$ distance queries in expectation, where they performed an exhaustive search on the Voronoi cells created by a call to a graph clustering algorithm inspired by Thorup and Zwick [36]. They also raised the open question of whether we can achieve an algorithm that uses $O(n \operatorname{polylog}(n))$ distance queries in expectation for bounded degree graphs. In a recent work [31], Mathieu and Zhou provided a partial answer for that open question by providing an algorithm that uses $O(n \operatorname{polylog}(n))$ distance queries in expectation for random Δ-regular graphs. However, this does not imply an algorithm with an expected query complexity of $O(n \operatorname{polylog}(n))$ distance queries for road networks as they are not necessarily regular. For general graphs of bounded degree, their algorithm uses $O(n^{5/3} \operatorname{polylog}(n))$ distance queries in expectation.

In another work, Afshar, Goodrich, Matias and Osegueda [6] introduced a parallel implementation of the graph clustering technique of Thorup and Zwick [36] and presented a parallel algorithm for reconstructing connected, unweighted graphs using $O(\Delta^2 n^{3/2+\epsilon})$ distance queries in $O(1)$ parallel rounds for constant $0 < \epsilon < 1/2$, with high probability.

1.2 Our Contributions

In this paper, we introduce a randomized incremental algorithm for exact reconstruction of bounded degree graphs and demonstrate through experiments that it has expected empirical query complexity $O(n \operatorname{polylog}(n))$, providing an empirical answer to the open question raised by [29] mentioned above.

The main idea of our algorithm is to cluster the graph into cells by incrementally selecting random vertices as centers. We then issue distance queries between that center and the rest of the graph to decide which vertices should be added to the new cell. We continue this process until the size of each cell is below some threshold value. The final step is to then perform exhaustive searches in each cell.

Our algorithm uses the same overall strategy used in [29], which is based on finding a Voronoi cell decomposition of the graph. However, our algorithm differs in a number of important ways. In [29], the goal of the algorithm is to produce cells such that the size of each cell is $O(\sqrt{n}/\Delta)$. Our algorithm, however, produces cells that have size at most a chosen constant. Since performing exhaustive searches on all of these cells requires only $O(n)$ queries, the query complexity of our algorithm depends mainly on the initial step of constructing the cells and not the exhaustive querying step. Moreover, our algorithm incrementally constructs the cell decomposition by updating it with each newly added center, whereas [29] updates the cell decomposition only after adding multiple centers.

We perform experiments on several real-world road networks and show, by considering the number of queries performed at each step, that our algorithm has expected empirical query complexity $O(n \operatorname{polylog}(n))$. Moreover, we theoretically analyze our algorithm and prove an upper bound of $O(d_{\max}^2 n \log n)$ expected queries, where $d_{\max}$ is the maximum degree in the dual graph of cells during our algorithm. To characterize $d_{\max}$, we collect data on the maximum cell degrees during our experiments, and find that the value of $d_{\max}$ scales logarithmically with respect to n for road networks. When combined with our theoretical analysis, this results in an alternative way to obtain an empirical upper bound of $O(n \operatorname{polylog}(n))$ expected queries for our algorithm. In addition, we perform experiments to directly compare the number of queries our algorithm issues to the number of queries issued by existing algorithms, and observe empirically that our algorithm issues significantly fewer queries.

Our paper is organized as follows. We provide some preliminaries in Section 2, our algorithm is in Section 3, the results from our analysis are in Section 4, experimental results are in Section 5, comparisons between theoretical/experimental results are in Section 6, and the conclusions are in Section 7.

2 Preliminaries

We reconstruct graphs $G = (V, E)$ that consist of $n = |V|$ vertices and $m = |E|$ edges, and are undirected, unweighted and connected. For a graph $G = (V, E)$, a *cell* is defined as any subset of V. A *cover* of G is a set of cells $\mathcal{C}$ such that $\bigcup_{C \in \mathcal{C}} C = V$ and for each edge $(u, v) \in E$ there exists at least one cell $C \in \mathcal{C}$ such that $u, v \in C$.

For two vertices $u, v \in V$, $\delta(u, v)$ denotes the number of edges on the shortest path between u and v in G. For a subset of vertices $A \subseteq V$, $\delta(v, A) = \min_{a \in A} \delta(v, a)$. For a vertex v and a cell C, the subroutine Distances(v,C) determines $\delta(v, u)$ $\forall u \in C$ by issuing distance queries between v and every vertex in C.

Algorithm 1 Graph Reconstruction.

```
1: Function RECONSTRUCT(V):
2: E ← ∅, C ← ∅, W ← V
3: Let M denote the maximum size of cells, initially +∞
4: while M > g do   // g is a chosen constant
5:     a ← a random vertex from W
6:     W ← W \ {a}
7:     C ← cover(V, C, a)
8:     M ← max_{C∈C} {|C|}
9: for C ∈ C do
10:    E ← E ∪ EXHAUSTIVE-QUERY(C)
11: return E
```

3 Algorithm

The first component of our algorithm is RECONSTRUCT(V), which takes as input the set of vertices V of the input graph and returns the reconstructed graph with the correct edge assignments. We start by choosing a constant, g, which is the threshold value for the maximum sized cell in our cover. In a loop, we randomly select an unselected vertex to be the center for the new cell, and call COVER$(V, \mathcal{C}, a)$ to get a new cover which includes the new cell with center a. We describe how COVER$(V, \mathcal{C}, a)$ works later in this section.

We keep performing this loop until the maximum sized cell in the cover becomes less than g, in which case we terminate the loop and perform an exhaustive search on each cell of the cover. The function EXHAUSTIVE-QUERY(C) takes as input a cell C and returns all edges between vertices in C by issuing distance queries for each pair of vertices in C. We provide details in Algorithm 1.

The second and main component of our algorithm is COVER$(V, \mathcal{C}, a)$ (see Algorithm 2), which takes as input the set of vertices V, a set of cells $\mathcal{C}$ and a vertex a, and returns a new cover where a is the center of a new cell N. We define S, which we call the *frontier*, to be the set of cells that we should search in expanding N, and we initialize it with the cells that a belongs to. The only exception is when we first call COVER$(V, \mathcal{C}, a)$, in which case we initialize S to be $\{V\}$ (see lines 3-6). Then, an arbitrary cell, C, from S is chosen, and we issue distance queries between a and all of the vertices of C.

Using the results from these queries, we determine which vertices in C are close to a, compared to their distances to all the other centers. We define A to be a global variable that stores the set of all centers that were added before the new center a. For a vertex $v \in C$, if $\delta(a, v) \leq \delta(A, v) - 1$, we remove v from all of the cells that contains it (see line 18). If, however, $\delta(a, v) = \delta(A, v)$ or $\delta(a, v) = \delta(A, v) + 1$, we consider v to be on the *boundary* between C and N and so we do not remove v from any cells. In both cases, we add v to the new cell N, and we add any unvisited cells that contain v to S since they might have vertices that are close to a as well.

We say that a cell $C_2 \in \mathcal{C}$ is a *neighbor* of $C_1 \in \mathcal{C}$ if $C_1 \cap C_2 \neq \emptyset$. In other words, two cells are neighbors if there exists a boundary vertex that belongs to both of them. So, each iteration of COVER$(V, \mathcal{C}, a)$ ends up adding to the frontier all unvisited neighbors of the current cell $C \in S$ that share at least one boundary vertex with C such that this boundary vertex can be added to N according to the closeness definition in line 15. Note that we do

Algorithm 2 COVER$(V, \mathcal{C}, a)$ algorithm for constructing a new cover after adding a cell centered at vertex a.

```
1: N ← {a}   // N is the new cell centered at a
2: L ← ∅   // L is the set of cells that have been visited
3: if C = ∅ then
4:    S ← {V}
5: else
6:    S ← {C ∈ C | a ∈ C}   // S is the set of cells that we should search in expanding N
7: while S ≠ ∅ do
8:    C ← an arbitrary cell from S
9:    S ← S \ {C}
10:   L ← L ∪ C
11:   Distances(a, C)
12:   // A is a global variable denoting the set of all cell centers
13:   If A = ∅, ∀v ∈ V: set δ(A, v) = +∞.
14:   for v ∈ C do
15:      if δ(a, v) ≤ δ(A, v) + 1 then
16:         S ← S ∪ {C' ∈ C | v ∈ C' and C' ∉ L}   // add all of the unvisited cells that
            contain v to the frontier S
17:         if δ(a, v) ≤ δ(A, v) − 1 then
18:            Remove v from all the cells that contain it
19:         N ← N ∪ {v}
20: A ← A ∪ {a}
21: return C ∪ {N}
```

not necessarily add all the neighbors of C to the frontier: if none of the boundary vertices v between C and a neighboring cell N' have distance at most $\delta(A, v) + 1$ to N's center, then it is clear that none of the vertices in N' can be added to N.

4 Correctness and Analysis

▶ **Theorem 1.** *For any undirected, unweighted and connected graph $G = (V, E)$,* RECONSTRUCT(V) *correctly reconstructs E.*

Proof. We use an inductive argument to prove that the union of exhaustive searches performed on the cells created by the algorithm discovers all $(u, v) \in E$.

Initially, there is a single cell containing all of the vertices V, which trivially covers all the edges of E. Now, let A_i represent the first i centers that we add in the algorithm and assume, at every step $2 \leq s \leq i$, that for each edge $(u, v) \in E$ there is a cell with its center in A_s that contains both u and v. We then prove that if we create a new cell N, centered at the $(i + 1)$-th center $a \in A_{i+1}$, the union of the new cells still covers all the edges in E.

Consider an edge $(e_1, e_2) \in E$. Let x be the last center among the first $i + 1$ centers such that $x = \text{argmin}_{a \in A_{i+1}}\{\min(\delta(a, e_1), \delta(a, e_2))\}$. In other words, x is the last center that is closest to either endpoint of the edge (e_1, e_2). If $\delta(x, e_1) \neq \delta(x, e_2)$, we denote the endpoint that is closer to x as v, and denote the other endpoint as u. Otherwise, we denote the endpoints arbitrarily as u and v. So, we have $\min(\delta(x, e_1), \delta(x, e_2)) = \delta(x, v) = \delta(A_{i+1}, v) \leq \delta(A_{i+1}, u)$. We prove that both u and v belong to the cell centered at x, after the $(i + 1)$-th iteration.

First, we prove that we add both v and u to the cell at x. Let $(s_1, s_2, \ldots, s_m)$ denote the ordered vertices on a shortest path from x to v, where $s_1 = x$ and $s_m = v$. Using the inductive hypothesis for each $2 \leq j \leq m$, there exists a cell that contains both s_{j-1} and s_j right before adding center x. Now, consider the smallest j such that s_j is not added to the cell at x during the loop at line 14. Since s_{j-1} is added to the cell at x, and since s_{j-1} and s_j are connected, then by the inductive hypothesis there is a cell C that contains both s_{j-1} and s_j. Therefore, when we add s_{j-1} to the cell at x, we add C to the set of cells that we should explore in expanding the cell centered at x (see line 16). On the other hand, since s_j is on the shortest path from x to v and $\delta(x, v) = \delta(A_{i+1}, v)$, then $\delta(x, s_j) = \delta(A_{i+1}, s_j)$. Therefore, s_j will be added to the cell at x when exploring cell C. Using this inductive approach, all vertices on the shortest path from x to v will be added to the cell at x. Finally, since $\delta(x, u) \leq \delta(x, v) + 1 = \delta(A_{i+1}, v) + 1 \leq \delta(A_{i+1}, u) + 1$, and since u and v also have a common cell, we add u to the cell at x.

Next, we prove that if we add v and u to the cell centered at x, no other cells that we create later on in the first $(i+1)$-th steps removes v or u from the cell at x. Note that for removing a node from cell x, the condition at line 17 must hold. Since $\delta(x, v) = \delta(A_{i+1}, v)$, there will be no center b among the first $(i+1)$ centers such that $\delta(b, v) \leq \delta(A_{i+1}, v) - 1$, meaning that v will stay in cell at x. On the other hand, we remove u from x only if for a center b: $\delta(b, u) \leq \delta(x, u) - 1$. If $\delta(b, u) \leq \delta(x, u) - 1$, and given the fact that $\delta(x, u) \leq \delta(x, v) + 1$ and $\delta(x, v) \leq \delta(x, u)$, then $\delta(b, u) \leq \delta(x, v) \leq \delta(x, u)$. However, we assumed that x is the last center, among the first $i + 1$ centers, that is closest to either of the endpoints u and v. Therefore, u will also stay in the cell at x. ◀

▶ **Theorem 2.** *The expected query complexity of* RECONSTRUCT(V) *is* $O(d_{\max}^2 n \log n)$*, where* $d_{\max}$ *is the maximum cell degree over all steps.*

Proof. We use a backwards analysis [34] to derive an expression for the expected query complexity of the algorithm. We assume i centers have already been added, and analyze the expected number of queries we issue at step i.

We observe that our algorithm only issues distance queries for cells in the set S. Moreover, the only cells we add to S are the ones that contain vertices that get added to the ith cell. This means that all cells in S will become neighbors of the ith cell at the end of step i. So the number of distance queries issued at step i is the sum of the sizes of each cell that gets added to S, which is at most the sum of the sizes of the ith cell and its neighbors at the end of step i. Denoting the set of cells at the end of step i as $\mathcal{C}_i$, and the set of cells neighboring any cell C as $N(C)$, we have that the expected number of queries issued at the ith step is

$$\leq \sum_{C \in \mathcal{C}_i} \frac{1}{i} (|C| + \sum_{C' \in N(C)} |C'|)$$
$$= \sum_{C \in \mathcal{C}_i} \frac{(d(C)+1)|C|}{i},$$

by observing that each cell size $|C|$ is summed $d(C) + 1$ times, where $d(C)$ denotes the *degree* of cell C, i.e. the number of neighboring cells it has. To bound this summation, we express each cell size as the sum of boundary and non-boundary vertices. We have

$$\sum_{C \in \mathcal{C}_i} \frac{(d(C)+1)|C|}{i} = \sum_{C \in \mathcal{C}_i} \frac{(d(C)+1)(|C|_{NB} + |C|_B)}{i}$$
$$\leq \frac{(d_{\max}+1)}{i} \left((\sum_{C \in \mathcal{C}_i} |C|_{NB}) + (\sum_{C \in \mathcal{C}_i} |C|_B) \right)$$

where $d_{\max}$ is the maximum degree of any cell during any step, $|\cdot|_B$ denotes the number of boundary vertices, and $|\cdot|_{NB}$ denotes the number of non-boundary vertices. We use the fact that $\sum_{C\in\mathcal{C}_i}|C|_{NB} \leq n$, and observe that $\sum_{C\in\mathcal{C}_i}|C|_B \leq (d_{\max}+1)\cdot n$ as each boundary vertex can belong to at most $d_{\max}+1$ cells, and thus can only be counted that many times at most in the summation. So, we have

$$\frac{(d_{\max}+1)}{i}\left((\sum_{C\in\mathcal{C}_i}|C|_{NB})+(\sum_{C\in\mathcal{C}_i}|C|_B)\right) < \frac{n(d_{\max}+2)^2}{i}.$$

The expected number of queries when all steps are considered is

$$< \sum_{i=1}^{\#\text{steps}} \frac{n(d_{\max}+2)^2}{i} = n(d_{\max}+2)^2 \sum_{i=1}^{\#\text{steps}} \frac{1}{i},$$

which is $O(d_{\max}^2 n \log n)$. To finish our analysis, we also need to consider the number of queries issued during the exhaustive searches in each cell. Since the total number of cells is $O(n)$, and each cell is of size at most a constant g, the exhaustive querying part has query complexity $O(n)$. ◀

5 Experimental results

5.1 Implementation and Datasets

We implemented our algorithm in C++, and simulated the distance query oracle by performing BFS in each iteration to compute distances between nodes while keeping track of how many distance queries would be necessary to find these distances. We selected the value of g to be 50. We include experimental results for road networks from 50 U.S. states and Washington, D.C. obtained from the formatted TIGER/Line dataset available from the 9th DIMACS Implementation Challenge website[1] and road networks from Belgium, the U.K. (limited to the road network of Great Britain), Italy, Luxembourg, and the Netherlands obtained from formatted OpenStreetMaps data available from the 10th DIMACS Implementation Challenge [12]. For all of the datasets, only the largest connected component is considered. In Section 6, we discuss how the upper bounds derived from these experiments compare to our theoretical upper bound.

5.2 Batch Length

We define the *batch length* of a step to be the number of distance queries issued at that step. To find the relation between batch length and the step number, based on the theoretical upper bound we derived in Section 4, we fit the function $\textsc{Batch-Length}(step) = a + b\frac{n}{\text{step}}$ to the data points in our results, where a and b are the fitting parameters. Data points for the batch lengths of some of our datasets are provided in Figure 1. We list our results for all of the datasets in Table 1, which includes the best-fit parameters in columns a and b, and the maximum number of cells visited at any step in column M.

We can see that parameter b does not exceed 2, and that parameter a is close to 0 for all of the datasets. This suggests that a constant or logarithmic factor of $\frac{n}{\text{step}}$ could be an upper bound for the batch size at any step, which leads us to predict an upper bound of $\log n \cdot \frac{n}{\text{step}}$ which we show in Figure 1. We report the percentage of steps that fall below this upper bound for each dataset in Table 1, column U.

[1] http://www.diag.uniroma1.it/~challenge9/data/tiger/

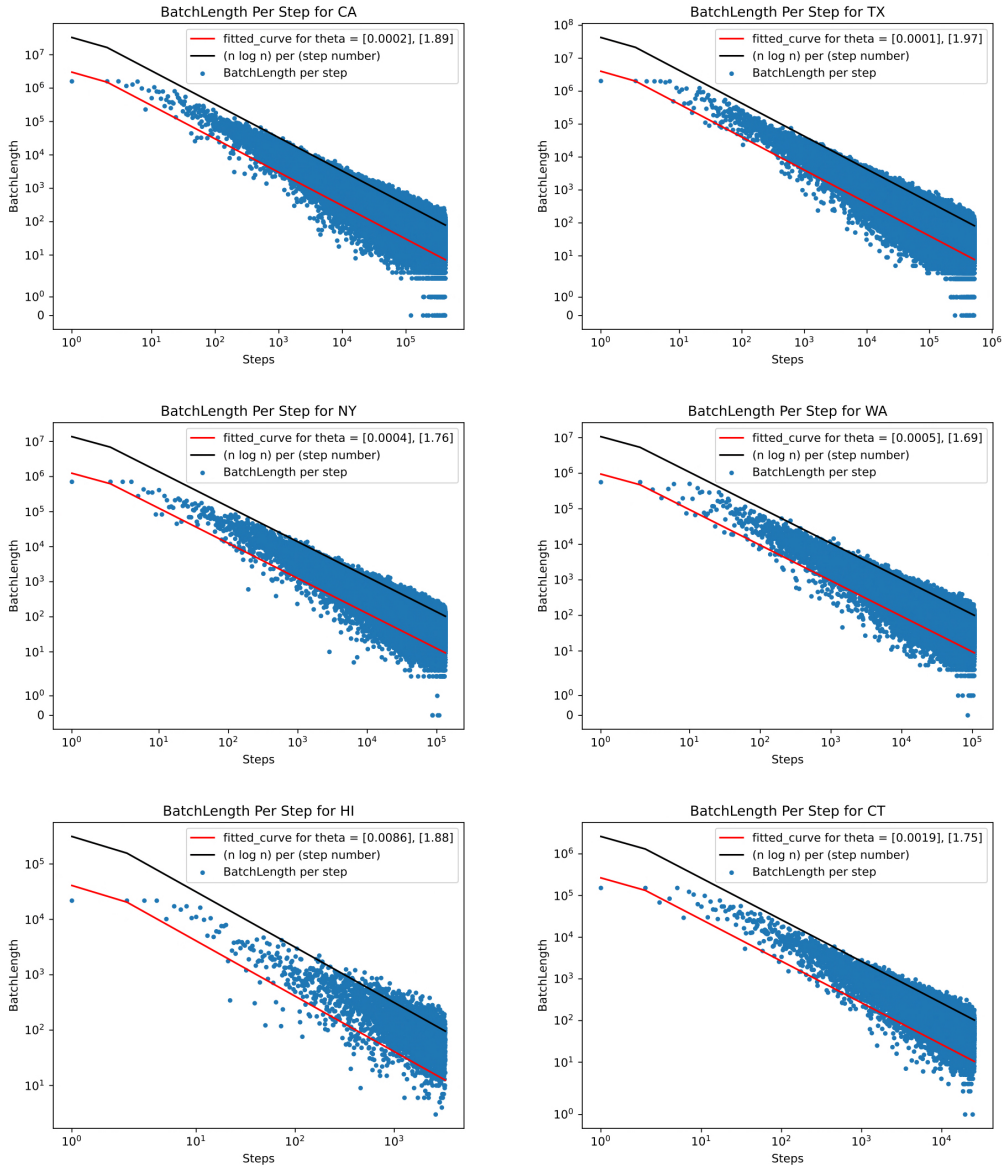


Figure 1 Batch lengths for select datasets of varying sizes.

Table 1 Batch length results for all datasets. Columns a, b and U were rounded to 4, 2 and 2 decimal places respectively.
a, b: best-fit parameters for BATCH-LENGTH($step\#$) $= a + b\frac{n}{step\#}$
U: percentage of batch lengths that are below the upper bound of $\frac{n}{step\#} \log n$.

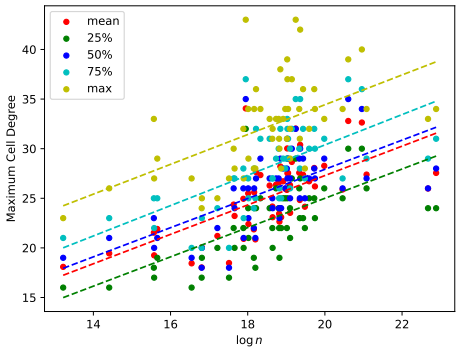
Dataset	n	a	b	U	Dataset	n	a	b	U
AK	48 560	0.0036	1.69	96%	ND	203 583	0.0015	1.76	92%
AL	561 459	0.0005	1.91	98%	NE	304 335	0.0011	1.99	93%
AR	478 024	0.0005	1.85	98%	NH	115 055	0.0023	1.91	97%
AZ	533 008	0.0005	1.95	95%	NJ	329 404	0.0010	1.90	94%
CA	1 595 577	0.0002	1.89	96%	NM	456 896	0.0006	1.88	97%
CO	436 084	0.0006	1.91	96%	NV	253 012	0.0009	1.85	94%
CT	152 036	0.0019	1.75	94%	NY	708 520	0.0004	1.76	97%
DC	9522	0.0321	1.89	70%	OH	672 527	0.0005	1.93	95%
DE	48 812	0.0053	1.77	91%	OK	535 032	0.0006	1.87	96%
FL	1 036 647	0.0003	1.86	97%	OR	529 702	0.0005	1.90	98%
GA	731 954	0.0004	1.97	97%	PA	866 352	0.0004	1.83	97%
HI	21 774	0.0086	1.88	90%	RI	51 642	0.0047	1.88	92%
IA	388 487	0.0008	1.92	93%	SC	460 763	0.0005	1.81	97%
ID	265 552	0.0010	1.81	97%	SD	206 998	0.0014	1.85	94%
IL	790 439	0.0004	1.89	96%	TN	578 981	0.0004	1.70	98%
IN	495 581	0.0007	1.88	94%	TX	2 037 156	0.0001	1.97	97%
KS	471 066	0.0007	1.87	93%	UT	242 432	0.0010	1.95	96%
KY	463 542	0.0006	1.90	98%	VA	620 680	0.0004	1.92	98%
LA	408 161	0.0006	1.84	97%	VT	95 672	0.0022	1.95	98%
MA	294 345	0.0009	1.98	95%	WA	560 336	0.0005	1.69	97%
MD	264 378	0.0010	1.86	95%	WI	514 687	0.0006	1.89	96%
ME	187 315	0.0013	1.81	99%	WV	292 557	0.0006	1.89	99%
MI	661 718	0.0005	1.74	95%	WY	243 545	0.0010	1.92	97%
MN	541 166	0.0006	1.76	95%	BE	1 441 295	0.0002	2.00	99%
MO	668 322	0.0004	1.89	97%	GB	7 733 822	0	1.81	100%
MS	409 994	0.0007	1.93	98%	IT	6 686 493	0	1.81	100%
MT	300 809	0.0007	1.80	98%	LU	114 599	0.0019	1.96	99%
NC	876 954	0.0003	1.72	99%	NL	2 216 688	0.0001	1.86	99%

5.3 Maximum Cell Degree

To combine our experimental results with our theoretical upper bound, we collected data on the maximum cell degree at each step. We combine the step-wise data in each dataset using different measures: mean, max, and the 1st, 2nd and 3rd quartiles to see how the data is spread. Then for each dataset, we represent the value corresponding to each measure as a point. Based on our intuition, we fit the function $a \log n + b$ for each measure. We list our results in Figure 2, which includes the best-fit parameters for each measure. We can see that $a < 2$, and b is a small constant for each measure. The datasets with the largest maximum cell degrees turned out to be VA, NV and OH, with values of 43, 43 and 42 respectively. It is clear from the figure that a small constant multiple of $\log n$ would be enough to produce an upper bound that covers all of the data points, suggesting that the maximum cell degree might have an upper bound of $O(\log n)$.

5.3.1 Road Networks with Subdivided Edges

We provide experimental results in Table 2 for the maximum cell degrees of the weighted road networks of the District of Columbia and the state of Hawaii, which we transform into unweighted graphs by replacing each edge e with $\lceil w(e) \rceil$ edges, where $w(e)$ is the weight of e. The size of each road network increased by factors of approximately 192 and 167 respectively, while the maximum cell degree values ended up decreasing for both road networks. This indicates that our algorithm can also perform efficiently on weighted road networks.



(a) Best-fit lines for the function $a + b \log n$.

Measure	a	b
mean	-2.34	1.48
1st quartile	-4.52	1.48
2nd quartile	-1.49	1.47
3rd quartile	-0.2	1.53
max	4.4	1.50

(b) Parameters for best-fit lines. Columns a and b were rounded to 2 decimal places.

Figure 2 Results from combining step-wise maximum cell degrees.

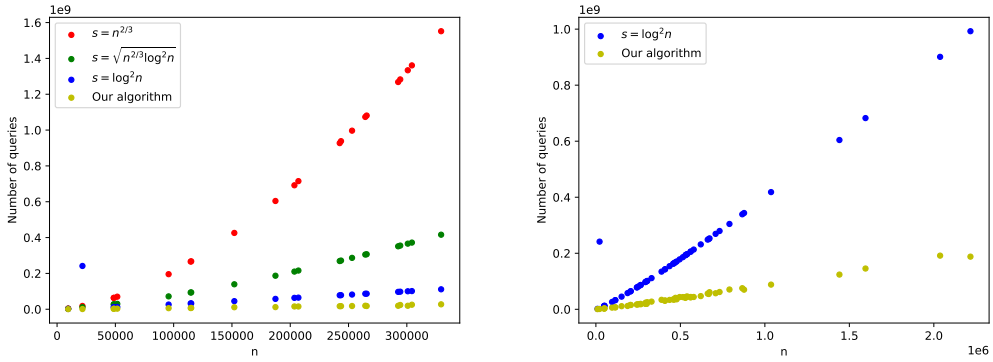


Figure 3 Number of queries issued by our algorithm compared to [31].

5.4 Comparisons with Existing Algorithms

We directly compare the number of queries issued by our algorithm to the algorithm introduced in [31], which takes as input a parameter s that affects the query complexity. The authors prove their query complexity bounds for Δ-regular graphs and bounded graphs with the value of s being set to $\log^2 n$ and $n^{2/3}$ respectively. We use these values for s in our experiments, and we also try setting s to be the geometric mean of these values. We summarize the results of our experiments on road networks in Figure 3.

We then compare our results to the number of queries issued by the algorithm in [29]. Without performing any experiments, it can be observed that this algorithm will issue significantly more queries than our algorithm: the first iteration of Estimated-Centers (Algorithm 2 in [29]) will issue at least $\Omega(n \cdot \Delta\sqrt{n} \cdot \log n \cdot \log\log n)$ distance queries.

Table 2 Maximum cell degrees for weighted road networks compared to their unweighted versions.

	Unweighted		Weighted	
	$\lvert V\rvert$	$d_{\max}$	$\lvert V\rvert$	$d_{\max}$
DC	9522	23	1 826 049	12
HI	21 774	26	3 643 818	11

6 Comparison of Theoretical/Experimental Results and Future Work

The theoretical upper bound we derived in Section 4 contains a $d_{\max}^2$ term, which can be $O(n^2)$ in the worst case. However, our experiments show that the maximum cell degree is actually low for road networks throughout the algorithm. From our results in Section 5.3, we can see that $O(\log n)$ would be a suitable upper bound for $d_{\max}$. In this case, the expected query complexity of our algorithm would be $O(n \log^3 n)$. The experimental results for batch length support this observation, as the upper bound for the batch length at any step number amounted to be $\frac{n \log n}{step\#}$, from which we get an expected query complexity of $O(n \operatorname{polylog}(n))$ as well. Future work might involve trying to find if there exists a better theoretical upper bound on the query complexity of our algorithm. This might require making some additional simplifying assumptions about the graphs being used as input.

We would like to point out a connection between our results and the graph-theoretical Delaunay triangulation of road networks.

6.1 Delaunay Triangulations and $d_{\max}$

We can consider the cells (resp. covers) that are constructed during our algorithm as a redefinition of graph-theoretical Voronoi cells (resp. diagrams) (e.g., see [23]). Similarly, we can consider the dual graph connecting neighboring cells in the cover as being a form of a graph-theoretical Delaunay triangulation of G. There exists prior work on bounding the expected maximum degree of the Delaunay triangulation of a set of points selected randomly from the Euclidean plane. In [14], the authors consider the Delaunay triangulation of a Poisson point process limited to the portion of the triangulation within a cube of d dimensions. They show that the expected maximum degree of this triangulation is $\Theta(\log n / \log \log n)$. Having such a bound for the expected maximum degree for our redefinition of the graph-theoretical Delaunay triangulation might allow us to prove a theoretical quasilinear bound for the query complexity of our algorithm, so another interesting direction for future work can be to adapt this result for random point sets in Euclidean d-space to our setting, where the point set is selected randomly from the vertex set of a road network.

7 Conclusions

We introduced an efficient exact reconstruction algorithm for road networks and showed through experiments on several real-world road networks that our algorithm has an expected empirical query complexity that is quasilinear. As mentioned in Section 6, an important direction for future work can be to derive a theoretical upper bound for our algorithm that matches our experimental results.

References

1 Mikkel Abrahamsen, Greg Bodwin, Eva Rotenberg, and Morten Stöckel. Graph reconstruction with a betweenness oracle. In Nicolas Ollinger and Heribert Vollmer, editors, *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, volume 47 of *LIPIcs*, pages 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.STACS.2016.5.

2 Peyman Afshani, Manindra Agrawal, Benjamin Doerr, Carola Doerr, Kasper Green Larsen, and Kurt Mehlhorn. The query complexity of finding a hidden permutation. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 1–11, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40273-9_1.

3 Ramtin Afshar, Amihood Amir, Michael T. Goodrich, and Pedro Matias. Adaptive exact learning in a mixed-up world: Dealing with periodicity, errors and jumbled-index queries in string reconstruction. In Christina Boucher and Sharma V. Thankachan, editors, *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2020. doi:10.1007/978-3-030-59212-7_12.

4 Ramtin Afshar, Michael T. Goodrich, Pedro Matias, and Martha C. Osegueda. Reconstructing binary trees in parallel. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 491–492. ACM, 2020. doi:10.1145/3350755.3400229.

5 Ramtin Afshar, Michael T. Goodrich, Pedro Matias, and Martha C. Osegueda. Reconstructing biological and digital phylogenetic trees in parallel. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 3:1–3:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.3.

6 Ramtin Afshar, Michael T. Goodrich, Pedro Matias, and Martha C. Osegueda. Parallel network mapping algorithms. In Kunal Agrawal and Yossi Azar, editors, *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, pages 410–413. ACM, 2021. doi:10.1145/3409964.3461822.

7 Ramtin Afshar, Michael T. Goodrich, Pedro Matias, and Martha C. Osegueda. Mapping networks via parallel kth-hop traceroute queries. In Petra Berenbrink and Benjamin Monmege, editors, *39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022, March 15-18, 2022, Marseille, France (Virtual Conference)*, volume 219 of *LIPIcs*, pages 4:1–4:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.STACS.2022.4.

8 Noga Alon and Vera Asodi. Learning a hidden subgraph. *SIAM J. Discret. Math.*, 18(4):697–712, 2005. doi:10.1137/S0895480103431071.

9 Noga Alon, Richard Beigel, Simon Kasif, Steven Rudich, and Benny Sudakov. Learning a hidden matching. *SIAM J. Comput.*, 33(2):487–501, 2004. doi:10.1137/S0097539702420139.

10 Dana Angluin and Jiang Chen. Learning a hidden hypergraph. *J. Mach. Learn. Res.*, 7:2215–2236, 2006. URL: http://jmlr.org/papers/v7/angluin06a.html.

11 Dana Angluin and Jiang Chen. Learning a hidden graph using o(logn) queries per edge. *J. Comput. Syst. Sci.*, 74(4):546–556, 2008. doi:10.1016/j.jcss.2007.06.006.

12 David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013. doi:10.1090/conm/588.

13 Zuzana Beerliova, Felix Eberhard, Thomas Erlebach, Alexander Hall, Michael Hoffmann, Matús Mihalák, and L. Shankar Ram. Network discovery and verification. *IEEE J. Sel. Areas Commun.*, 24(12):2168–2181, 2006. doi:10.1109/JSAC.2006.884015.

14 Marshall W. Bern, David Eppstein, and F. Frances Yao. The expected extremes in a delaunay triangulation. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez-Artalejo, editors, *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings*, volume 510 of *Lecture Notes in Computer Science*, pages 674–685. Springer, 1991. doi:10.1007/3-540-54233-7_173.

15 Anna Bernasconi, Carsten Damm, and Igor Shparlinski. Circuit and decision tree complexity of some number theoretic problems. *Information and Computation*, 168(2):113–124, 2001. doi:10.1006/inco.2000.3017.

16 Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010. doi:10.1007/978-3-642-17373-8_33.

17 Sung-Soon Choi and Jeong Han Kim. Optimal query complexity bounds for finding graphs. *Artificial Intelligence*, 174(9):551–569, 2010. doi:10.1016/j.artint.2010.02.003.

18 Padraig Corcoran, Musfira Jilani, Peter Mooney, and Michela Bertolotto. Inferring semantics from geometry: the case of street networks. In Jie Bao, Christian Sengstock, Mohammed Eunus Ali, Yan Huang, Michael Gertz, Matthias Renz, and Jagan Sankaranarayanan, editors, *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, pages 42:1–42:10. ACM, 2015. doi:10.1145/2820783.2820822.

19 Luca Dall'Asta, J. Ignacio Alvarez-Hamelin, Alain Barrat, Alexei Vázquez, and Alessandro Vespignani. Exploring networks with traceroute-like probes: Theory and simulations. *Theor. Comput. Sci.*, 355(1):6–24, 2006. doi:10.1016/j.tcs.2005.12.009.

20 Shahar Dobzinski and Jan Vondrak. From query complexity to computational complexity. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 1107–1116, New York, NY, USA, 2012. ACM. doi:10.1145/2213977.2214076.

21 David Eppstein and Michael T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. *CoRR*, abs/0808.3694, 2008. arXiv:0808.3694.

22 David Eppstein and Siddharth Gupta. Crossing patterns in nonplanar road networks. In Erik G. Hoel, Shawn D. Newsam, Siva Ravada, Roberto Tamassia, and Goce Trajcevski, editors, *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2017, Redondo Beach, CA, USA, November 7-10, 2017*, pages 40:1–40:9. ACM, 2017. doi:10.1145/3139958.3139999.

23 Martin Erwig. The graph voronoi diagram with applications. *Networks*, 36(3):156–163, 2000. doi:10.1002/1097-0037(200010)36:3<156::AID-NET2>3.0.CO;2-L.

24 Stefan Funke, Robin Schirrmeister, and Sabine Storandt. Automatic extrapolation of missing road network data in openstreetmap. In Ioannis Katakis, François Schnitzler, Thomas Liebig, Dimitrios Gunopulos, Katharina Morik, Gennady L. Andrienko, and Shie Mannor, editors, *Proceedings of the 2nd International Workshop on Mining Urban Data co-located with 32nd International Conference on Machine Learning (ICML 2015), Lille, France, July 11th, 2015*, volume 1392 of *CEUR Workshop Proceedings*, pages 27–35. CEUR-WS.org, 2015. URL: http://ceur-ws.org/Vol-1392/paper-04.pdf.

25 Stefan Funke and Sabine Storandt. Provable efficiency of contraction hierarchies with randomized preprocessing. In Khaled M. Elbassioni and Kazuhisa Makino, editors, *Algorithms and Computation - 26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9-11, 2015, Proceedings*, volume 9472 of *Lecture Notes in Computer Science*, pages 479–490. Springer, 2015. doi:10.1007/978-3-662-48971-0_41.

26 Esha Ghosh, Seny Kamara, and Roberto Tamassia. Efficient graph encryption scheme for shortest path queries. In Jiannong Cao, Man Ho Au, Zhiqiang Lin, and Moti Yung, editors, *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, pages 516–525. ACM, 2021. doi:10.1145/3433210.3453099.

27 Vladimir Grebinski and Gregory Kucherov. Reconstructing a hamiltonian cycle by querying the graph: Application to DNA physical mapping. *Discret. Appl. Math.*, 88(1-3):147–165, 1998. doi:10.1016/S0166-218X(98)00070-5.

28 Vladimir Grebinski and Gregory Kucherov. Optimal reconstruction of graphs under the additive model. *Algorithmica*, 28(1):104–124, 2000. doi:10.1007/s004530010033.

29 Sampath Kannan, Claire Mathieu, and Hang Zhou. Graph reconstruction and verification. *ACM Trans. Algorithms*, 14(4), August 2018. doi:10.1145/3199606.

30 Valerie King, Li Zhang, and Yunhong Zhou. On the complexity of distance-based evolutionary tree reconstruction. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 444–453. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644179.

31 Claire Mathieu and Hang Zhou. A simple algorithm for graph reconstruction. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 68:1–68:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.68.

32 Lev Reyzin and Nikhil Srivastava. On the longest path algorithm for reconstructing trees from distance matrices. *Inf. Process. Lett.*, 101(3):98–100, 2007. doi:10.1016/j.ipl.2006.08.013.

33 Guozhen Rong, Wenjun Li, Yongjie Yang, and Jianxin Wang. Reconstruction and verification of chordal graphs with a distance oracle. *Theor. Comput. Sci.*, 859:48–56, 2021. doi:10.1016/j.tcs.2021.01.006.

34 Raimund Seidel. Backwards analysis of randomized geometric algorithms. In János Pach, editor, *New Trends in Discrete and Computational Geometry*, pages 37–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993. doi:10.1007/978-3-642-58043-7_3.

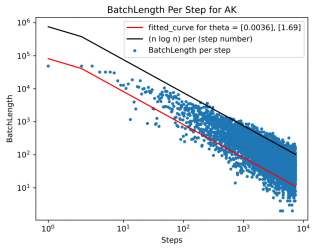
35 G. Tardos. Query complexity, or why is it difficult to separate $NP^A \cap coNP^A$ from P^A by random oracles A? *Combinatorica*, 9(4):385–392, December 1989. doi:10.1007/BF02125350.

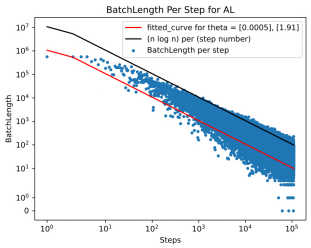
36 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005. doi:10.1145/1044731.1044732.

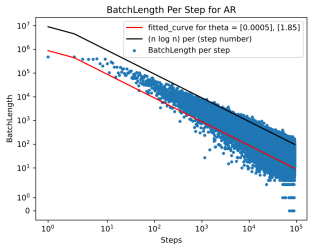
37 M.S. Waterman, T.F. Smith, M. Singh, and W.A. Beyer. Additive evolutionary trees. *Journal of Theoretical Biology*, 64(2):199–213, 1977. doi:10.1016/0022-5193(77)90351-4.

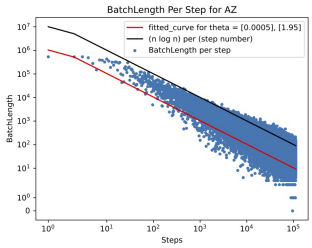
38 Andrew Chi-Chih Yao. Decision tree complexity and Betti numbers. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 615–624, New York, NY, USA, 1994. ACM. doi:10.1145/195058.195414.

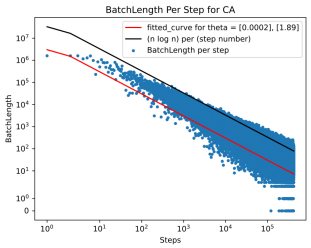
A Batch Length Results for All Datasets

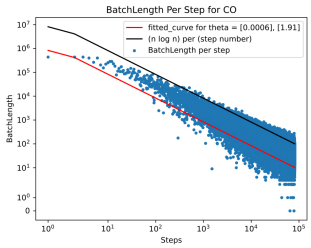


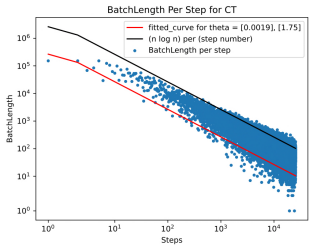


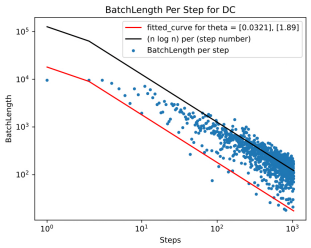


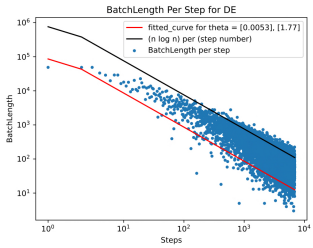


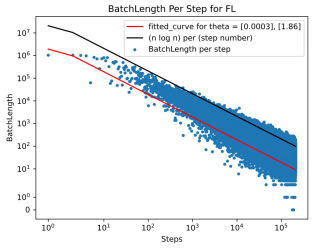


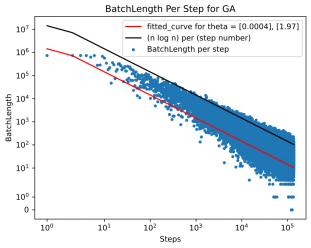


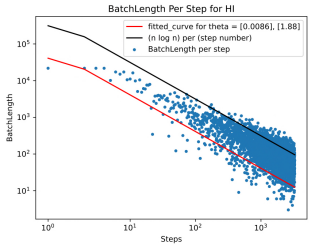


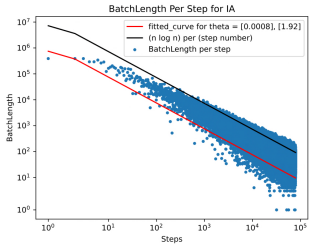


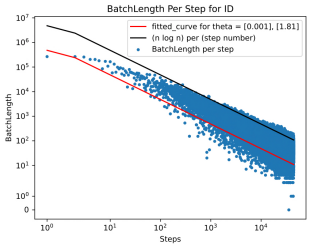


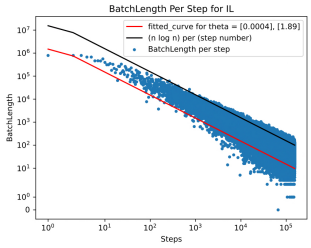


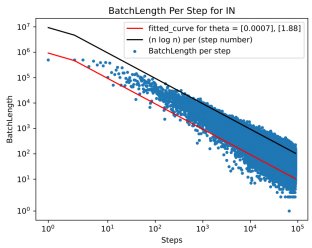


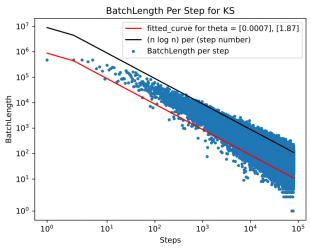


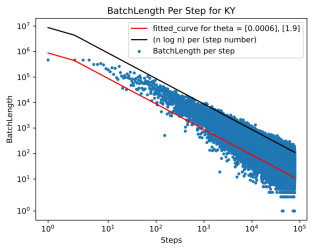


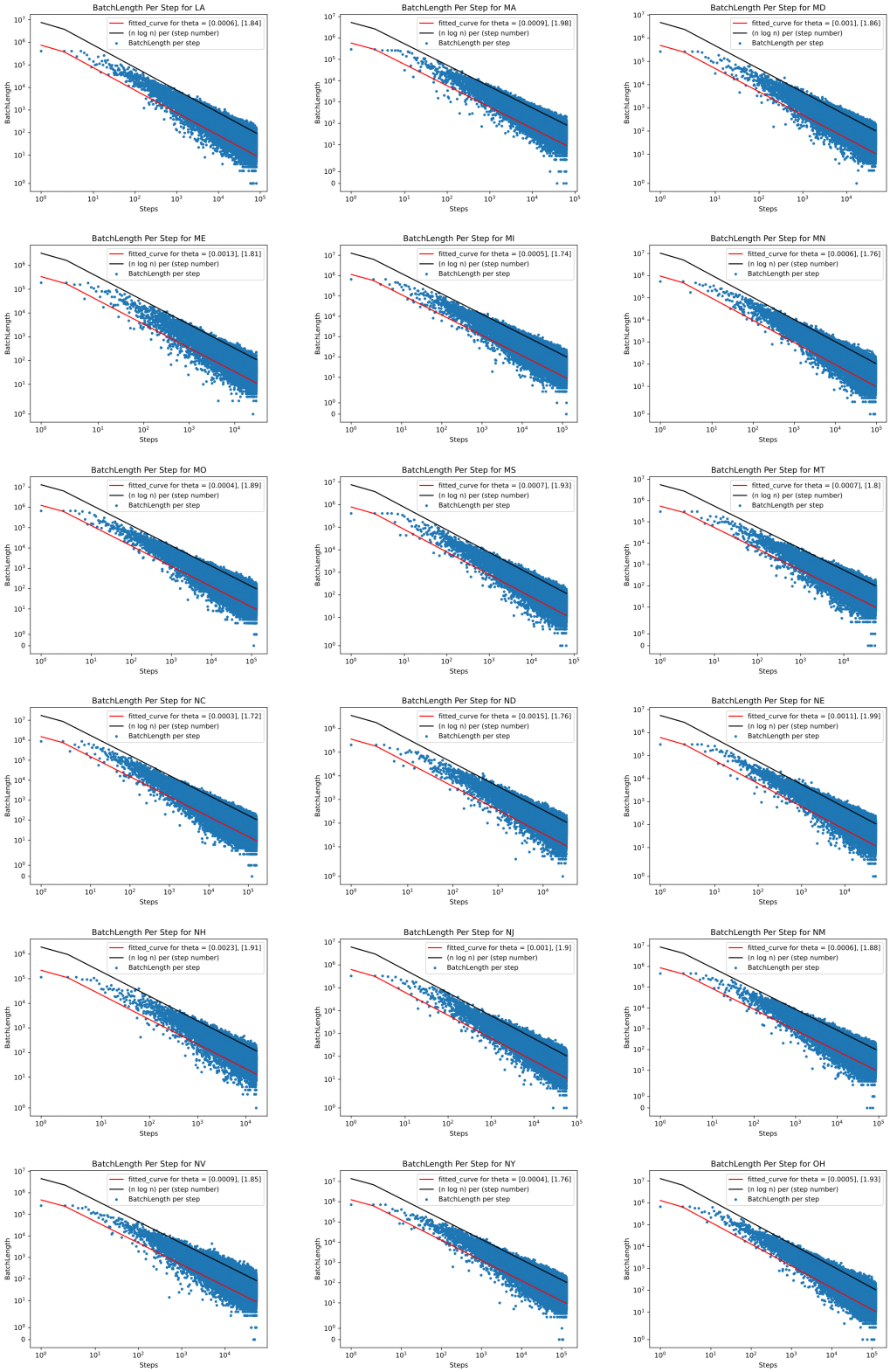


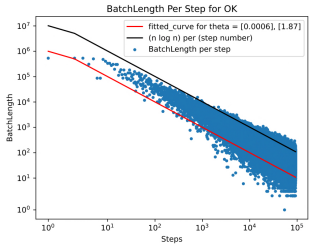


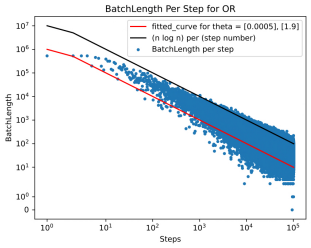


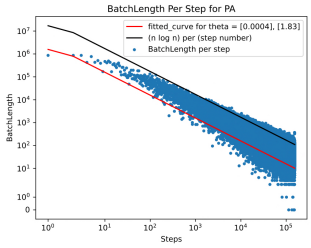


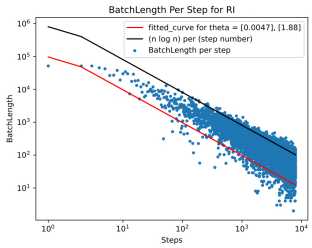


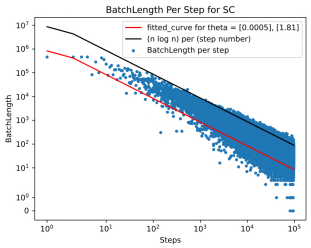

BatchLength Per Step for LA
fitted_curve for theta = [0.0006], [1.84]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for MA
fitted_curve for theta = [0.0009], [1.98]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for MD
fitted_curve for theta = [0.001], [1.86]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for ME
fitted_curve for theta = [0.0013], [1.81]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for MI
fitted_curve for theta = [0.0005], [1.74]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for MN
fitted_curve for theta = [0.0006], [1.76]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for MO
fitted_curve for theta = [0.0004], [1.89]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for MS
fitted_curve for theta = [0.0007], [1.93]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for MT
fitted_curve for theta = [0.0007], [1.8]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for NC
fitted_curve for theta = [0.0003], [1.72]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for ND
fitted_curve for theta = [0.0015], [1.76]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for NE
fitted_curve for theta = [0.0011], [1.99]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for NH
fitted_curve for theta = [0.0023], [1.91]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for NJ
fitted_curve for theta = [0.001], [1.9]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for NM
fitted_curve for theta = [0.0006], [1.88]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for NV
fitted_curve for theta = [0.0009], [1.85]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for NY
fitted_curve for theta = [0.0004], [1.76]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for OH
fitted_curve for theta = [0.0005], [1.93]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

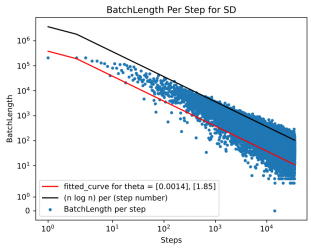

BatchLength Per Step for OK
fitted_curve for theta = [0.0006], [1.87]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

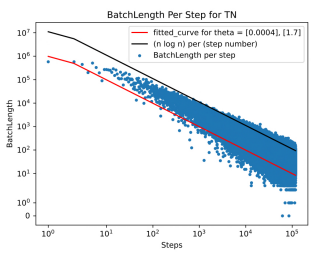

BatchLength Per Step for OR
fitted_curve for theta = [0.0005], [1.9]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

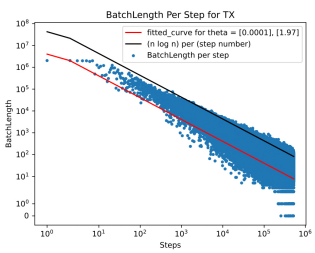

BatchLength Per Step for PA
fitted_curve for theta = [0.0004], [1.83]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

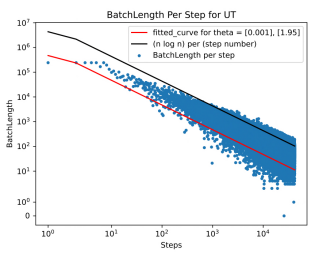

BatchLength Per Step for RI
fitted_curve for theta = [0.0047], [1.88]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

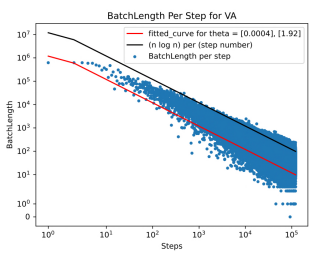

BatchLength Per Step for SC
fitted_curve for theta = [0.0005], [1.81]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

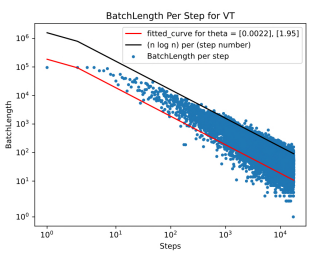

BatchLength Per Step for SD
fitted_curve for theta = [0.0014], [1.85]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

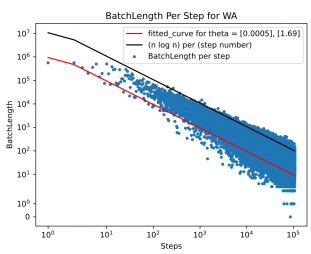

BatchLength Per Step for TN
fitted_curve for theta = [0.0004], [1.7]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

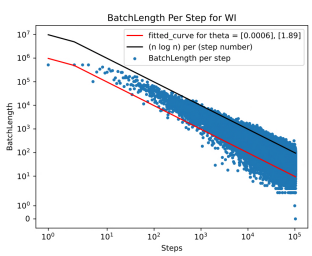

BatchLength Per Step for TX
fitted_curve for theta = [0.0001], [1.97]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

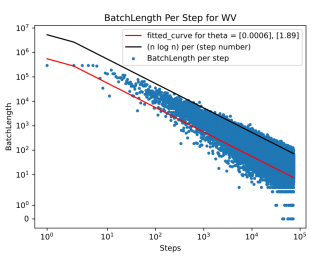

BatchLength Per Step for UT
fitted_curve for theta = [0.001], [1.95]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

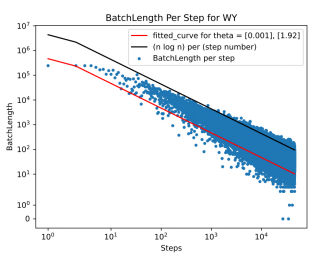

BatchLength Per Step for VA
fitted_curve for theta = [0.0004], [1.92]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

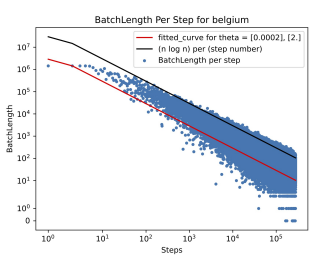

BatchLength Per Step for VT
fitted_curve for theta = [0.0022], [1.95]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

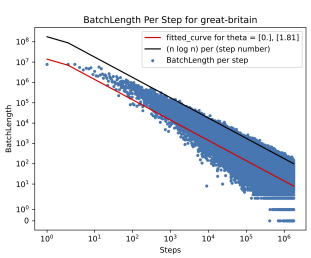

BatchLength Per Step for WA
fitted_curve for theta = [0.0005], [1.69]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

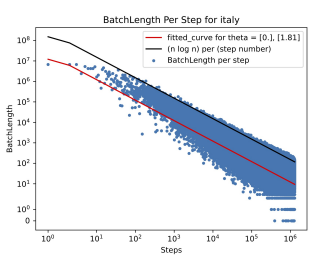

BatchLength Per Step for WI
fitted_curve for theta = [0.0006], [1.89]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

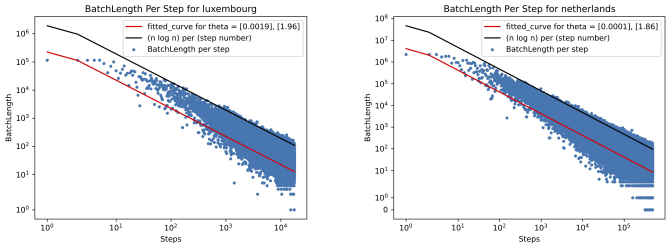

BatchLength Per Step for WV
fitted_curve for theta = [0.0006], [1.89]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps


BatchLength Per Step for WY
fitted_curve for theta = [0.001], [1.92]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps


BatchLength Per Step for belgium
fitted_curve for theta = [0.0002], [2.]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps


BatchLength Per Step for great-britain
fitted_curve for theta = [0.], [1.81]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps


BatchLength Per Step for italy
fitted_curve for theta = [0.], [1.81]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps


BatchLength Per Step for luxembourg
fitted_curve for theta = [0.0019], [1.96]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps
BatchLength Per Step for netherlands
fitted_curve for theta = [0.0001], [1.86]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

A Parallel Framework for Approximate Max-Dicut in Partitionable Graphs

Nico Bertram
Department of Computer Science, Technische Universität Dortmund, Germany

Jonas Ellert
Department of Computer Science, Technische Universität Dortmund, Germany

Johannes Fischer
Department of Computer Science, Technische Universität Dortmund, Germany

Abstract

Computing a maximum cut in *undirected* and weighted graphs is a well studied problem and has many practical solutions that also scale well in shared memory (despite its NP-completeness). For its counterpart in *directed* graphs, however, we are not aware of practical solutions that also utilize parallelism. We engineer a framework that computes a high quality approximate cut in directed and weighted graphs by using a graph partitioning approach. The general idea is to partition a graph into k subgraphs using a parallel partitioning algorithm of our choice (the first ingredient of our framework). Then, for each subgraph in parallel, we compute a cut using any polynomial time approximation algorithm (the second ingredient). In a final step, we merge the locally computed solutions using a high-quality or exact parallel Max-Dicut algorithm (the third ingredient). On graphs that can be partitioned well, the quality of the computed cut is significantly better than the best cut achieved by any linear time algorithm. This is particularly relevant for large graphs, where linear time algorithms used to be the only feasible option.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Design and analysis of algorithms

Keywords and phrases maximum directed cut, graph partitioning, algorithm engineering, approximation, parallel algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.10

Supplementary Material *Software (Source Code)*: `https://github.com/NicoBertram/par-max-dicut`

Funding This work has been supported by the German Research Association (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A6.

1 Introduction

A *directed* and *weighted graph* G is a tuple (V, E, w) with the set of *vertices* $V = \{1, ..., n\}$, the set of *edges* $E \subseteq V^2$ and nonnegative *weights* $w\colon V^2 \to \mathbb{R}_{\geq 0}$. If $(u, v) \notin E$, we set $w(u, v) = 0$. A *cut* in a directed and weighted graph $G = (V, E, w)$ is defined by a partitioning of V into two complementary subsets $S \subseteq V$ and $T = V \setminus S$. The value of a cut with respect to S and T is defined by $C(S, T) = \sum_{i \in S, j \in T} w(i, j)$, i.e. we sum the weights of the edges with origin in S and target in T. The maximum cut is then defined by $C_{max} = \max_{S \subseteq V, T = V \setminus S} C(S, T)$.

The problem of finding a maximum cut in a directed and weighted graph is denoted by Max-Dicut. It can be seen as a generalization of its well-studied counterpart Max-Cut in *undirected* graphs, which is one of the classical NP-complete problems listed by Karp [13]. In fact, Max-Dicut is at least as hard as Max-Cut, since every instance of Max-Cut can be trivially reduced to an instance of Max-Dicut (by replacing each undirected edge (u, v) with two directed edges (u, v) and (v, u) of the same weight). This reduction also shows that

Max-Dicut is NP-hard. Additionally, it is known that Max-Dicut is APX-hard, even when restricted to *directed acyclic graphs (DAGs)* [16]. Thus (assuming P $\neq$ NP), we cannot hope for more than a constant factor approximation algorithm.

One of the many practical applications of approximate Max-Dicut is as a subroutine in grammar-based text compression [12], where we encounter large graphs that may contain millions of nodes. In this setting, we are particularly interested in algorithms that not only obtain a good approximation ratio, but also utilize shared memory parallelism to accelerate the computation. To the best of our knowledge, there currently exists no such algorithm.

1.1 Related Work

Computing an *undirected* Max-Cut is a well-studied problem in theory as well as in practice. Among the practical solutions are a variety of exact Max-Cut solvers like BiqMac [20] and BiqCrunch [15]. However, these solvers do not use parallelism to accelerate the computation. Recently, new exact parallel solvers were introduced, e.g., BiqBin [9] and MADAM [11]. Unfortunately, these solvers cannot easily be modified to compute a *directed* maximum cut instead. They are also only practical for relatively small graphs.

There is a variety of sequential Max-Dicut approximation algorithms. A naive randomized algorithm assigns each node independently with probability $\frac{1}{2}$ either to S or T, which results in a cut with an expected performance guarantee of $\frac{1}{4}$. (As usual, we say that an algorithm has *performance guarantee* $\alpha \in (0, 1)$ if it always computes a cut of value at least $\alpha \cdot C_{max}$, where C_{max} is the value of a maximum cut.) By using the method of *conditional expectations* [22, 19], we can derandomize this algorithm. A linear time algorithm for unweighted graphs with a performance guarantee of $\frac{9}{20}$ was described in [10]. In [6], a set of algorithms with a performance guarantee in $[0.25, 0.5)$ was described. In these algorithms, the decision whether a node is assigned to S or T only depends on the in-degree and out-degree of the node. Max-Dicut can be seen as a maximization of a *submodular function*. In [4], a linear time algorithm with an expected performance guarantee of $\frac{1}{2}$ and one with deterministic ratio $\frac{1}{3}$ were described, both using maximization of a submodular function as their main ingredient. Max-Dicut was also considered in an online model [2] and shown to have an online algorithm with a performance guarantee of $\frac{1}{3}$.

By solving a relaxation of an integer linear program (ILP) and using a simple rounding scheme, we can achieve an expected performance guarantee of $\frac{1}{2}$ [18]. The currently best known performance guarantee uses an idea that was first described by Goemans and Williamson [7]. It relaxes an integer program into a *semidefinite program*, and then uses an interesting rounding technique to achieve a performance guarantee of 0.79607. The performance guarantee was subsequently improved to 0.859 [23] and 0.874 [17]. It has been shown that it can only be improved up to 0.878 in case that the *Unique Games Conjecture* [14] is true. These algorithms have a polynomial running time and do not perform well in practice. Also, they cannot easily be parallelized.

Summing up, when faced with graphs of nonhomeopathic size, the only practical option to date is to use one of the (randomized) linear time algorithms [4, 6], resulting possibly in poor quality directed cuts. Better performing algorithms such as [7] cannot be applied to dense graphs of more than a few thousand nodes.

1.2 Our Contributions

To bridge the gap between the linear time algorithms and the expensive ILP-based solutions, we propose a practical framework that computes high quality Max-Dicut approximations in well partitionable graphs using shared memory parallelism. In recent years, practical

improvements for a variety of graph problems were achieved by using graph partitioning [5, 1]. The general approach is to partition a graph into multiple subgraphs (usually aiming at minimizing the number of edges between the subgraphs), solve the problem locally on each subgraph, and then merge the local solutions into a global solution. Since we can independently compute a solution for each subgraph, this approach is well suited to be parallelized in shared (and potentially also distributed) memory.

This is also the idea behind our framework, which consists of three main algorithmic components and an optional post-processing step:

S1 A *graph partitioner* is used to split the input graph G into $k \ll n$ subgraphs G_i of roughly equal size, such that the dependency between the subgraphs is small, i.e. the sum of the edge-weights between the subgraphs is small.

S2 An *approximation algorithm for* Max-Dicut is used to compute a cut S_i, T_i for each subgraph G_i (processing up to p subgraphs simultaneously, using p processors).

S3 A *merger* is used to integrate all the local solutions S_i, T_i into a global cut. This is achieved by defining a new contracted graph with $2k$ nodes, where each node represents a partition set S_i or T_i. Merging the local cuts then corresponds to computing a Max-Dicut in the contracted graph. Thus, the merger is just another Max-Dicut algorithm, which can either be a high-quality approximation algorithm or even an exact solver, if k is sufficiently small.

S4 (optional) A *local search* is performed in order to improve the current solution by swapping nodes between the partitions until we cannot further increase the value of the cut.

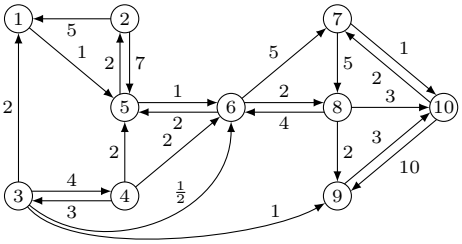
The C++ implementation of the framework is publicly available on GitHub (see supplementary material on the title page). At the time of writing, the framework provides three different partitioners, four sequential Max-Dicut approximation algorithms, and four mergers. It has been designed with extendability in mind, such that it should be little effort for the user to add new algorithms.

We evaluated the different options for each component on various real world graphs of up to 2^{23} nodes and 2^{24} edges. If we choose the right components, then the framework scales well in practice, while computing high quality cuts. The quality of the cut depends heavily on how well the graph can be partitioned in the first step. For graphs that can be partitioned well, we obtain a cut that is significantly better than the cut achieved by linear time algorithms. This is particularly relevant for large graphs, where linear time algorithms used to be the only feasible option.

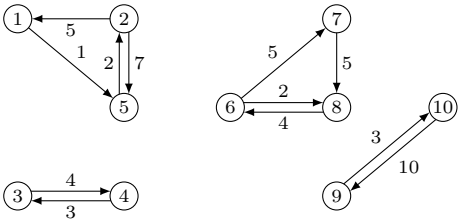
The remainder of the paper is structured as follows. First, we describe the main steps of the framework in more detail, providing examples of each step (Section 2). Then, we give implementation details, focusing on the actual algorithms that we use as components of the framework (Section 3.1). Finally, we provide practical results (Section 3.2), and discuss future work and open problems (Section 4).

2 Framework

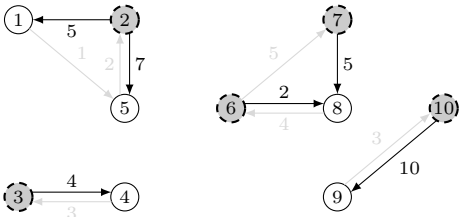
In this section, we describe the framework in more detail. The description is accompanied by a full example in Figure 1.



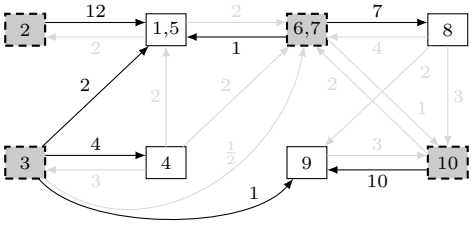
(a) The directed and weighted input graph G for which we want to compute a maximum directed cut.



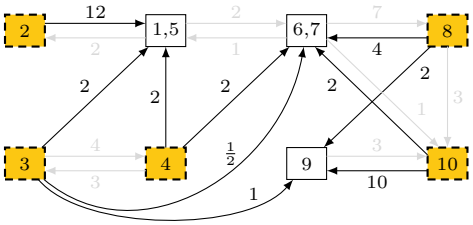
(b) Step 1: We partition the input graph into subgraphs G_1, G_2, G_3, G_4 induced by the node sets $V_1 = \{1, 2, 5\}$, $V_2 = \{6, 7, 8\}$, $V_3 = \{3, 4\}$ and $V_4 = \{9, 10\}$. The partition aims to minimize the sum of edge weights between subgraphs.



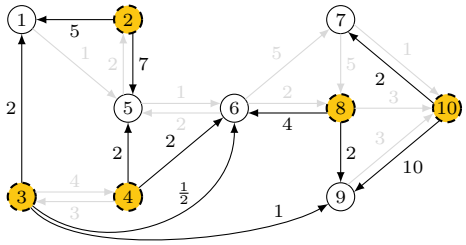
(c) Step 2: We compute a directed cut S_i, T_i on each G_i. The source nodes, i.e., the elements of any S_i, are filled and dashed.



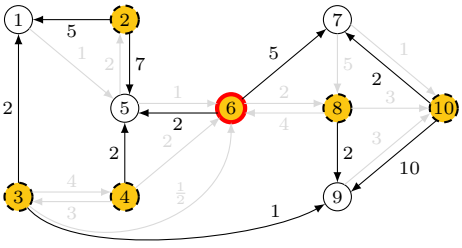
(d) Step 3.1: We obtain a contracted graph H from G by contracting each S_i and each T_i into a single node. The number of nodes is twice the number of subgraphs. The value of the naive cut $S = \bigcup S_i$, $T = V \setminus S$ is 37.



(e) Step 3.2: We compute an exact Max-Dicut S_H, T_H on H. Nodes from S_H are filled and dashed. The value of the cut is 37.5.



(f) Step 3.3: We obtain the final cut S, T on G by defining $S = \bigcup_{X \in S_H} X$ and $T = V \setminus S$. The unchanged value of the cut is 37.5.



(g) Step 4 (optional): We perform the local search and repeatedly identify nodes that have a positive gain, i.e., nodes that can be moved to the other partition set such that the overall cut improves. After Step 3.3, the only node with positive gain is node 6 with gain$(6) = (5 + 2) - (2 + \frac{1}{2} + 4) = \frac{1}{2}$. After moving node 6 from T to S (left part of the drawing), the cut has value 38. Now the only node with positive gain is node 4 with with gain$(4) = 4 - 2 = 2$. After moving it from S to T, (right part of the drawing), there is no node with positive gain, and the value of the cut is 40.

Figure 1 Running our framework on a small example.

2.1 Graph Partitioning

The first step of our framework is to partition the input graph $G = (V, E, w)$ into k disjoint subgraphs $G_1, \dots, G_k$ (where k is a freely choosable parameter), such that we can run on each subgraph independently an algorithm for approximating Max-Dicut. In order to improve the quality of the computed cut, we want to retain as much information as possible in each subgraph, while losing as little information as possible *between* the subgraphs. This can be achieved by maximizing the sum of the edge-weights in each subgraph, or, vice versa, minimizing the sum of the edge-weights between the subgraphs, i.e. we want to minimize $\sum_{i,j\in\{1,\dots,k\},i\neq j} E_{ij}$, where E_{ij} is the sum of the edge-weights between subgraph G_i and G_j.

We use partitioning algorithms that split G into subgraphs $G_i = (V_i, E_i, w)$ of roughly equal size, i.e. we allow for a multiplicative error $\epsilon > 0$ such that $|V_i| \leq (1+\epsilon)\left\lceil \frac{|V|}{k} \right\rceil$. Although graph partitioning is a hard problem itself, there are high-quality parallel graph partitioners that scale well on multiple processors [1, 8]. In Figures 1a and 1b, we see an example of an input graph partitioned into four subgraphs.

2.2 Compute Local Solutions

In the next step, we run on each subgraph G_i an algorithm that computes an approximation S_i, T_i for Max-Dicut. We process up to p subgraphs at the same time, where $p \leq k$ is the number of processors that we want to use. The choice of the approximation algorithm depends on the available time and the size of the subgraphs. If the subgraph are rather small, e.g., 100 nodes each, then we may be able to compute an optimal solution for each of them. If the subgraphs are slightly bigger, e.g., around 1000 nodes each, then a superlinear approximation algorithm with a relatively high performance guarantee might be feasible. If the subgraphs are even bigger, then we may have to use a simple linear time approximation algorithm with a lower performance guarantee. Depending on our choice of k, we have the flexibility to choose the algorithm such that we achieve the best trade-off between the quality of the cut and the runtime of the framework. As an example, we show in Figure 1c the optimal Max-Dicut for all subgraphs that were computed in Figure 1b.

2.3 Merging

In a final step, we have to merge the computed local cuts into a global cut. A naive approach is to define $S = \bigcup_i S_i$ and $T = \bigcup_i T_i$ as the trivial cut. The problem with this approach is that we did not consider the edges between the subgraphs. For some $i \in \{1, \dots, k\}$, it might be more advantageous to swap the subsets S_i and T_i in the global graph, or even to put S_i and T_i into the same partition. To consider the dependencies between the subgraphs, we reduce the problem of merging the local solutions to another Max-Dicut instance. We build a complete graph H with $2k$ nodes, where each node represents a locally computed partition set S_i or T_i. For every pair X, Y of nodes in H, we add an edge (X, Y) to H with weight $\sum_{i\in X, j\in Y} w(i, j)$ (see Figure 1d). Since the graph H has only $2k$ nodes, we can use an expensive algorithm to compute an exact Max-Dicut defined by S_H and T_H (see Figure 1e). Finally, the global cut is defined by $S = \bigcup_{X\in S_h} X$ and $T = \bigcup_{X\in T_h} X$ (see Figure 1f).

2.4 Optimization by Local Search

To further optimize the computed cut, we use in an optional fourth step, a *local search*. The idea of the local search is to check if we can improve the cut by swapping the partitioning of a node. We repeatedly swap the partitioning of nodes to improve the cut until we can no longer swap any nodes.

More precisely, for a node u we calculate the *gain* for swapping the partitioning of u. If $u \in S$, then we calculate the gain by $\text{gain}(u) = \sum_{(v,u)\in E, v\in S} w(v,u) - \sum_{(u,v)\in E, v\in T} w(u,v)$, and if $u \in T$ by $\text{gain}(u) = \sum_{(u,v)\in E, v\in T} w(u,v) - \sum_{(v,u)\in E, v\in S} w(v,u)$. We repeatedly find a node u with $\text{gain}(u) > 0$ and swap its partitioning, until for all nodes u we have $\text{gain}(u) \leq 0$. In this case, we have arrived at a local maximum. An example is provided in Figure 1g.

There are multiple ways of choosing a node u with $\text{gain}(u) > 0$. We use a simple *first candidate* strategy, which chooses the first node u with $\text{gain}(u) > 0$ that it encounters. We also implemented a *best candidate* strategy, which chooses the node u with maximal gain. However, in practice taking the best candidate rather than the first candidate improves the cut only by a small margin, while it significantly increases the running time.

We point out that this step is hard to parallelize, as multiple iterations of the local search are dependent of each other.

3 Experimental Evaluation

In this section, we present practical results of our framework. We implemented the framework in C++17 and used OpenMP for parallelization.

We evaluate our framework on the input graphs as summarized in Table 1. The graphs `great-britain`, `luxembourg`, `flixster`, `flickr`, `Stanford3`, and `web-sk-2005` were taken from *Network Repository* [21]. The graphs `luxembourg` and `great-britain` are road networks of Luxembourg and Great Britain, respectively. The graph `flixster` is a graph which represents all links between users of the website Flixster and the graph `flickr` is a graph where an edge represents if one user added another user as a contact on the website Flickr. The graph `Stanford3` is a Facebook graph with all users from Stanford University and the graph `web-sk-2005` is a crawl from 2005 for the .sk domain by using UbiCrawler [3]. For a comparison of our results with the optimal cut, we also took 121 small graphs from Network Repository on which we can compute the optimal cut in the time limit of one hour by doing three runs with different random seeds each (see Appendix A for a list).

To specify how well we can partition a weighted graph $G = (V, E, w)$, we additionally provide the *coverage* measure of a weighted graph for a partitioning $P = \{G_1, ..., G_k\}$. Intuitively, the coverage describes how many edges are covered by the partitioning. It is defined as $\text{cov}(G, P) = \frac{W(P)}{W}$, where $W(P) = \sum_{G_i=(V_i,E_i,w)\in P} \sum_{e\in E_i} w(e)$ and W is simply the sum of all edge-weights in G. The more the coverage of a graph tends towards 1, the less influence the merging step of our framework has on the computed cut. In case that G is clear from the context, we omit it in the notation. When we want to emphasize that there exists a partitioning P into k subgraphs with $\text{cov}(P) = x$, we also write $\text{cov}(k) = x$ (so x is a lower bound on the coverage of a graph with a partitioning into k subgraphs).

3.1 Implementation Details

For partitioning the input graph, we use the shared memory algorithm KaMinPar [8], which aims at minimizing the number of edges between subgraphs. To use this algorithm, we first have to transform our directed graph into an undirected graph by constructing a new graph $G' = (V, E', w')$ where $E' = \{(u,v) \mid (u,v) \in E \vee (v,u) \in E\}$ and $w'(u,v) = w(u,v) + w(v,u)$. We configured KaMinPar with an imbalance value of 0.01 and use a random seed for each computation.[1]

[1] We also tried two naive partitioning algorithms: simply slicing the list of nodes ore edges into equal-size parts, but, despite being fast, these turned out to produce cuts of inferior quality.

Table 1 A summary of our large input graphs. We provide for each graph the number of nodes n, the number of edges m and the coverage with four different partitionings. We partitioned the graph using KaMinPar into $k \in \{64, 256, 1024, 4096\}$ subgraphs and calculated for each the coverage measurement $\text{cov}(k)$ (in percent).

Graph	n	m	cov(64)	cov(256)	cov(1024)	cov(4096)
great-britain	7,733,822	16,313,034	99.96	99.92	99.82	99.58
flixster	2,523,386	15,837,602	55.26	49.26	44.41	39.83
flickr	513,969	6,380,904	69.78	53.68	37.14	27.50
luxembourg	114,599	239,332	99.62	99.10	97.64	92.83
web-sk-2005	121,422	668,838	99.59	98.45	93.98	77.07
Stanford3	11,586	1,136,618	27.76	15.56	6.20	1.69

We implemented several sequential algorithms that approximate Max-Dicut. As discussed earlier, they can be used both for computing the local solutions, as well as for merging them into the global solution. Our most naive implementations are the randomized algorithm that puts every node independently with probability $\frac{1}{2}$ either in S or T, and its derandomized counterpart. We call these $\frac{1}{4}$-approximation algorithms Random and Derandomized respectively. Additionally, we implemented the algorithm by Buchbinder et al. [4] with performance guarantee $\frac{1}{3}$, which we call Buchbinder. We also implemented the algorithm by Goemans and Williamson [7] with expected performance guarantee 0.79607, which we call Goemans. For solving semidefinite programs (which is needed as a subroutine of Goemans), we use the MOSEK Fusion API[2] with enabled parallelism. Lastly, we also implemented an algorithm that computes the exact Max-Dicut. This algorithm uses an *Integer Linear Program (ILP)* representation of Max-Dicut which was described in [10] and solves it using Gurobi[3] using only a single thread. We call this algorithm ILP.

To denote which of the sequential Max-Dicut algorithms we use in step 2, we use notations like Buchbinder_{S2}, meaning that we used the algorithm Buchbinder for computing the local solutions. (The merging step might still use other algorithms; see the following paragraph.)

When k gets large, the complete graph H as defined in Section 2.3 can get very large. For example, if we set k to 2048, then H has 4096 nodes and up to 16 million edges. In this case, the approximation algorithms with high performance guarantee are not practical anymore and the time to construct H can get too slow. To deal with this problem, we also implemented a tree-like merging algorithm. More precisely, we first divide the subgraphs $G_1, ..., G_k$ computed by the graph partitioner (with their local cuts) into $\frac{k}{l}$ groups of l subgraphs each (for some fixed $l < k$) where group i consists of the graphs $G_{(i-1)l+1}, ..., G_{il}$. Then, for each group we merge its subgraphs using the merge algorithm described in Section 2.3, resulting in $\frac{k}{l}$ subgraphs G'_i, each with a directed cut. We repeat this process with this new partitioning until the number of partitions gets smaller than l, where a final merging step produces the ultimate cut of G. We parallelized this algorithm with the exception of the Max-Dicut algorithm used to merge the directed cuts.

We denote this merge algorithm as Merge-Tree. In our experiments we use Merge-Tree with Goemans and $l = 256$ as our default configuration.

[2] https://www.mosek.com/
[3] https://www.gurobi.com/

3.2 Results

We now present experimental results on the running times and quality of our new algorithms in comparison with other existing approaches. When running an algorithm on a particular instance, we run it three times and for each execution we stop its execution after a time limit of 10 hours. Then, we take the average of the computed cuts and running times. This reflects the main research question that motivated our research: *Which quality can we achieve for directed cuts under given time and hardware constraints?*

We conducted our experiments on a Linux machine running Ubuntu version 18.04.6 with two AMD EPYC 7452 processors with 32 physical cores each and 1 TB of RAM. Thus when running the framework, we can compute the local cuts of up to $p = 64$ subgraphs at the same time. The code was compiled using GCC 7.5.0 with flag `-O3` enabled using Gurobi version 9.1.2 and Mosek version 9.3.

3.2.1 Overview

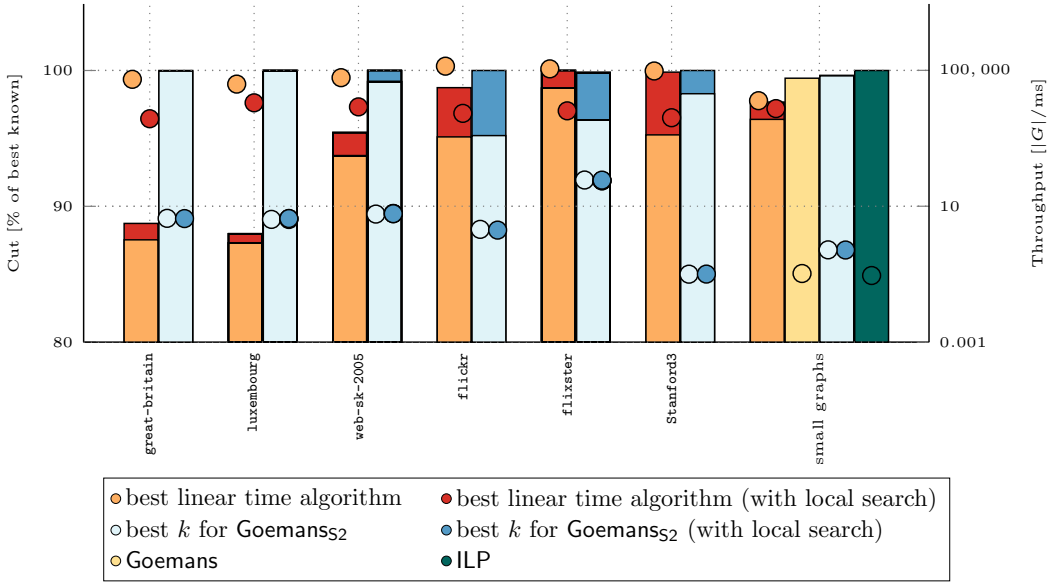


Figure 2 Overview of our results. For each algorithm, we visualize the quality of the cuts by a bar plot, and the throughput (size of the graph divided by the running time) by circles. The graphs from Table 1 are sorted in descending order by the value cov(64) to visualize the effect of the coverage on the computed cut quality. (Best viewed in color.)

Figure 2 provides an overview of what can be achieved with our framework. As we will see in Section 3.2.3 that the best performing configuration of our framework is to use the Goemans algorithm in step 2 (be it for varying k), we compare this configuration with other existing algorithms.

First look at the bar plots for the large graphs from Table 1, where currently the only feasible option is to use a linear time algorithm (Buchbinder, Derandomized, Random): we can observe that our new algorithms compute significantly better cuts than the best linear time algorithm on well partionable graphs which is indicated by their coverage. This is especially the case for `great-britain`, `luxembourg`, and `web-sk-2005`. Also, the computed cuts are

never worse for the other graphs (`flickr`, `flixster`, and `Stanford3`). We can also see that the final local search step (Section 2.4) pays off for all algorithms, in particular for the graphs that are not particularly well partitionable.

All this comes, of course, at the price of a lower throughput (cf. the circles in Figure 2), by several orders of magnitude. Nonetheless, we can conclude that within the given time and hardware constraints our new algorithms compute often better (and never worse) directed cuts than all previous existing approaches.

The final 4 bars show the cut quality of the small graphs in relation to the best possible cut. We calculated for each small graph the percentage and provide an (unweighted) average over the percentages of all small graphs since these values are independent of a graph. In addition to the linear time algorithms and our new parallel ones we could also run the Goemans algorithm on the entire graph, and the exact solution with ILP for each small graph. We see that our new algorithms again give better results than the best linear time algorithm with a final local search, and also slightly better results than Goemans. However, the difference between Goemans and our algorithms is not very large, which might be due to the fact that both of them are already very close to the exact solution (see final bar). On the positive side, our algorithms have a higher throughput than Goemans for the small graphs.

3.2.2 Scaling on Multiple Processors

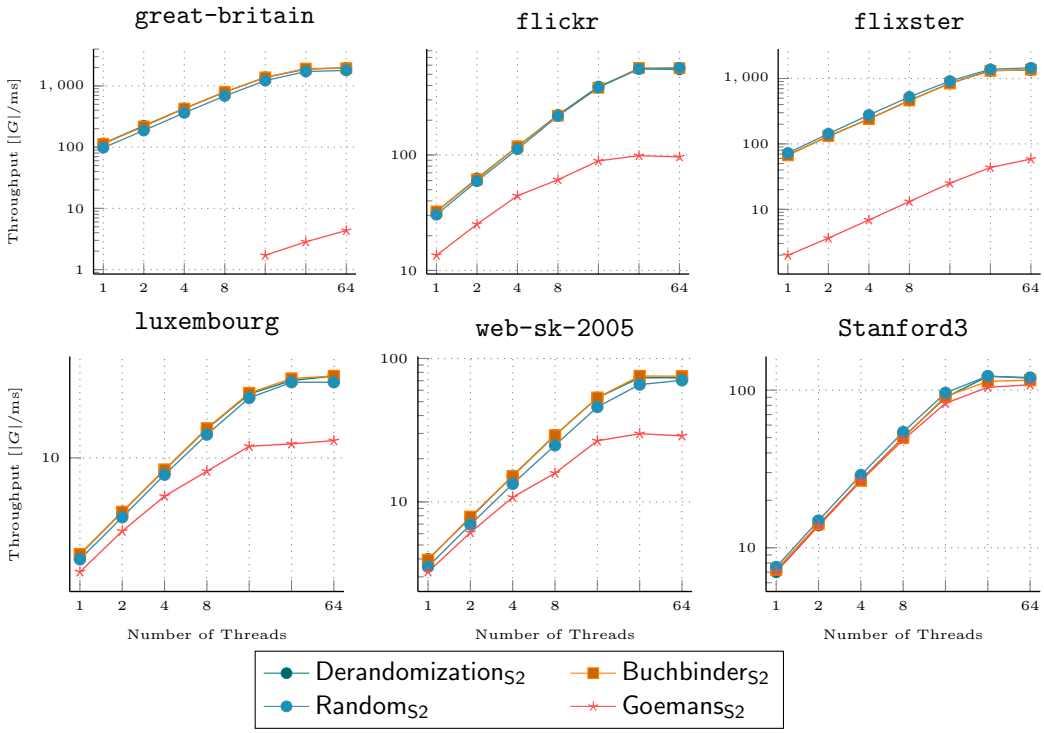


Figure 3 Scaling experiments on all graphs in Table 1. In all experiments, we partition the graph into 8192 subgraphs and run our framework with up to 64 threads. For this experiments we compare the algorithms Derandomized$_{S2}$, Buchbinder$_{S2}$, Random$_{S2}$ or Goemans$_{S2}$. On the x-axis we show the number of used threads and on the y-axis we show the throughput (size of graph divided by running time) of our framework.

Since our algorithms employ shared memory parallelism, we now briefly evaluate their scalability. Figure 3 shows the throughput for different choices of the approximation algorithms for the local solutions ($\mathsf{Derandomized}_{S2}$, $\mathsf{Buchbinder}_{S2}$, Random_{S2} or $\mathsf{Goemans}_{S2}$). For a better comparison, in all experiments we partition the graph into 8192 subgraphs and run our framework with up to 64 threads. We chose the value 8192 because our framework calculates for that value in all experiments a solution before the time limit is exceeded. We observe almost perfect scalability for up to 32 threads, with only slightly less good results for 32 threads, and significantly worse results for 64 threads. There are two possible explanations for the non-optimal scaling with a large number of threads. First, the used Max-Dicut algorithm in the final merging step is always performed sequentially. If we have many threads available, then the computation of the partition and the local solutions becomes faster, and thus the sequential merging step becomes more relevant for the total execution time. Second, our test system has two CPUs with 32 cores each, where each CPU is part of a separate NUMA node. When using close to or even more than 32 threads, the system will use cores from both CPUs. Some cores will therefore inevitably access memory outside their local NUMA node, which is slower than accessing local memory[4].

Note that for `great-britain` (the largest graph that we considered) we require at least 16 threads in order to use Goemans. With fewer threads, the framework will not finish within the given time limit.

3.2.3 Number of Subgraphs vs. Quality

Finally, we evaluate the influence of the number k of subgraphs on the quality of the computed dicut. This will also allow us to draw conclusions on how the framework should be best configured when running on different graphs. As we have seen already, the simple local search sometimes significantly improves the quality of the computed cut. For the following experiments, we therefore omit the local search. This allows us to see the true effect of the choice of k on the cut quality. Also, we always use 64 threads in order to get the best performance that is possible with our test system.

Figure 4 shows plots for both the quality and the achieved throughput for k between 64 and 8192. For $\mathsf{Derandomized}_{S2}$, Random_{S2}, and $\mathsf{Buchbinder}_{S2}$, the throughput decreases for increasing k. This is due to the fact that we use the superlinear Goemans algorithm for merging, and the graph used for merging has $2k$ nodes. On the other hand, the throughput increases with k when running $\mathsf{Goemans}_{S2}$. The reason for this is that a large value of k implies smaller subgraphs, for which the local solutions can then be computed faster. As seen for the graphs `flickr`, `luxembourg` and `web-sk-2005`, there is a range of k for which the throughput of $\mathsf{Goemans}_{S2}$ is in an equilibrium; increasing k within this range seems to equally accelerate the local solutions and decelerate the merging into a global solution. However, once k becomes too large, the throughput of $\mathsf{Goemans}_{S2}$ appears to decrease with k. We could only observe this for our smallest graph `Stanford3`.

Now let us focus on the quality of the cut. For $\mathsf{Goemans}_{S2}$, the quality generally decreases for larger values of k. This can be seen for all graphs except for `great-britain`, where $k = 8192$ is the only configuration that finished in time, and for `Stanford3`, where the cut quality first decreases, and then increases again once k exceeds 1024. This anomaly is likely due to the fact that `Stanford3` only has around 10000 nodes, and for large k the

[4] For more information on NUMA architectures see `https://uefi.org/specs/ACPI/6.4/17_NUMA_Architecture_Platforms/NUMA_Architecture_Platforms.html`

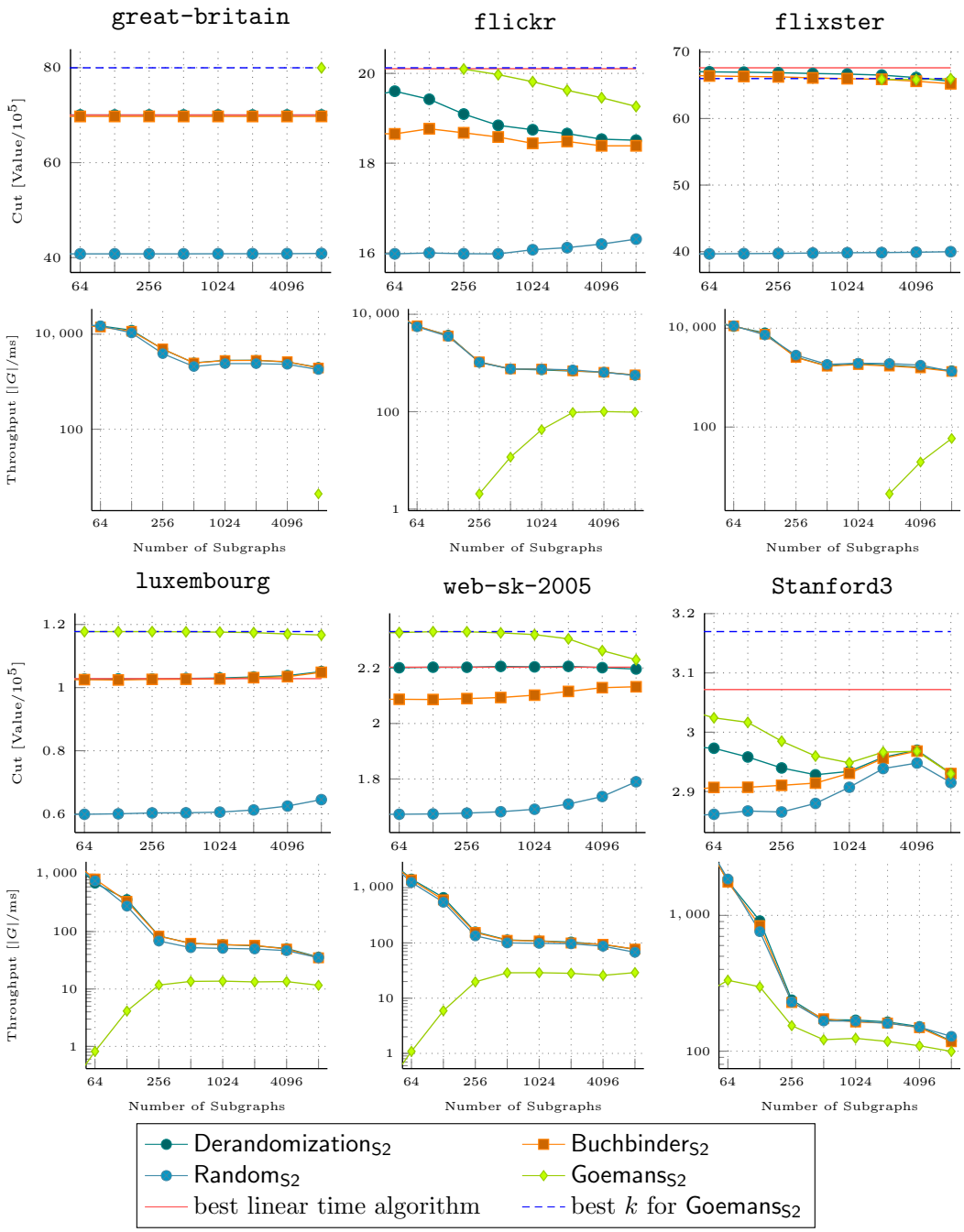


Figure 4 Scaling experiments for our framework where we visualize the effect of partitioning different graphs into a varying number of subgraphs on the computed cut quality and the running time of our framework. We partition our graph into between 64 to 8192 subgraphs while the number of used threads is fixed to 64. For this experiments we compare the algorithms Derandomized$_{S2}$, Buchbinder$_{S2}$, Random$_{S2}$ or Goemans$_{S2}$. In the plot above we see the computed cut and in the plot below the throughput (size of graph divided by running time) of our framework for each configuration. We additionally provide the cut quality of the best linear time algorithm (as continuous red line) that does not use our framework and the best computed cut for some $k \in \{2, 4, 8, ..., 8192\}$ by using Goemans$_{S2}$ in our framework as dashed blue line.

computation becomes more and more similar to a simple sequential execution of Goemans on the entire graph. The cuts computed by Derandomized$_{S2}$, Random$_{S2}$ and Buchbinder$_{S2}$ are generally worse than the one computed by Goemans$_{S2}$. The overall order of quality is (from worst to best): Random$_{S2}$, Buchbinder$_{S2}$, Derandomized$_{S2}$, Goemans$_{S2}$. The only exception is `flixster`, where the cut quality of most algorithms is almost the same, and only Random$_{S2}$ performs very poorly. With increasing k, the cut quality of Derandomized$_{S2}$, Random$_{S2}$ and Buchbinder$_{S2}$ on the smaller graphs (`luxembourg`, `web-sk-2005`, `Stanford3`) increases. This is because for large k, a significant fraction of the total edge weights is between the subgraphs. Thus, the simple linear time algorithms used for the local solutions loose relevance, while the better Goemans algorithm used for merging becomes more relevant. However, the quality does not reach the one of Goemans$_{S2}$.

We conclude that Goemans$_{S2}$ produces the best cuts, and that the best choice of k is the smallest k that allows the framework to finish within the given time limit. Increasing k further than that will only decrease the cut quality, and not even necessarily accelerate the computation.

4 Conclusion

We described a parallel framework for computing an approximate Max-Dicut that scales well for large graphs and produces high quality cuts for graphs with high coverage. It is also extendable, such that it is easy to add new algorithms.

The experiments showed that the best quality cuts are obtained by dividing the graphs into as few partitions as the Goemans algorithm finishes within the time limit on each of the partitions. While the sizes of resulting partitions can be used as a first indicator for choosing the right number of partitions, it remains an open problem to find an exact predictor for this. Also, our algorithms only sensibly scale for $\approx \sqrt{n}$ processors, as with more processors the sequential merging part becomes too expensive. Maybe a recursive (multilevel) approach can be used to fill this gap.

References

1 Y. Akhremtsev, P. Sanders, and C. Schulz. High-Quality Shared-Memory Graph Partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2710–2722, 2020. doi:10.1109/TPDS.2020.3001645.

2 Amotz Bar-Noy and Michael Lampis. Online maximum directed cut. *Journal of combinatorial optimization*, 24(1):52–64, 2012. doi:10.1007/s10878-010-9318-6.

3 Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004. doi:10.1002/spe.587.

4 N. Buchbinder, M. Feldman, J. Naor, and R. Schwartz. A Tight Linear Time (1/2)-Approximation for Unconstrained Submodular Maximization. In *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 649–658, 2012. doi:10.1137/130929205.

5 A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. *Recent Advances in Graph Partitioning*, pages 117–158. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-49487-6_4.

6 U. Feige and S. Jozeph. Oblivious Algorithms for the Maximum Directed Cut Problem. *Algorithmica*, 71(2):409–428, February 2015. doi:10.1007/s00453-013-9806-z.

7 M. X. Goemans and D. P. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *J. ACM*, 42(6):1115–1145, 1995. doi:10.1145/227683.227684.

8 Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep Multilevel Graph Partitioning. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 48:1–48:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2021.48.

9 N. Gusmeroli, T. Hrga, B. Luvzar, J. Povh, M. Siebenhofer, and A. Wiegele. BiqBin: a parallel branch-and-bound solver for binary quadratic problems with linear constraints. *arXiv: Optimization and Control*, 2020. doi:10.48550/arxiv.2009.06240.

10 E. Halperin and U. Zwick. Combinatorial Approximation Algorithms for the Maximum Directed Cut Problem. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 1–7, USA, 2001. Society for Industrial and Applied Mathematics.

11 T. Hrga and J. Povh. MADAM: A parallel exact solver for Max-Cut based on semidefinite programming and ADMM. *Computational Optimization and Applications*, 80:347–375, 2021. doi:10.1007/s10589-021-00310-6.

12 A. Jez. Recompression: A Simple and Powerful Technique for Word Equations. *J. ACM*, 63(4):1–51, 2016. doi:10.1145/2743014.

13 R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, 1972. doi:10.1007/978-1-4684-2001-2_9.

14 S. Khot, G. Kindler, E. Mossel, and R. O'Donnell. Optimal Inapproximability Results for MAX-CUT and Other 2-Variable CSPs? *SIAM J. Comput.*, 37(1):319–357, 2007. doi:10.1137/S0097539705447372.

15 N. Krislock, J. Malick, and F. Roupin. BiqCrunch: A Semidefinite Branch-and-Bound Method for Solving Binary Quadratic Problems. *ACM Trans. Math. Softw.*, 43(32):1–23, 2017. doi:10.1145/3005345.

16 M. Lampis, G. Kaouri, and V. Mitsou. On the Algorithmic Effectiveness of Digraph Decompositions and Complexity Measures. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Algorithms and Computation*, pages 220–231, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-92182-0_22.

17 M. Lewin, D. Livnat, and U. Zwick. Improved Rounding Techniques for the MAX 2-SAT and MAX DI-CUT Problems. In W. J. Cook and A. S. Schulz, editors, *Integer Programming and Combinatorial Optimization*, pages 67–82, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-47867-1_6.

18 Michael Luby and Noam Nisan. A parallel approximation algorithm for positive linear programming. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 448–457, 1993. doi:10.1145/167088.167211.

19 P. Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences*, 37(2):130–143, 1988. doi:10.1016/0022-0000(88)90003-7.

20 F. Rendl, G. Rinaldi, and A. Wiegele. Solving Max-Cut to optimality by intersecting semidefinite and polyhedral relaxations. *Mathematical Programming*, 121:307–335, 2010. doi:10.1007/s10107-008-0235-8.

21 R. A. Rossi and N. K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. URL: http://networkrepository.com.

22 J. Spencer. *Ten Lectures on the Probabilistic Method*, volume CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 2nd edition, 1994. doi:10.1137/1.9781611970074.

23 U. Zwick. Analyzing the MAX 2-SAT and MAX DI-CUT approximation algorithms of Feige and Goemans, 2000. manuscript.

A Small Graphs

Table 2 Our used small graphs.

Graph	n	m
08blocks.mtx	300	592
3elt.mtx	4,720	27,444
BA-1_10_60-L5.edges	805	46,410
BZR.edges	14,480	31,070
COIL-RAG.edges	11,758	23,588
CSphd.mtx	1,882	1,740
California.mtx	9,664	16,150
DD199.edges	842	1,902
DD21.edges	5,749	14,267
DD242.edges	1,285	3,303
DD244.edges	292	822
DD349.edges	898	2,087
DD497.edges	904	2,453
DD6.edges	4,153	10,320
DD68.edges	776	2,093
DD687.edges	726	2,600
D_11.mtx	461	2,952
ENZYMES118.edges	97	121
ENZYMES123.edges	91	127
ENZYMES295.edges	125	139
ENZYMES296.edges	127	141
ENZYMES297.edges	123	149
ENZYMES8.edges	89	133
EPA.mtx	4,772	8,965
EVA.mtx	8,497	6,726
Franz3.mtx	2,800	11,520
G11.mtx	800	3,200
G12.mtx	800	3,200
G13.mtx	800	3,200
G32.mtx	2,000	8,000
G33.mtx	2,000	8,000
G34.mtx	2,000	8,000
G48.mtx	3,000	12,000
G49.mtx	3,000	12,000
G57.mtx	5,000	20,000
G62.mtx	7,000	28,000
G65.mtx	8,000	32,000
G66.mtx	9,000	36,000
G67.mtx	10,000	40,000
Letter-high.edges	10,508	20,250
Letter-low.edges	10,523	14,092
Letter-med.edges	10,519	14,426

Graph	n	m
MSRC-21C.edges	8,419	40,380
OHSU.edges	6,480	31,546
PTC-FM.edges	4,926	10,110
PTC-FR.edges	5,111	10,532
PTC-MM.edges	4,696	9,624
PTC-MR.edges	4,916	10,108
Peking-1.edges	3,342	13,150
SW-10000-6-0d3-L2.edges	10,001	30,000
SW-10000-6-0d3-L5.edges	10,001	30,000
TerroristRel.edges	882	8,592
aves-sparrow-social.edges	53	516
aves-weaver-social.edges	446	1,426
aves-wildbird-network.edges	203	11,900
bio-CE-GN.edges	2,220	53,683
bio-CE-GT.edges	924	3,239
bio-CE-HT.edges	2,617	2,985
bio-CE-LC.edges	1,387	1,648
bio-CE-PG.edges	1,871	47,754
bio-DM-HT.edges	2,989	4,660
bio-DM-LC.edges	658	1,129
bio-HS-HT.edges	2,570	13,691
bio-HS-LC.edges	4,227	39,484
bio-SC-CC.edges	2,223	34,879
bio-SC-GT.edges	1,716	33,987
bio-SC-LC.edges	2,004	20,452
bio-SC-TS.edges	636	3,959
bio-celegans.mtx	453	4,050
bio-celegansneural.mtx	297	2,345
bio-diseasome.mtx	516	2,376
bio-grid-fission-yeast.edges	2,031	25,274
bio-grid-mouse.edges	1,455	3,272
bio-grid-plant.edges	1,745	6,196
bio-grid-worm.edges	3,518	13,062
bio-yeast.mtx	1,458	3,896
bn-mouse-kasthuri_graph_v4.edges	1,029	1,700
c-fat200-1.mtx	200	3,068
c-fat200-2.mtx	200	6,470
c-fat500-1.mtx	500	8,918
chesapeake.mtx	39	340
citeseer.edges	3,244	4,536
cora.edges	2,709	5,429
delaunay_n10.mtx	1,024	6,112
delaunay_n11.mtx	2,048	12,254
delaunay_n12.mtx	4,096	24,528

Table 3 Our used small graphs.

Graph	n	m
eco-florida.edges	129	2,106
eco-foodweb-baydry.edges	129	2,137
eco-foodweb-baywet.edges	129	2,106
eco-mangwet.edges	98	1,492
eco-stmarks.edges	55	356
email-dnc-corecipient.edges	2,030	12,085
email-dnc.edges	2,029	39,264
email-enron-only.mtx	143	1,246
email-univ.edges	1,134	5,451
fb-forum.edges	900	33,720
gene.edges	1,094	1,672
ia-contacts_hypertext2009.edges	114	20,818
ia-dnc-corecipient.edges	2,030	12,085
ia-hospital-ward-proximity-attr.edges	1,661	32,424
ia-hospital-ward-proximity.edges	1,661	32,424
ia-workplace-contacts.edges	876	9,827
inf-euroroad.edges	1,175	1,417
insecta-ant-trophallaxis-colony1.edges	42	308
insecta-ant-trophallaxis-colony2.edges	40	330
internet-industry-partnerships.edges	218	631
mammalia-primate-association.edges	26	1,340
mammalia-raccoon-proximity.edges	25	1,997
mammalia-voles-bhp-trapping.edges	1,687	5,324
mammalia-voles-kcs-trapping.edges	1,219	4,258
mammalia-voles-plj-trapping.edges	1,264	3,863
mammalia-voles-rob-trapping.edges	1,481	4,569
reptilia-tortoise-network-bsv.edges	137	554
reptilia-tortoise-network-cs.edges	74	258
reptilia-tortoise-network-fi.edges	788	1,713
reptilia-tortoise-network-hw.edges	17	22
reptilia-tortoise-network-lm.edges	46	134
reptilia-tortoise-network-mc.edges	16	45
reptilia-tortoise-network-pv.edges	36	104
reptilia-tortoise-network-sg.edges	25	29
reptilia-tortoise-network-sl.edges	12	18

A Fast Data Structure for Dynamic Graphs Based on Hash-Indexed Adjacency Blocks

Alexander van der Grinten
Humboldt-Universität zu Berlin, Germany

Maria Predari
Humboldt-Universität zu Berlin, Germany

Florian Willich
Humboldt-Universität zu Berlin, Germany

Abstract

Several dynamic graph data structures have been proposed in literature. Yet, these data structures either offer limited support for arbitrary graph algorithms or they are designed as part of specific frameworks (e.g., for GPUs or specialized hardware). Such frameworks are difficult to adopt to arbitrary graph computations and lead practitioners to fall back to less sophisticated solutions when dealing with dynamic graphs. In this work, we propose a new "dynamic hashed blocks" (DHB) data structure for sparse dynamic graphs and matrices on general-purpose CPU architectures. DHB combines an efficient block-based memory layout to store incident edges with an additional per-vertex hash index for high degree vertices. This hash index allows us to quickly insert edges without introducing duplicates, while the block-based memory layout retains advantageous cache locality properties of traditional adjacency arrays.

Experiments show that DHB outperforms competing dynamic graph structures for edge insertions, updates, deletions, and traversal operations. Compared to static CSR layouts, DHB exhibits only a small overhead in traversal performance. DHB's interface is similar to general-purpose abstract graph data types and can be easily used as a drop-in replacement for traditional adjacency arrays. To demonstrate that, we modify the well-known NetworKit framework to use DHB instead of its own dynamic graph representation. Experiments show that this modification only slightly penalizes the performance of graph algorithms while considerably boosting update rates.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases dynamic graph data structures, sparse matrix layout, dynamic algorithms, parallel algorithms, graph analysis

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.11

Supplementary Material *Software (Source Code)*: `https://github.com/hu-macsy/dhb`
archived at `swh:1:dir:9576f651115c810985803de5214f519bfc9600ef`

Funding *Alexander van der Grinten*: The author was supported by German Research Foundation (DFG) grant GR 5745/1-1 (DyANE).
Maria Predari: The author was supported by German Research Foundation (DFG) DFG grant ME 3619/4-1 (ALMACOM).

Acknowledgements The authors would like to thank Duy Le Thanh for his help in setting up some competitors.

1 Introduction

Large-scale graph data are ubiquitous in various areas of science and engineering [1, 16]. Yet, their efficient processing is still challenging. For one, the graphs in question are large and sparse. Typically, the number of neighbors [non-zero values] of a vertex [row/column] in a sparse graph [matrix] is bounded by a small constant and most possible edges [matrix entries]

do not exist [are zero].[1] Thus, sparse-friendly data structures are employed to avoid wasting processing and memory on empty entries. Dynamic graph applications introduce a second challenge as, in practice, most well-known graph processing frameworks still use static data structures [17, 26, 27]. Static graph representations are memory-efficient and support fast operations but lack flexibility in terms of dynamic updates. Examples of highly dynamic data are the Facebook and Twitter graphs, where users and connections are added/removed continuously [1]. Those mutations on the data structure (updates) are typically followed by graph queries to ensure consistency of analytics.

Recently, a number of dynamic graph frameworks were proposed, enabling graph processing and analysis for dynamically changing data [5, 8, 30, 31]. These frameworks often perform updates and queries concurrently, either via snapshots (simultaneous graph copies) or via batched updates. Existing dynamic graph frameworks differ in multiple aspects: their design, concurrency strategy, API support for graph analytics and applicability to generic architectures. In general, there are two performance goals for dynamic graph algorithms. The first goal aims at maximizing update rates, while maintaining low memory utilization. The second one aims at accelerating graph analytics running on top of their graph structure. Most dynamic graph frameworks consider the first goal but ignore the second. These frameworks focus mainly on graph updates and offer limited support for developing higher-level graph analytics [30]. Moreover, the ones that also consider the second goal are systematically slower than the ones that only focus on the first [19].

We propose a new data structure (DHB) for efficient processing of dynamically mutating large-scale sparse graphs. DHB is designed for general-purpose CPU architectures and combines an efficient block-based memory layout to store incident edges with an additional hash index for high degree vertices. The data structure is conceptually simple (and straightforward to implement); yet, we are not aware of any systematic experimental evaluation of this (or an equivalent) data structure. DHB utilizes on average the same memory as NetworKit, a static graph data framework using adjacency arrays. In a single-threaded environment, DHB outperforms all competitors regarding insertions, deletions and weight updates for different graph types and sizes, being on average from 1.9 to 93.8 × faster. Our data sets include static and temporal graphs with up to 1.8B edges. In a parallel environment, DHB's performance is similar to aspen which is reported to be one of the fastest dynamic graph frameworks [4, 7]. Moreover, DHB implements efficient lookups and other graph operations to accelerate common graph algorithms. Our experiments demonstrate that running BFS in a dynamic setting on top of DHB outperforms the corresponding BFS execution on top of aspen by a factor of 2.5. Finally, we demonstrate that the overhead of using DHB instead of a static graph structure is low. More precisely, we integrate DHB as a drop-in replacement into NetworKit and run BFS in a static setting (no updates). We observe that BFS on top of DHB is only 15% slower than BFS on top of NetworKit's adjacency arrays. This is a low overhead compared to the significant performance improvement under edge updates.

The paper is organized as follows: in Section 2, we present relevant background on traditional graph structures and in Section 3 we briefly review existing solutions for dynamic graphs. In Section 4, we present our newly proposed graph structure (DHB) while in Section 5 we evaluate it against both static and dynamic competitors. Finally, in Section 6 we give our concluding remarks.

[1] Due to the correspondence of matrices and graphs [11], in the remainder, we use these terms interchangeably.

2 Preliminaries

Static graph representation

We consider directed sparse graphs $G = (V, E)$. Undirected graphs are modelled by splitting each undirected edge $\{u, v\}$ into two directed ones. General graphs are commonly represented by their $n \times n$ adjacency matrix A, where $n = |V|$ (the number of non-zero entries of A correspond to the number of edges $m = |E|$). The data structures that we consider, store A into a sparse layout. Common sparse data structures are the adjacency array (or adjacency list), the coordinate list (COO) and the compressed sparse row (CSR) (or compressed sparse column). In the adjacency array representation, each vertex u has an associated array that maintains the IDs of all vertices in its neighborhood $N(u)$. Adjacency arrays use $\mathcal{O}(n + m)$ space and check connectivity of two vertices in $\mathcal{O}(\deg(u)) \subseteq \mathcal{O}(\deg_{\max})$ time, where $\deg(u)$ is the degree of u and $\deg_{\max}$ is the maximum degree in G. COO holds an array of (row, column, value) tuples and is similar to the adjacency array in the asymptotic time, space complexity, and general design. The main difference is that each edge is stored explicitly, with both its source and destination vertex. The above storage formats allow for a limited number of updates but have big overheads due to re-allocation of data and slow search operations. Finally, CSR uses three arrays to store a sparse graph: a node array, an edge array, and a values array. Each entry in the node array contains the starting index in the edge array where the edges from that node are stored in sorted order by destination. The edge array stores the destination vertices of each edge. CSR stores a graph in $\mathcal{O}(n + m)$ space. Inserting an edge into the CSR format takes linear time in the worst case. The entire edge array may need to be copied into a larger block of memory if there are too many elements in the structure. As a result, CSR is also not a suitable format for dynamic updates.

Updates

When talking about dynamic graphs, we consider edge updates, i.e., insertion and deletions of edges, as well as edge weight changes. Insertions and/or deletions of vertices are handled by standard techniques, i. e. by resizing the data structure to be able to hold enough vertex IDs and by storing an additional bit per vertex to determine if the vertex is deleted or not.

Dynamic challenges

For dynamic graph structures there is a trade off between efficient updates, optimal memory layout, and fast lookups. Traditional adjacency arrays allocate one array per vertex. *Block-based* data structures refine this strategy by storing incident edges in blocks. A block is an array whose size comes from a set of size classes (often powers of two). Blocks of the same size are stored in a superblock of fixed size (e.g., 2 MiB). On the other hand, *compressed sparse* layouts simply concatenate all adjacency arrays to a single memory allocation. This makes it easy to iterate over edges, but difficult to resize the data structure. Finally, other data structures do not store edges in arrays at all but opt for a different primitive, such as hash tables. These data structures usually exhibit considerable overhead when iterating over incident edges. Regarding edge lookups (e. g. to detect duplicates or to update edge weights), many implementations simply loop over the entire adjacency list. Other data structures *sort* the edges to support $\mathcal{O}(\log n)$ lookups; however, this comes with the cost of more expensive dynamic updates (i. e. $\mathcal{O}(\deg_{\max})$ to delete an arbitrary edge). Implementations based on hash tables can usually look up edges in expected $\mathcal{O}(1)$ time.

3 Related Work

Recently, a number of dynamic graph frameworks have been proposed in the literature. We focus on frameworks that are designed for CPU architectures. STINGER [8] is a dynamic graph structure for multi-core architectures that stores the adjacency information of each vertex using blocked linked lists of pre-selected, fixed size. GraphIn [24] allows for incremental graph processing by combining two static graph data structures: CSR for the original input and a COO to store new edge updates. The framework has similar limitations as COO; it is constrained to a limited number of updates (pre-defined by the users). PSCR [28] (later extended to PPCSR [29]) is a dynamic data structure based on the packed memory array (PMA) [3]. Sha et al. [25] also uses a variant of PMA. PMA is an array with all neighborhoods (i.e., essentially a CSR) augmented with an implicit binary tree structure that enables edge insertions and deletions in $\mathcal{O}(\log^2 n)$ time. Unfortunately, the above solutions can not be easily integrated into existing graph frameworks, due to their limited support for arbitrary graph operations. Moreover, ASPEN [7] uses a novel probabilistic tree called a C-tree to store the graph structure and is reported to be one of the faster frameworks targeting CPUs for insertions and deletions [4]. Recently, TERRACE [20] introduced a hierarchical graph structure for dynamic graphs that uses both arrays and trees to store adjacencies, depending on their size.

Although our focus is on solutions for general-purpose CPU architectures, we briefly describe HORNET [5], a data structure designed for GPU architectures. HORNET is relevant to our work as it uses a similar block-based mechanism. More precisely, HORNET groups adjacency information of several vertices together in blocks, whose sizes are a power of two. Additionally, it uses a vectorized bit tree, and B+trees for managing memory blocks. HORNET is shown to outperform competing dynamic graph structures implemented for GPU architectures [4, 5]. An excellent survey on dynamic graph data structures, frameworks and databases can be found in [4].

4 Dynamic Hashed Blocks (DHB)

Our new DHB data structure uses a block-based memory layout with an additional hash index for high degree vertices to accelarate lookups of neighbors. While DHB builds on simple algorithmic primitives, the resulting data structure is highly competitive with state-of-the-art graph data structures, as demonstrated by our experiments in Section 5. Our data structure is designed around the following properties:

Basic operations. DHB supports the usual operations expected from an abstract data type for (dynamic) graphs: changing the number of vertices, insertion, update, and deletion of edges, edge existence queries, as well as neighborhood traversals. In contrast to frameworks such as Ligra or ASPEN [7], our data structure allows direct access to the neighbors of a vertex (while these frameworks only allow access via an EDGEMAP function). Unlike many other dynamic graph frameworks, we do *not only* focus on batch updates.

Random access. Likewise, many algorithms expect the ability to access the i-th neighbor of vertex u, where $i \in [0, \deg(u))$. For example, this is frequently used to sample a random neighbor of a vertex. To support this operation in $\mathcal{O}(1)$, it is convenient to store the neighbors of a vertex in a contiguous array. This requirement motivates the use of a block-based storage format in DHB.

Arbitrary neighbor ordering. Some algorithms require neighborhoods to be ordered in specific ways to work correctly. For example, the SUITOR algorithm to approximate the weighted matching problem requires edges to be sorted according to non-increasing edge

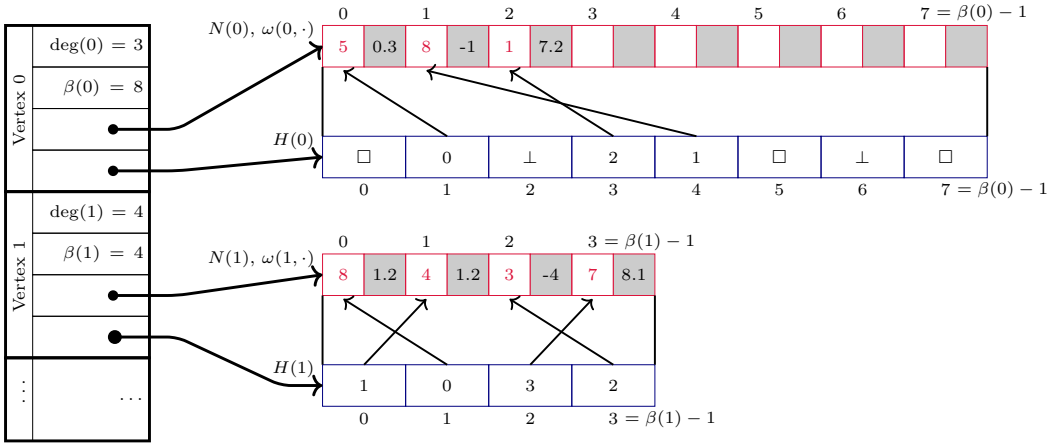


Figure 1 Layout of DHB. Each vertex has an associated adjacency block consisting of $N(\cdot)$ (red) and $H(\cdot)$ (blue). The adjacency block of vertex 0 has five empty slots left, while the adjacency block of vertex 1 is fully occupied. Gray boxes indicate edge weights that are stored interleaved with $N(\cdot)$.

weights [18]. To support these algorithms, the graph data structure should not impose a fixed order on the neighbors of each vertex, but preserve a user-specified order. This makes it possible to support an operation to re-order edges according to an arbitrary order. These requirements essentially rule out data structures that store neighbors directly in hash tables and data structures that rely on sorting to efficiently look up neighbors.

Support for concurrency. Many parallel algorithms (e. g. parallel graph generation algorithms) expect that neighbors of different vertices can be mutated in parallel. For this reason, we choose to use per-vertex hash indices (and not a global edge index) to accelerate lookups of neighbors.

4.1 Neighbors and Hash Index

The layout of DHB's main data structure is illustrated in Figure 1. DHB associates four data fields with each vertex u: (i) the current degree $\deg(u)$, (ii) a non-negative integer $\beta(u)$ that will store the number of neighbors currently reserved for vertex u, (iii) a pointer to an array $N(u)$ that can hold up to $\beta(u)$ neighbors of u, and (iv) a pointer to the hash index $H(u)$ of u. These fields are stored in an array indexed by the vertex ID u.[2] We define an *adjacency block* as the combination of the array that holds $N(u)$ and the array that holds $H(u)$. We call $\beta(u)$ the block size of the adjacency block of u. $\beta(u)$ will always be a power of two to accelerate the maintenance of the hash index. By extending the data structure appropriately, we always guarantee that $\deg(u) \leq \beta(u)$ (see below for details). In particular, the first $\deg(u)$ entries of $N(u)$ always hold the current neighbors of u, while the remaining $(\beta(u) - \deg(u))$ entries remain empty until new neighbors are inserted.

When edge weights and/or other per-edge data needs to be stored, we store this data interleaved with $N(u)$ (i. e. using an "array of structures", the gray boxes in Figure 1). This minimizes the number of pointers that our data structure has to store per vertex. For algorithms that only rarely access associated data, the data structure can be modified to

[2] Without loss of generality, we assume that vertices are identified by non-negative integer IDs in the range $[0, n)$. If this assumption is not satisfied (e. g. due to large gaps between IDs), an additional hash map can be used to map input vertex IDs to internal vertex IDs.

store pointers to additional per-vertex arrays that hold data associated with edges (yielding a "structure of arrays"). Another implementation strategy is assigning an edge ID to each edge, and storing associated data in a separate array that is indexed by edge ID (e. g. this is what NetworKit does).

The hash index $H(u)$

The hash index of a vertex u is used to quickly look up the position of neighbors of u in the adjacency array $N(u)$. We maintain $H(u)$ only for high degree vertices; for low-degree vertices, it is more efficient in practice to simply scan the entire adjacency list to find the index of a neighbor. We define high degree vertices as those whose neighborhood spans several cache lines (in our experiments, this threshold is set to 16 cache lines). If $H(u)$ is present, it consists of an array of $\beta(u)$ non-negative integers. $H(u)$ is used to implement a hash table based on open addressing. We use standard linear probing to resolve collisions. However, our hash table does not directly store any graph data; instead, it stores indices into $N(u)$. Initially, all slots of $H(u)$ are set to a special value $\square$ to indicate that the slots are empty. Another special value $\perp$ is used to indicate slots that became empty after deletions (i. e. $\perp$ represents a tombstone). If $H(u)[j] \notin \{\square, \perp\}$, then $H(u)[j]$ is always a valid index into $N(u)$, i. e. $H(u)[j] \in [0, \deg(u))$. We say that a slot j of the hash index corresponds to neighbor v of u if $N(u)[i] = v$, where $i = H(u)[j]$. The hash index will be constructed such that each non-empty slots of $H(u)$ correspond exactly to the neighbors of u. In Figure 1, this is represented by the arrows from $H(u)$ to $N(u)$.

Since the operations of DHB depend on the correct maintenance of the hash index, we briefly discuss how operations on the hash index itself behave. Looking up the possible neighbor v in the hash index of vertex u proceeds as follows: we start by evaluating a hash function $h : V \to \mathbb{N}$ to probe $H(u)$ at $j := (h(v) \mod \beta(u))$. Since $\beta(u)$ is a power of two, the modulo operation can be computed by a simple bitwise AND. If $H(u)[j] = \square$, then v is not a neighbor of u. In case we want to insert v, we can now set $N(u)[i] \leftarrow v$ and $H(u)[j] \leftarrow i$, where i is the smallest previously unused index of $N(u)$. On the other hand, if $H(u)[j] = \perp$, we increment $(j \mod \beta(u))$ (i. e. we probe linearly). Otherwise, $H(u)[j] \notin \{\square, \perp\}$. Let $i = H(u)[j]$. We can assume that i is a valid index into $N(u)$. If $N(u)[i] = v$, then v is a neighbor of u and we found its index i into $N(u)$. Otherwise, we continue probing linearly by incrementing $(j \mod \beta(u))$ and repeating the procedure.

We note that when updating $H(u)$ during the insertion of a new neighbor v, we can overwrite the first slot j with $H(u)[j] = \perp$, if we encounter such a slot; however, to correctly detect duplicates, we first have to finish through the entire probe sequence (i. e. until we either find v or $H(u)[j] = \square$). Furthermore, we remark that we can periodically purge all tombstones by rehashing a block (e.g., when there are more tombstones than entries present); this operation amortizes over many deletion operations and does not affect the overall running time complexity.

Reallocation

If $\deg(u) = \beta(u)$, i. e. the adjacency block of vertex u does not have any unused indices left, we need to allocate a new adjacency block for u before we can insert new neighbors. Note that allocating a new adjacency block does not necessarily trigger an OS-level allocation (e. g. `malloc`) if our custom memory allocation scheme is used (which is described in detail in Section A of our appendix). Since the performance of our hash index depends on the fill factor of $H(u)$, it is not advisable to wait until $\deg(u) = \beta(u)$, i. e. until the fill factor

Table 1 Asymptotic complexity (amortized and in expectation) of various common data structures that implement the graph abstract data type. DHB is at least as fast as the best competing algorithm for all operations. d: degree of modified source vertex, β: size of adjacency storage ($\beta \geq d$), $n = |V|$. For simplicity, block sizes (which only yield constant speedups) are omitted from this table. We also remark that low degree vertices (e. g. the vertices which are not hashed in DHB) do not affect the overall complexities.

	Insert	Delete	Change weight	Iterate over $N(\cdot)$	Query edge	Arbitrary order
DHB	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(d)$	$\mathcal{O}(1)$	yes
Adj. arrays (e. g. STINGER)	$\mathcal{O}(d)$	$\mathcal{O}(d)$	$\mathcal{O}(d)$	$\mathcal{O}(d)$	$\mathcal{O}(d)$	yes
Adj. arrays (sorted)	$\mathcal{O}(d)$	$\mathcal{O}(d)$	$\mathcal{O}(\log d)$	$\mathcal{O}(d)$	$\mathcal{O}(\log d)$	no
Hashing only	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\beta)$	$\mathcal{O}(1)$	no
ASPEN	$\mathcal{O}(\log n + \log d)$	$\mathcal{O}(\log n + \log d)$	$\mathcal{O}(\log n + \log d)$	$\mathcal{O}(\log n + d)$	$\mathcal{O}(\log n + \log d)$	no
TERRACE	$\mathcal{O}(\log d)$	$\mathcal{O}(\log d)$	$\mathcal{O}(\log d)$	$\mathcal{O}(d)$	$\mathcal{O}(\log d)$	no

reaches 100 %. Instead, we already reallocate the adjacency block once $\deg(u) \geq C \cdot \beta(u)$ for some constant $C < 1$ (e. g. $C = \frac{1}{2}$). When reallocating the adjacency block, we allocate a new adjacency block of block size $2 \cdot \beta(u)$, thereby increasing $\beta(u)$ to the next power of two.[3] Afterwards, we copy $N(u)$ into the new block, rebuild the hash index, update the pointers to $N(u)$ and $H(u)$ and deallocate the old adjacency block. Rebuilding the hash index is done by resetting all entries of $H(u)$ to $\square$, followed by a re-insertion of all neighbors of $N(u)$ into the hash index.

To perform actual allocations and/or deallocations, DHB can either use the system allocator (i. e. `malloc`), or a custom memory allocation scheme that is optimized for the block sizes that DHB uses. Our custom memory management works similarly to allocators in other block-based graph data structure (e. g. HORNET [5]); due to space constraints, we describe it in Section A of our appendix. In our experiments, we always use our custom memory allocator for DHB.

4.2 Operations

We briefly review the operations that DHB supports and their computational efficiency. The asymptotic running times given below are amortized over many updates (to account for reallocations) and in expectation (because of the hash index). We summarize these complexities in Table 1. In all cases, our running times are equally fast, or faster, than our competitors. Compared to traditional adjacency arrays, our hash index does not add asymptotic complexity for any operation.

Insertion of a new edge (u, v) at the end of the adjacency block first uses the hash index to check whether v is already a neighbor of u. If that is not the case, v is inserted at index $\deg(u)$ of $A(u)$ and $\deg(u)$ is incremented. This operation runs in $\mathcal{O}(1)$ time. Insertion in an arbitrary position needs to move trailing entries of $N(u)$ to free space for the new edge. Afterwards, the hash table needs to be updated for all entries that were moved. Overall, the operations runs in $\mathcal{O}(\deg(u))$.

Weight changes (or changes of other associated data) of an edge (u, v) can be performed in $\mathcal{O}(1)$ time by using the hash index to find the index of v in $N(u)$, followed by an update of the edge weight (which is stored interleaved with $N(u)$).

[3] Due to this reallocation strategy, the size of $N(u)$ can only ever reach $C \cdot \beta(u)$. Hence, it is actually enough to only allocate $C \cdot \beta(u)$ slots (and not $\beta(u)$ slots) for $N(u)$.

Deletion of an edge (u, v) first looks up the index of v in $N(u)$ by using the hash index. If the order does not need to be preserved, v can be swapped to the end of $N(u)$ and deleted in $\mathcal{O}(1)$ time. Otherwise, trailing entries of $N(u)$ need to be moved, incurring $\mathcal{O}(\deg(u))$ time.

Iteration over all neighbors of u simply iterates over the first $\deg(u)$ indices of $N(u)$, without involving $H(u)$ at all.

Edge queries check whether an edge (u, v) exists in $\mathcal{O}(1)$ time by looking up v in the hash index of u.

Reordering the neighbors of u (e. g. when sorting edges) is done by reordering $N(u)$ first. Afterwards, $H(u)$ is rebuilt from scratch in $\mathcal{O}(\deg(u))$ time.

Parallel updates

While not the focus of our data structure, we can support parallel batch updates of edges due to the fact that DHB allows adjacency blocks of distinct vertices to be manipulated concurrently. We achieve this by distributing the batch to all available threads in such a way that no threads t and t' receive edges (u, v) and (u, v') that share the same source vertex u. We achieve this by using a hash function to map source vertices u to threads. Integer sorting is used to sort the source vertices according to their hash value; afterwards, each thread applies all updates that concern the source vertices that are mapped to itself.

5 Evaluation

We perform experiments to evaluate the behavior of our data structure on several static and temporal graphs coming from SNAP [15], NR [22], and the KONECT [12] interactive data repository (see Tables 2 and 3 in Section B of the appendix). The largest graph has around 1.8B edges. Temporal graphs represent real dynamic applications and typically consist of a sequence of edges along with their timestamps. That sequence expresses some predefined pattern of edge additions related to the underlying application. We compare DHB to both static and dynamic graph frameworks. We choose NetworKit as the representative for static graph frameworks. NetworKit uses adjacency arrays to store the input graph, and the CSR representation for matrix based operations. For the comparison with dynamic graph frameworks, we choose STINGER, aspen and Terrace. STINGER is the representative framework for block-based adjacencies, aspen represents tree-based graph structures and Terrace uses different data structures depending on the neighborhood degree. We do not compare against a hashing-only implementation, as hashing alone is not competitive with other approaches when downstream algorithm performance (e. g. traversal of the data structure) is considered (and thus, state-of-the art dynamic graph data structures do not rely only on hashing).

We group the experiments into two categories. In Section 5.1, we evaluate all competitors in terms of common dynamic operations, i. e. insertions, deletions and edge weight updates. Then, in Section 5.2, we evaluate the performance of DHB for common graph applications, under dynamic and static settings. More experiments can be found in the appendix regarding: memory consumption and batch size evaluation (See Section C of the appendix) and scalability experiments for DHB (See Section D of our appendix). Experiments were conducted on a shared-memory parallel machine equipped with an 2x 18-Core Intel Xeon 6154 CPU

(2 sockets, 18 cores each), and a total of 1,5 TB RAM. [4] To ensure reproducibility, all experiments were managed by SimexPal [2]. Our code and the experimental pipeline is publicly available at `https://github.com/hu-macsy/dhb`.

Configuration of competitors

STINGER reserves half of the available physical memory and also requires the user to set the number of adjacency blocks allocated for the data structure. We set the number of expected neighbors per vector to STINGER's default (i. e. `STINGER_DEFAULT_NEB_FACTOR` $\cdot |V|$) and let STINGER use enough memory to fit these blocks into memory (i. e. 768 GiB). We do not use STINGER's client-server architecture or vertex mappings features (and also do not remap vertex IDs for all other data structures). All edge updates are performed by using STINGER's `stinger_update_directed_edge()` function.

ASPEN implements a single-writer, multi-reader interface following a lock-free approach i. e. allowing any number of concurrent readers and a single writer on a graph snapshot. To enable parallelism, ASPEN uses its own scheduler, similar to Cilk [14]. All edge insertions [deletions] are performed via ASPEN 's `insert_edges_batch()` [`delete_edges_batch()`].

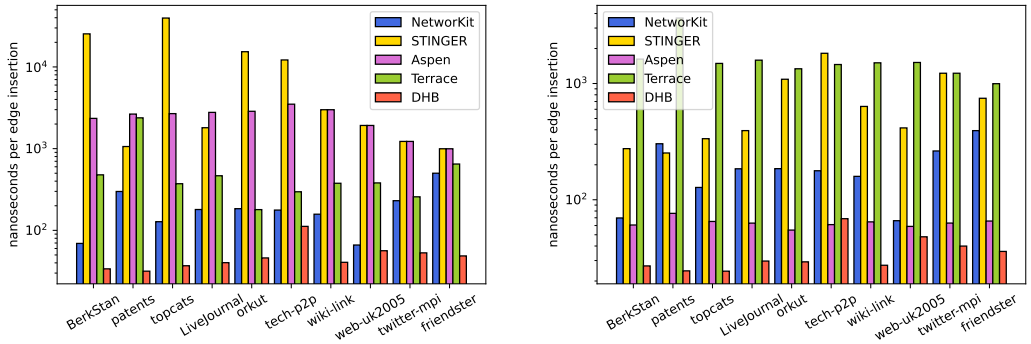
The recommended instructions for building TERRACE require a non-standard branch of the LLVM compiler (TAPIR) and use Cilk Plus [6] for multi-threading. These options introduce additional optimizations and make TERRACE difficult to compare to other, arbitrary graph libraries. More precisely, the TAPIR branch improves upon mainline LLVM by optimizing across parallel regions [23], which contributes to an increased performance for TERRACE [20]. To ensure similar build settings for all involved frameworks, we compile TERRACE with GCC and use OpenMP for multi-threading (since Cilk Plus support was recently deprecated and removed from GCC). Finally, we set TERRACE's `MEDIUM_DEGREE` to 2^{10} to avoid memory issues (after discussion with the authors of TERRACE).

5.1 Insertion, Update and Deletion Performance

For edge insertions, we perform experiments with initially empty graphs and insert all edges after verifying existence in the graph. We use two different modes for the insertion, a single-edge insertion and a bulk insertion in one batch – depicted in Figures 2a and 2b, respectively. The above experiments are performed in a single-threaded environment and include all competitions. Note that since TERRACE pre-allocates data structures in its constructor (i. e. enough memory for up to 15 edges per vertex), we also include the construction time in our measurements. This does not hinder the fairness of the experiment since all other frameworks have insignificant construction times (less than a millisecond). It is clear that DHB outperforms all competitors for both insertion modes in the single-threaded case. More precisely, DHB is on average 3.6 × faster than NETWORKIT and 9.4 × faster than TERRACE (the best competitors) for single insertions and 1.9 × faster than ASPEN (the best competitor) for the bulk insertion. Compared to STINGER, DHB is on average 17.3 [93.8] × faster for single [bulk] insertions.

Moreover, we perform experiments in a multi-threaded environment with 18 threads, depicted in Figure 3. NETWORKIT is not included in the multi-threaded experiment as it does not support parallelism. Moreover, it seems that building TERRACE with OpenMP highly penalizes its performance (causing it to run slower than in a single-threaded run).

[4] `https://www2.hu-berlin.de/macsy/technical-overview.html`



(a) Single edge insertions (one-by-one). **(b)** Bulk edge insertion (all in one batch).

Figure 2 Edge insertion experiments for static graphs on single-threaded environment.

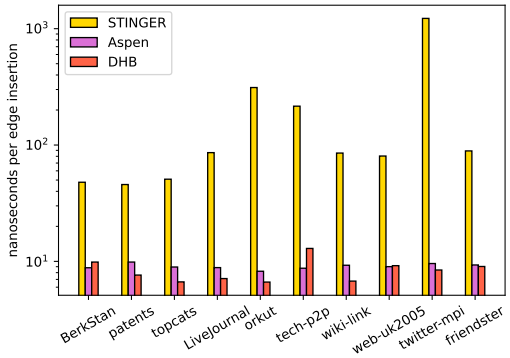
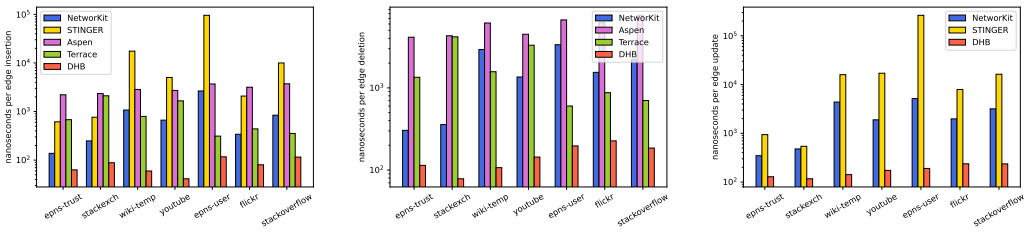


Figure 3 Bulk edge insertion for static graphs on multi-threaded environment (18 threads).

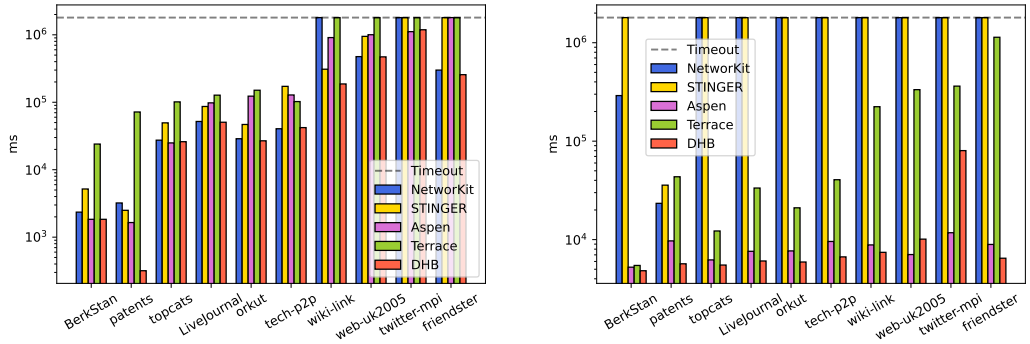
The results we obtain do not reflect the expected performance of TERRACE, as reported in the original paper [20] (which uses the custom TAPIR branch of LLVM). Therefore, to avoid ill-founded conclusions regarding TERRACE's performance in the multi-threaded experiment, we choose to exclude such results. In Figure 3, we observe that DHB is slightly faster than ASPEN (10% on average) regarding multi-threaded insertions and both competitors are on average around 14.2 × faster than STINGER.

Moreover, we perform experiments for the temporal graphs of Table 3 including edge insertions, deletions and edge weight updates (changing the weight of an edge – not inserting a new one). For deletions and weight updates, we pick the affected edges uniformly at random from the set of all edges after insertion. In this way, we do not risk altering the degree distribution of the updated graph. For edge weight updates ASPEN and TERRACE are excluded from the experiments. The former because it does not support weighted graphs while the latter because it does not offer an explicit method for edge weight updates (other than deleting and re-inserting the edge). The results are reported in Figure 4. For insertions [deletions] to temporal graphs, DHB is on average 7.4 [8.9] × faster than NETWORKIT, as seen in Figures 4a and 4b. Unfortunately, STINGER times out for all edge deletion experiments at 1 800 secs, so we exclude it from Figure 4b. The general trend is similar for edge weight updates too: DHB is 10.2 × faster than NETWORKIT and 53.1 × faster than STINGER, as seen in Figure 4c.



(a) Random edge insertions. **(b)** Random edge deletions. **(c)** Random edge weight update.

Figure 4 Single-threaded edge insertions, deletions and weight updates for temporal graphs.



(a) BFS. **(b)** SpGEMM.

Figure 5 Performance of common graph applications on top of dynamic graph structures (dynamic setting).

5.2 Applications

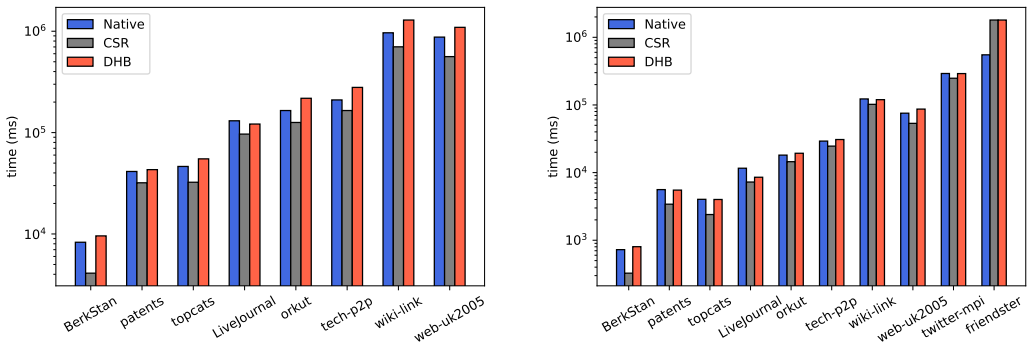
We evaluate how DHB performs in various application scenarios. For this purpose, we pick the popular BFS and SpGEMM benchmarks. We also demonstrate that DHB can easily be integrated into existing graph applications.

Breadth-first search (BFS)

We compare the performance of alternating edge insertions and BFS queries of DHB and its competitors. In this experiment, we initialize the graph to all but 10M edges (without measuring the initialization time). Afterwards, we insert 100k edges into the graph and run a BFS from a random source vertex. This process is iterated 100 times, i. e. until all edges of the graph are inserted. Identical source vertices are picked for all competitors. Figure 5a depicts the results (reporting end-to-end running time). DHB is on average 1.7 × faster than NetworKit and 2.5 × faster than ASPEN.

Sparse matrix-matrix multiplication (SpGEMM)

In a second experiment, we aggregate the results of a SpGEMM computation into an existing matrix. The need to aggregate the result of a SpGEMM computation arises in various graph mining applications and/or in distributed matrix multiplication algorithms [9]. In particular, we compute the first 100M non-zeros of $\mathbf{A}^2$ where $\mathbf{A}$ is the adjacency matrix of each graph and measure the running time of this computation. We use the standard sparse row-by-row algorithm by Gustavson [10]. Note that ASPEN does not support weights, but only aggregates



(a) BFS. **(b)** SpMV.

Figure 6 Comparison of DHB and NetworKit's native graph structures on static graph algorithms (no updates).

the structure and not the actual values in this experiment. Furthermore, since ASPEN only supports batch updates, we always buffer one row of the output before inserting it into ASPEN's data structure; this strategy improves ASPEN's performance considerably. All other data structures support weights and perform individual edge updates without buffering. Figure 5b shows that DHB and ASPEN report the best performance results for SpGEMM and are on average 8.4 × faster than TERRACE. Unfortunately, most of the runs time out for STINGER and NetworKit at 1 800 secs.

Integrating DHB into custom graph structures

Our final experiment demonstrates the viability of DHB as a faster drop-in replacement for custom graph data structures. We integrate DHB into the well-known graph framework NetworKit. NetworKit includes two graph data structures: a native adjacency array, and a CSR representation. We evaluate the overhead of DHB's integration compared to both NetworKit's representations. For the evaluation we pick the BFS and SpMV (= sparse matrix times dense vector multiplication) benchmarks, since they are both used as primitives in more sophisticated graph algorithms. Adjusting NetworKit's BFS and SpMV to work on top of DHB is easy to do since our data structure has the same interface as custom graph data structures. Experiments demonstrate that the DHB-enhanced version of NetworKit slightly penalizes the performance of BFS and SpMV (being on average 15% [25%] slower than NetworKit's native adjacency array [CSR] representation). The results suggest only a small overhead for graph algorithm performance compared to a significant performance improvement for edge updates. Specifically, the overhead is in line with what other authors have observed when moving from static to dynamic graphs [7].

6 Conclusions

In this work, we present DHB, a new data structure for storing and processing dynamic, large-scale, sparse graphs and matrices. DHB is designed for general-purpose CPU architectures and combines an efficient block-based memory layout to store incident edges with an additional hash index for high degree vertices. Our dynamic data structure supports edge insertions, deletions and edge weight updates. We demonstrate experimentally that DHB outperforms competing dynamic graph data structures in terms of update rates and graph applications for

both static and temporal (real) graph data. To show the viability of our data structure, we integrate DHB as a drop-in replacement for NetworKit's native dynamic graph structure (adjacency arrays). Experiments demonstrate that using DHB instead of NetworKit's native graph layout incur a small overhead for graph algorithms, while significantly increasing the update rates for edge insertions, deletions and, weight updates.

References

1. Khaled Ammar. Techniques and systems for large dynamic graphs. In Eduard C. Dragut and Heng Tao Shen, editors, *Proceedings of the SIGMOD 2016 PhD Symposium, San Francisco, California, USA, June 26, 2016*, pages 7–11. ACM, 2016.
2. Eugenio Angriman, Alexander van der Grinten, Moritz von Looz, Henning Meyerhenke, Martin Nöllenburg, Maria Predari, and Charilaos Tzovas. Guidelines for experimental algorithmics: A case study in network analysis. *Algorithms*, 12(7):127, 2019.
3. Michael A. Bender and Haodong Hu. An adaptive packed-memory array. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 20–29, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1142351.1142355.
4. Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. Practice of streaming processing of dynamic graphs: concepts, models, and systems, 2021. URL: https://open.bu.edu/handle/2144/42895.
5. Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *The 22nd Annual IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*, pages 1–7, Los Alamitos, CA, 2018. IEEE Computer Society. doi:10.1109/HPEC.2018.8547541.
6. Intel Corporation. Intel cilk plus language specification, 2010. URL: http://software.intel.com/sites/products/cilkplus/cilk_plus_language_specification.pdf.
7. Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 918–934, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314598.
8. David Ediger, Robert McColl, Jason E. Riedy, and A. David Bader. Stinger: High performance data structure for streaming graphs. *HPEC*, pages 1–5, 2012.
9. Jianhua Gao, Weixing Ji, Zhaonian Tan, and Yueyan Zhao. A systematic survey of general sparse matrix-matrix multiplication. *CoRR*, abs/2002.11273, 2020. arXiv:2002.11273.
10. Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, September 1978. doi:10.1145/355791.355796.
11. Jeremy Kepner, Peter Aaltonen, A. David Bader, Aydin Buluç, Franz Franchetti, R. John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, E. José Moreira, D. John Owens, Carl Yang, Marcin Zalewski, and G. Timothy Mattson. Mathematical foundations of the graphblas. *HPEC*, pages 1–9, 2016.
12. Jérôme Kunegis. Konect: The koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, pages 1343–1350, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2487788.2488173.
13. Daan Leijen, Benjamin Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. In *APLAS*, volume 11893 of *Lecture Notes in Computer Science*, pages 244–265. Springer, 2019.
14. Charles E. Leiserson. *Cilk*, pages 273–288. Springer US, Boston, MA, 2011. doi:10.1007/978-0-387-09766-4_289.

15 J. Leskovec. Stanford Network Analysis Package (SNAP). URL: http://snap.stanford.edu/index.html.

16 Chun Liu, Shuhang Zhang, Hangbin Wu, and Qiang Fu. A dynamic spatiotemporal analysis model for traffic incident influence prediction on urban road networks. *ISPRS International Journal of Geo-Information*, 6(11), 2017. URL: https://www.mdpi.com/2220-9964/6/11/362.

17 Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, pages 340–349, Arlington, Virginia, USA, 2010. AUAI Press.

18 Fredrik Manne and Mahantesh Halappanavar. New effective multithreaded matching algorithms. In *IPDPS*, pages 519–528. IEEE Computer Society, 2014.

19 Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

20 Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, pages 1372–1385, New York, NY, USA, 2021. Association for Computing Machinery.

21 J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *Comput. J.*, 20(3):242–244, 1977.

22 Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL: http://networkrepository.com.

23 Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 249–265, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3018743.3018758.

24 Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. Graphin: An online high performance incremental graph processing framework. In *Proceedings of the 22nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*, pages 319–333, 2016.

25 Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on gpus. *Proc. VLDB Endow.*, 11(1):107–120, September 2017.

26 Julian Shun and E. Guy Blelloch. Ligra: a lightweight graph processing framework for shared memory. *PPOPP*, pages 135–146, 2013.

27 Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016. doi:10.1017/nws.2016.20.

28 Brian Wheatman and Helen Xu. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*, pages 1–7, September 2018. doi:10.1109/HPEC.2018.8547566.

29 Brian Wheatman and Helen Xu. A parallel packed memory array to store dynamic graphs. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 31–45. SIAM, 2021.

30 Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. Faimgraph: High performance management of fully-dynamic graphs under tight memory constraints on the gpu. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

31 Martin Winter, Rhaleb Zayer, and Markus Steinberger. Autonomous, independent management of dynamic graphs on gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, 2017. doi:10.1109/HPEC.2017.8091058.

A Memory Management

Since expanding the storage associated with a vertex (i. e. increasing $\beta(u)$) happens frequently when using the DHB data structure, care must be taken to avoid costly OS-level memory allocations whenever possible. Like other block-based dynamic graph data structure, we implement a custom memory manager to allocate adjacency blocks for this purpose. While sophisticated `malloc()` implementations exist, our allocator is able to improve upon invoking `malloc()` directly because of two reasons: first, due to the design of our data structure (and the hash index in particular), we only need to manage blocks of size 2^k for some k. In contrast, general purpose `malloc()` implementations usually maintain more fine-grained size classes to avoid wasting memory on arbitrary workloads. Secondly, our block deallocation procedure does not need to recover a pointer to the (more coarse-grained) OS-level memory allocation from a pointer to the adjacency block; instead, we can store additional meta data in our data structure.

The auxiliary data structures of our memory allocator are depicted in Figure 7. As all blocked-based dynamic graph data structures, our allocation scheme groups multiple adjacency blocks into a single contiguous superblock. Hence, we only need to perform an OS-level memory allocation per superblock. In our design, all blocks of the same superblock have the same block size (i. e. the same $\beta(u)$). To protect the memory allocator against concurrent access, we use a mutex per size class (i. e. per power of two).

Block allocation

We usually use a first fit allocation strategy to allocate blocks. This allocation strategy is known to result in low fragmentation [21]. Since it causes long-lived allocations to accumulate in the first few superblocks, we expect it to reduce the number of partially occupied superblocks that we have to maintain. More precisely, we allocate from the oldest superblock of a given block size that currently has unused blocks available. For this purpose, we store all superblocks of the same block size that are not fully occupied in a balanced binary search tree ordered by their age, i. e. the order in which they were created.

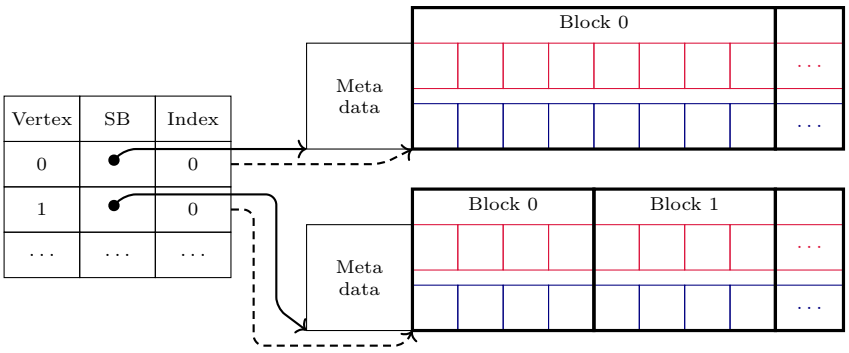


Figure 7 Layout of superblocks and block handles. Each vertex stores a pointer to its superblock and the index of its adjacency block within this superblock. The adjacency block contains both $N(\cdot)$ (red) and $H(\cdot)$ (blue); colors match Figure 1.

Table 2 Static graphs. The columns of the table correspond (in order) to network name, abbreviation, minimum degree, maximum degree, mean degree, maximum vertex ID, and edge count.

Network	Abbrev.	$\deg_{\min}$	$\deg_{\max}$	$\deg_{\text{mean}}$	$\lvert V\rvert$	$\lvert E\rvert$
web-BerkStan	BerkStan	0	249	11.1	685K	7.6M
cit-Patents	patents	0	770	2.8	6.01M	16.5M
wiki-topcats	topcats	0	3.91K	16.0	1.79M	28.5M
soc-LiveJournal1	LiveJournal	0	20.3K	14.3	4.85M	69M
com-orkut	orkut	0	33K	38.2	3.07M	117M
tech-p2p	tech-p2p	1	10.7K	25.6	5.79M	148M
web-wikipedia_link_en13	wiki-link	0	37K	20.2	27.2M	601M
web-uk-2005	web-uk2005	1	1.78M	23.7	39.5M	936M
soc-twitter-mpi-sws	twitter-mpi	0	3M	35.3	41.7M	1.47B
com-friendster	friendster	0	3.62K	14.5	125M	1.81B

Table 3 Temporal graphs from real dynamic applications. The columns of the table correspond (in order) to network name, abbreviation, minimum degree, maximum degree, mean degree, maximum vertex ID, edge count during initial read, edge cout after update accordingly.

Network	Abbrev.	$\deg_{\min}$	$\deg_{\max}$	$\deg_{\text{mean}}$	$\lvert V\rvert$	E_{in}	E_{out}
soc-epinions-trust-dir	epns-trust	0	2.07K	6.4	132K	841K	841K
ia-stackexch-user-marks-post-und	stackexch	0	4.92K	2.4	545K	1.3M	1.3M
wiki-talk-temporal	wiki-temp	0	142K	3.0	1.14M	7.83M	3.31M
soc-youtube-growth	youtube	0	83.3K	3.0	3.22M	12.2M	9.38M
rec-epinions-user-ratings	epns-user	0	162K	18.1	756K	13.7M	13.7M
soc-flickr-growth	flickr	0	26.4K	14.4	2.3M	33.1M	33.1M
sx-stackoverflow	stackoverflow	0	42.2K	10.9	2.58M	47.9M	28.2M

The are two exceptions to this first fit rule: within a superblock, we simply allocate an arbitrary block (by storing a per-superblock stack of free blocks). This does not affect fragmentation since we can only release entire superblocks to the OS at a time. Secondly, to avoid frequent modifications of the balanced binary search tree, we do not immediate re-insert a fully occupied superblock into the tree once one of its blocks is deallocated. Instead, we employ a strategy similar to the one used by mimalloc [13]. In particular, we wait until $\frac{\beta}{2}$ of a superblock's blocks are deallocated before allocating from the superblock again. This reduces the overhead of the memory allocator without affecting its asymptotic properties.

To be able to free adjacency blocks, we let each vertex store a pointer to the superblock of its current adjacency block, and the index of this adjacency block within the superblock (depicts as arrows in Figure 7. When freeing the adjacency block, we simply push its index back to the stack of free blocks that is stored within the superblock. Since block reallocations happen only $\mathcal{O}(\log k)$-times for k edge updates, our implementation does not store these information within our main per-vertex array (i. e. the array depicted in Figure 1). Instead, we use a differnet array to improve memory locality.

B Instances

All graph used in the experiments can be found in Tables 2 (static) and 3 (temporal).

C Memory and Batch Size Experiments

We perform additional experiments regarding memory consumption and scaling of the batch size.

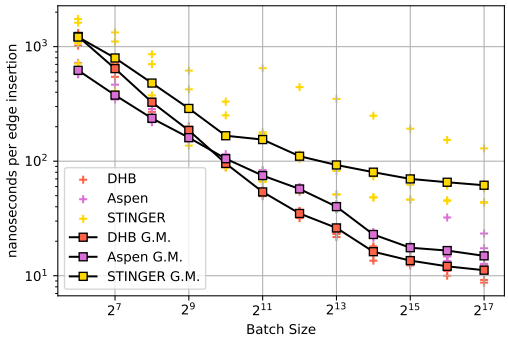


Figure 8 Edge insertion rate over the largest static graphs for an increasing batch size. We report individual and aggregated results using geometric mean over the graphs.

Batch size experiment

We use the five largest static graphs of Table 2 and scale the batch size for batched edge insertions in a parallel environment with 36 threads. Batch sizes range from 2 to 2^{17} updates per batch. The edges to be inserted are randomly generated and their existence is verified prior to insertion. In Figure 8 we report times per edge insertion for DHB, ASPEN, and STINGER (in logarithmic scale). DHB performs better than ASPEN for larger batch sizes while both exhibit a linear scaling. The best performance for DHB corresponds to a batch size of 2^{16} edges.

Memory consumption

Figure 9 presents memory utilization results for all involved data structures. We measure the peak memory consumption of each competitor for all temporal instances of Table 3. More precisely, we consider the peak resident set size after constructing and reading in the temporal graphs. DHB allocates similar amounts of memory as NETWORKIT and is on average 30% more memory-efficient that STINGER. STINGER's memory allocation is less dynamic, since it allocates multiple blocks of fixed size per neighborhood. It also does not seem to be optimized for memory efficiency. Moreover, DHB uses on average 20% less memory than ASPEN, although in some cases (see `flickr`) the trend may be opposite. ASPEN's overhead is due to the tree-based data structure which requires more space. TERRACE appears to have, on average, similar memory utilization as ASPEN. In particular, both TERRACE and ASPEN show significant memory savings over the competitors on the `epns-user` and `flickr` instances which have the highest average degrees.

D Scalability Experiments

We perform experiments to test the scalability of DHB w. r. t. a growing graph size. We also evaluate the parallel scalability of DHB in a multi-threaded parallel environment. For the first experiment, we perform $15 \times n$ random edge insertions for an increasing number of vertices, i. e. $n = 2^{20}, \ldots, 2^{26}$. In Figure 10a we observe that DHB exhibits a linear scaling behavior w. r. t. the graph size. For the parallel scalability, we perform edge insertion experiments for the five larger temporal graphs of Table 3. In Figure 10b we report aggregated speed ups of DHB w. r. t. a sequential run, for a thread count of up to 36 threads. We also include

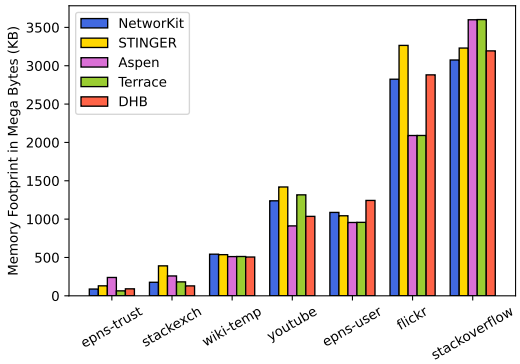
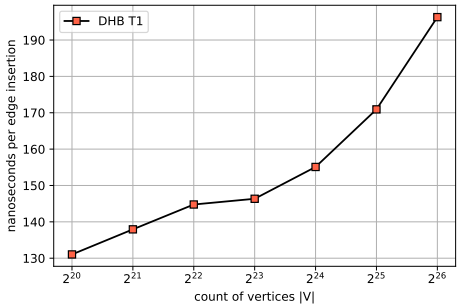
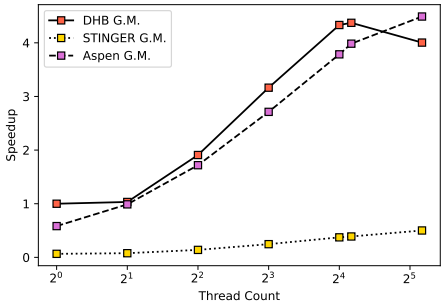


Figure 9 Memory footprint for DHB, STINGER, ASPEN, TERRACE and NETWORKIT on temporal graphs.



(a) Edge insertion rate of DHB for scaling graph size in a single-threaded execution.



(b) Geometric mean of speedups for DHB, ASPEN, STINGER on multiple threads w. r. t. a sequential run.

Figure 10 Scaling behavior of DHB w. r. t. growing number of vertices (10a) and increasing thread count (10b).

STINGER and ASPEN's speed ups for comparison. DHB scales slightly better than ASPEN while STINGER performs rather poorly. Finally, DHB's performance drops for 36 threads probably due to the NUMA issues across the two sockets of our parallel system.

Efficient Minimum Weight Vertex Cover Heuristics Using Graph Neural Networks

Kenneth Langedal[1]
University of Bergen, Norway

Johannes Langguth
Simula Research Laboratory, Oslo, Norway

Fredrik Manne
University of Bergen, Norway

Daniel Thilo Schroeder
Simula Research Laboratory, Oslo, Norway

Abstract

Minimum weighted vertex cover is the NP-hard graph problem of choosing a subset of vertices incident to all edges such that the sum of the weights of the chosen vertices is minimum. Previous efforts for solving this in practice have typically been based on search-based iterative heuristics or exact algorithms that rely on reduction rules and branching techniques. Although exact methods have shown success in solving instances with up to millions of vertices efficiently, they are limited in practice due to the NP-hardness of the problem.

We present a new hybrid method that combines elements from exact methods, iterative search, and graph neural networks (GNNs). More specifically, we first compute a greedy solution using reduction rules whenever possible. If no such rule applies, we consult a GNN model that selects a vertex that is likely to be in or out of the solution, potentially opening up for further reductions. Finally, we use an improved local search strategy to enhance the solution further.

Extensive experiments on graphs of up to a billion edges show that the proposed GNN-based approach finds better solutions than existing heuristics. Compared to exact solvers, the method produced solutions that are, on average, 0.04% away from the optimum while taking less time than all state-of-the-art alternatives.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms; Theory of computation → Randomized local search

Keywords and phrases Minimum weighted vertex cover, Maximum weighted independent set, Graph neural networks, Reducing-peeling

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.12

Supplementary Material
Software (Source Code): `https://github.com/KennethLangedal/GNN-MWVC`
Dataset (Results): `https://github.com/KennethLangedal/MWVC-GNN-LS`

Funding The work has benefited from the Experimental Infrastructure project eX3, which is financially supported by the Research Council of Norway under contract 270053.
Kenneth Langedal: Supported by the Research Council of Norway under contract 303404.
Johannes Langguth: Supported by the Research Council of Norway under contract 303404.
Daniel Thilo Schroeder: Supported by the Research Council of Norway under contract 303404.

[1] Corresponding author

1 Introduction

Consider an undirected graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. A vertex cover is a set $C \subseteq V$ such that $\forall\{u, v\} \in E \implies u \in C \vee v \in C$, or with words, that each edge has at least one of its endpoints in C. The minimum vertex cover problem (MVC) is to find a vertex cover where $|C|$ is minimized. The minimum weighted vertex cover problem (MWVC) includes a positive weight $w : V \to \mathbb{R}^+$ for each vertex and the problem is then to find a vertex cover where $\sum_{u \in C} w(u)$ is minimized. The decision version of MVC was one of Karp's original 21 NP-complete problems and it thus follows that MVC is NP-hard [13]. MWVC is at least as difficult as MVC since the case where the weights are all one is precisely the same as the unweighted version.

It is well known that kernelization techniques can speed up the computation of solutions to NP-hard problems. They shrink the instance by applying reduction rules so that an optimal solution for the reduced instance can be expanded to an optimal solution for the original instance. Several such reduction rules have been developed both for the MVC and the MWVC problem. Using reduction rules, Lamm et al. were able to solve large instances of both MVC and MWVC with millions of edges and vertices [8, 10, 18].

Reduction rules have also been used to speed up heuristics for computing vertex covers and solving associated problems. It is straightforward to see that such rules can be used as a preprocessing step before running an iterative search heuristic [19]. However, they can also be used in combination with heuristics that classify vertices when no reduction rules apply. This technique was introduced by Chang et al., who named it *reducing-peeling* [6]. Examples of such methods that have been used for MVC include genetic algorithms [17] and graph neural networks [21].

We present a hybrid approach that combines several strategies to create an effective heuristic for the MWVC problem. First, we use a combination of reduction rules, a graph neural network, and an exact solver to compute an initial vertex cover. The vertex cover is then further enhanced using an improved iterative search strategy. The proposed heuristic is compared to the best existing methods, yielding the following main results:

- We demonstrate the first successful application of graph neural networks on the MWVC problem.
- We give an improved local search implementation for MWVC on very large sparse graphs.
- On instances that can be solved exactly we compute solutions that are on average 0.04% heavier than the optimal ones.
- We obtain consistently better solutions than existing heuristics for the MWVC problem.
- Finally, we give results on several hundred graphs from SuiteSparse, including instances with more than 1 billion edges, which is significantly larger than those computed in previous efforts. Even at these sizes, our proposed heuristic finds vertex covers in less than one hour using a standard CPU.

In the remainder of the paper, we first introduce the main concepts along with previous work in Section 2 and present our approach in Section 3. Sections 4 and 5 present our experiments along with their results, while Section 6 concludes the paper.

2 Background and Related Work

The MWVC problem has several real-world applications, including dynamic map labeling [2], biological network alignment [1], and network engineering [26]. Furthermore, the problem of finding a minimum vertex cover is equivalent to the problem of finding a maximum

independent set. For any feasible vertex cover C, the vertices not in the vertex cover $V \setminus C$ make an independent set, and if $|C|$ is a minimum vertex cover, then $V \setminus C$ is a maximum independent set. This also extends to the weighted versions of these problems. Thus, results for the maximum weighted independent set problem (MWIS) carry over to MWVC.

When constructing a cover C for a graph $G = (V, E)$, each decision whether a vertex $u \in V$ should be in C or not has immediate implications for its neighborhood $N(u)$, i.e. the set of vertices adjacent to u. The resulting graph G' is smaller and can be solved independently from previous decisions. If u is added to C, then every edge connecting u to the rest of the graph will be covered. From that point, the problem is to find an MWVC on $G' = G \setminus \{u\}$. Similarly, if u is excluded from C, it follows from the definition of a vertex cover that $N(u)$ must be in C. In that case, u and its neighborhood can be removed from the graph, yielding $G' = G \setminus N[u]$, where $N[u] = N(u) \cup \{u\}$.

This observation points to two immediate questions. First, how to decide for a vertex u whether to include u or $N(u)$, and second in which order such decisions should be made. In the following, we discuss the principal ways in which this has been done previously.

2.1 Reduction Rules

There has been extensive research on computing exact solutions to NP-hard problems using kernelization combined with various branch-and-bound (BB) techniques. BB makes temporary decisions but is able to backtrack in order to find an exact solution.

The currently best exact solver for the MWVC problem is the branch-and-reduce solver B & R by Lamm et al. [18]. B & R is actually an MWIS solver, but as stated earlier, it can be directly applied to solve the MWVC problem as well. B & R selects vertices for branching based on degree, breaking ties based on weight. Clique covers are used to find an upper bound for the optimal solution to prune the search. What extends it from branch-and-bound to branch-and-reduce is the addition of reduction rules. Before each branch, the remaining graph is checked to see if any reduction rules can be applied to the current remaining graph. It is crucial to check before each branch, not only the first, since branching on a vertex and temporarily labeling it can enable further reductions. The graph is also checked for connectivity after applying reduction rules. If there are multiple connected components, these are solved separately, combining the partial results afterward. Reduction rules are not always applicable, but they are exact in the sense that decisions taken by reduction rules never prevent an optimum solution. By using an extensive set of reduction rules, B & R can often find minimum weight vertex covers on graphs with millions of vertices in a reasonable amount of time.

Our heuristic makes use of the same reduction rules as B & R. Therefore, only an outline of the main ideas is provided here. At a high level, reduction rules come in two varieties. Rules of the first type, called *removal*, directly decide whether vertices should be added to the cover and remove vertices or neighborhoods from the graph. Rules of the second type, called *folding*, also reduce the size of the graph, but without making immediate decisions about vertices. The idea is that after solving the reduced graph, it can be unfolded to extend the solution to the original graph. The following rules are examples. Other rules used in our heuristic are described by Lamm et al. [18].

Neighborhood Removal. If the weight of a vertex is greater than or equal to the combined weight of its neighborhood, i.e, $w(u) \geq w(N(u))$, then some MWVC C includes $N(u)$ and not u. To see this assume $u \in C$. Then the cost of the solution will not increase if u is replaced by $N(u)$.

Neighborhood Folding. Let u be a vertex such that no vertices in $N(u)$ are adjacent and let $v \in N(u)$ be a lightest neighbor. If $w(u) < w(N(u))$ but $w(u) \geq w(N(u) \setminus v)$ then some MWVC includes exactly one of u and $N(u)$. This follows since if a vertex cover includes u and at least one vertex from $N(u)$, then swapping u for the remaining vertices in $N(u)$ will not increase the cost. Unlike the previous rule, there is still the possibility that an MWVC could include u and no vertices from $N(u)$. Therefore, $N(u)$ cannot directly be classified yet. Instead, $N[u]$ is folded into a new vertex u' connected to every vertex adjacent to $N(u)$. The weight of this new vertex is set to $w(N(u)) - w(u)$. The new vertex u' represents the choice between $N(u)$ and u. Choosing $N(u)$ costs more than u, but could cover more edges. To unfold the reduced graph, include $N(u)$ if u' was part of the solution and u otherwise.

Note that if there had been any edges between vertices in $N(u)$, these edges would need to be covered. Therefore, selecting u and no vertices from $N(u)$ would not be an option, and $N(u)$ should be added and its neighborhood removed as in the first rule.

Typically, different reduction rules are executed in a fixed order, checking the applicability of the rule for each vertex before moving on to the next rule. The precise order is based on the computational cost associated with each rule, and the idea is to apply the cheaper rules frequently and the more expensive ones less often. Whenever a rule successfully reduces the graph, the vertices whose neighborhoods have changed are checked again, starting from the least expensive rule.

2.2 Local Search

Since the MWVC problem is NP-hard, only heuristics and approximation algorithms are feasible for large instances. Popular heuristics include genetic algorithms [22], ant-colony approaches [12], tabu search [27], and local search [3, 19, 20, 23]. Among these, local search heuristics are the most successful.

Consequently, local search plays an essential part in our proposed approach. These heuristics efficiently search for improvements to an existing solution, with a predictable running time for each iteration of the search. Previous studies show that local search can quickly find high-quality solutions and scale to graphs with several million vertices and edges [20].

Algorithm 1 Local search overview, outlining the core ideas used by several iterative local search procedures for the MWVC problem.

```
1:  C ← ConstructWVC                          ▷ Construct the initial vertex cover
2:  C' ← C                                    ▷ Best vertex cover found
3:  while elapsed_time < max_time do
4:      Remove vertex u with lowest score(u) from C
5:      while C is not a vertex cover do
6:          Add vertex v with highest score(v) to C
7:          Add 1 to the weight of each uncovered edge
8:      end while
9:      if w(C) < w(C') then
10:         C' ← C
11:     end if
12: end while
```

Most of the heuristic algorithms utilize an edge weighting strategy that dynamically changes as the search proceeds [3,20,27]. The edge weights, denoted by $edge_w$, are initialized to one. We show the common strategy of these heuristics in Algorithm 1. Here

$$cost(C) = \sum_{\{u,v\} \in E | u \notin C \wedge v \notin C} edge_w(\{u,v\})$$

$$dscore(u) = \begin{cases} cost(C \setminus \{u\}) - cost(C), & u \in C \\ cost(C) - cost(C \cup \{u\}), & u \notin C \end{cases}$$

$$score(u) = \frac{dscore(u)}{w(u)}.$$

In addition to the procedure described so far, all heuristics mentioned above make use of configuration checking, which was first introduced by Cai et al. [4]. This aims at preventing a vertex that was recently removed or added to the solution from going back to its previous state in the next iteration. Only after some other change has occurred in its neighborhood will it be allowed to change state again. Incorporating configuration checking can be as simple as defining a Boolean flag for each vertex. Only vertices with set flags are eligible for selection (Line 4). When a vertex is added to C (Line 6), the flag is set to false, but its neighbors' flags are flipped to true.

Each search iteration always starts and ends with a valid vertex cover. However, the solution quality is not guaranteed to improve during every iteration. This relaxation is necessary to escape local minima since restricting the search to only make changes that improve the solution cost will cause it to get stuck quickly. To better understand how local search works, consider the initial iteration when every edge weight equals one. The first step of each iteration removes a vertex from C with the lowest score value. The score of a vertex $u \in C$ is the weight of every edge that u alone covers, divided by $w(u)$. As a sanity check, if u is redundant and could be removed without leaving any edges uncovered, then $score(u)$ would be zero. In general, vertices with high weight and a low number of covered edges will have low scores and are likely candidates for removal. The idea being that these vertices contribute less to the overall solution. After removing a vertex, the neighbors that are not part of the solution are added back in order of their score, increasing the weight of the uncovered edges along the way. The effect of increasing the edge weights is that a newly added vertex gets higher scores than it would otherwise, and is therefore less likely to be chosen for removal. This scheme effectively handles the balance between intensifying and diversifying the search and has been demonstrated to be an efficient technique in practice [3, 19, 20, 27].

Many successful heuristics have used the technique outlined in Algorithm 1. One such heuristic, FastWVC [20], improved performance on large graphs using a new construction procedure and exchange step that removed two vertices instead of one. This was further improved in DynWVC2 [3] by using dynamic strategies for vertex selection. Another heuristic, Hybrid Iterated Local Search (Hils) [23], alternated between efficient neighborhood swaps and random permutations to balance quality and diversity. NuMWVC [19] used simple reduction rules to construct the initial solution, improved configuration checking, and dynamic vertex selection strategies. The most recent heuristic named Master-Apprentice Evolutionary Algorithm with Hybrid Tabu Search (MAE-HTS) [27] showed further improvements over NuMWVC.

2.3 Graph Neural Networks

Graph neural networks (GNNs) [25] are machine learning architectures adapted to handle graph data. There are several issues with conventional neural network architectures when working on graphs. For instance, typical convolutional neural networks (CNNs) [15] work on images of fixed size. However, graphs differ in size and cannot easily be scaled. Furthermore, results for graphs should be invariant to different vertex permutations, which CNNs do not normally support. GNNs overcome these limitations and, following the success of CNNs, neural network architectures for graphs have recently made significant progress on combinatorial optimization problems [5]. GNN models can be trained for several different tasks, including vertex labeling, edge prediction, or whole graph predictions. GNNs can learn structural information about graphs since they consider neighborhood information for each vertex.

Starting from a feature vector on each vertex, the GNN propagates information along the edges of the graph for a fixed number of rounds. When finished the GNN outputs a value for each vertex that indicates if this vertex should be part of the solution or not. Values that are being moved between two neighbors are processed using a multi-layered perceptron that has been trained on graph instances where an optimal solution is known. Like local search, decisions taken by the GNN are not exact. A previous study on the MVC problem demonstrated the effectiveness of graph neural networks in this setting, but at a significantly smaller scale [21].

3 Approach

Our proposed heuristic consists of two stages, where the first computes a vertex cover using reduction rules, output from a GNN model, and an exact solver. In the second stage, the obtained vertex cover is improved using an efficient local search procedure. In the following, we outline the stages in more detail.

3.1 Initial Computation

The first stage starts with all vertices unclassified. It then follows a greedy strategy where reduction rules are used whenever possible to classify vertices, thereby reducing the size of the remaining unsolved graph. Classified here refers to the decision made for a vertex, regardless of whether it is in the vertex cover or not. A connectivity check is performed when no reduction rule applies, and any sufficiently small component is solved exactly using a branch-and-reduce strategy. For larger components that cannot be reduced further, the vertex with the highest probability of being either in or out of the solution is classified, according to the GNN's evaluation. This procedure is repeated until a complete vertex cover for the whole graph has been obtained.

As the computation progresses, the size of the remaining unclassified graph shrinks. As a consequence, the predictions from the GNN will also change. However, computing predictions from the GNN has a cost that is linear in the size of the remaining graph, and thus, it can be expensive to apply this too often. Therefore, it is essential to decide when to update the probabilities by rerunning the GNN and when to use probabilities from the last GNN computation. The following scheme is used to decide when to recompute the probabilities from the GNN.

Each application of the GNN gives a prediction for each vertex regarding whether or not it should be part of the vertex cover. The list of predictions is ordered by decreasing assurance. When consulting this list, the top choice might already have been classified by a reduction rule. If the current configuration of that vertex and the prediction by the GNN disagree, it is an indication that the predictions should be updated. This could still occur often, so it is also required that the remaining graph's size has shrunk sufficiently since the last time the GNN output was computed.

Like the GNN computation, checking the graph's connectivity also has a cost that is linear in the size of the remaining graph and is therefore only done before each GNN computation. Small connected components are solved exactly using a simplified version of the solver presented by Lamm et al. [18]. Due to the limited input size, some aspects have been omitted, such as branch elimination based on lower and upper bounds and connectivity checking. Instead, it is more important to target the worst-case scenario, which seems to occur in very dense graphs. To this end, a degree-based branching technique is used, as described by Imamura and Iwama [11], and based on an observation by Karpinski and Zelikovsky [14]. The idea is that for some set $S \subseteq V$ with $|S| = k$, where $\forall u \in S \implies degree(u) \geq k$. Then an optimal solution C will either include the whole set $S \subseteq C$ or the entire neighborhood for one of the vertices $\exists u \in S : N(u) \subseteq C$. In a branch-and-reduce setting, this means that one can branch based on $k + 1$ options, and each of these branches will already have at least k vertices classified. This should be compared to a traditional branching on a single vertex, where one branch typically only reduces the size of the problem by one.

To recap the construction procedure, first, apply reduction rules. Then check if the size of the graph has been sufficiently reduced. If so, perform a connectivity check, solve small connected components exactly, and apply the GNN model to the rest. If the size of the graph has not decreased enough, consult the most recent GNN predictions. Finally, when the size of the remaining graph reaches zero, unfold the graph up until the first non-exact decision was made. The last unfolding only occurs after local search.

3.2 Graph Neural Network Architecture

The GNN architecture used is a combination of message-passing steps interleaved with multi-layered perceptrons (MLPs). The message-passing step is inspired by the layer-wise propagation rule presented by Kipf and Welling [16]. It is slightly modified using a few handcrafted features and a direct passthrough, similar to a residual link used in CNNs [9]. The message-passing step then looks like this:

$$H^{l+1} = AH^l|H^l|D|W|N.$$

Here, H^l is the $|V| \times f$ matrix at layer l, where f is the number of activations for each vertex at this layer and A is the graph's adjacency matrix. Then, H^l is concatenated unchanged, followed by D, W, and N, who are $|V| \times 1$ matrices corresponding to vertex degree, weight, and neighborhood weight. The output H^{l+1} then becomes a $|V| \times 2f + 3$ matrix. Initially, $f = 1$ since the only attribute of a vertex is its weight.

The MLPs used are extensions of the original perceptron model introduced by Rosenblatt [24]. Each layer is a dense matrix of trainable weights, and in between the layers are non-linear activation functions. The forward flow through one of these layers is given by:

$$H^{l+1} = \sigma(H^l W^l + b^l)$$

where σ is the activation function, W^l are the trainable weights, and b^l is the bias term at this layer. The dimensions of W^l is $f^l \times f^{l+1}$, meaning these layers can scale f as desired. The vector b^l is added to each row of $H^l W^l$. The activation functions used are $ReLU$ and $sigmoid$. These are elementwise functions defined as:

$$ReLU(x) = max(0, x)$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}}.$$

The model used in the proposed heuristic consists of three message-passing layers interleaved with three-layer MLPs. The MLP layers consist of 32 activations with a single activation for each vertex at the output layer. ReLU is used between every MLP layer, except in the output layer, where sigmoid is used. The benefit of the sigmoid activation in the output layer is that each value can be interpreted as a probability.

3.3 Optimized Local Search

For the second stage, a local search procedure is employed, similar to that shown in Algorithm 1. However, this contains three costly operations that need to be analyzed carefully.

1. Finding the next vertex u to remove (Line 4)
2. Reconstructing the vertex cover (lines 5-7)
3. Storing the new solution if it turns out to be an improvement (Line 10)

When done naively, the first and third points will take $O(|V|)$ time and the second $O(|N(u)|^2)$, where u is the vertex from Line 4. This might not be too costly if the graph is small and the current solution is close to the optimal. However, since our aim is to find vertex covers on massive graphs with millions of vertices, a more careful implementation could make a significant difference.

To address the linear cost of finding the next vertex to remove, we store every vertex in the graph in a binary heap. Whenever the score of a vertex changes, its position in the heap is also updated. In one remove and reconstruct step, the neighborhood of u and the neighborhood of every added vertex will change their *score* value. This means that the cost of maintaining the heap during each iteration will be $\log(|V|)$ times the size of the distance-2 neighborhood of u.

The next point is how to reconstruct the solution efficiently. Since u is not allowed to enter the solution again in the same step as it was removed, the vertices with a positive *score* are precisely the neighbors of u that are currently not in the vertex cover. Furthermore, since these vertices are not part of the vertex cover, every other neighbor they might have besides u, will already be in the vertex cover. This means that the score value for each $v \in N(u)$ where $v \notin C$ will be equal to the weight of its edge incident on u divided by $w(v)$. To improve the speed of reconstructing the vertex cover, we sort the adjacency list of u based on edge weight and then add them in that order. This improves the running time for this step from $O(|N(u)|^2)$ to $O(|N(u)| \log(|N(u)|))$. Reconstructing the vertex cover this way is not equivalent to Algorithm 1. However, it is faster and based on our experiments, has negligible impact on solution quality.

Finally, the last consideration is how to keep track of the best solution found. The best solution is not used for anything during the execution of the algorithm. It is only stored to be returned at the end. If the initial solution is far from the eventual local minima, repeatedly storing the improved vertex covers could become a bottleneck. To address this issue, multiple iterations are performed without checking if the solution quality has improved. Initially,

the number of iterations is high, but gradually shrinks as the search continues. This idea is not new, as NuMWVC uses a similar technique called *self-adaptive-vertex-removing* [19]. However, in NuMWVC, the authors changed the number of vertices removed during each iteration, unlike here, where only the updating is omitted. Compared with NuMWVC, our approach also varies significantly in scope. NuMWVC starts by removing three vertices and gradually moves down to one, whereas in our approach, the idea is to let the search run for thousands of iterations without updating the best solution. It is important to note that these improvements only speed up the search and does not necessarily lead to solutions that other existing heuristics would not have found given sufficient time.

4 Experimental Setup

In the following, we present the computation platform, benchmark datasets, and details on how the GNN model was trained. We refer to our heuristic as GNN with local search (GNN & LS). The exact solver is employed on connected components with ≤ 75 vertices and requires a 5% size reduction before each connectivity check and recomputation of GNN probabilities. We require a minimum of 1024 iterations during the local search before updating the best solution.

Computing Platform. All heuristics are implemented in C++ and compiled using GCC 9.3 with the O3 optimization flag. All the experiments were run on a single thread of an Intel Xeon Silver 4112 CPU with 2.60 GHz and 38 GB of memory. The machine runs Ubuntu 18.04.6 and Linux kernel version 5.4.0-109.

4.1 Benchmark Data

We use graphs from the SuiteSparse collection [7]. In order to get graphs with a wide variety of meaningful sizes and densities, a subset of the entire SuiteSparse collection was used, selected on the basis of file size. The first set of graphs, Dataset 1, contains graphs with a file size between 40 MB and 4 GB, 371 graphs in total. The smallest graphs start at roughly 500 thousand edges, while the largest exceed 180 million[2]. Most of these graphs do not initially have vertex weights or undirected edges, so the graphs are first converted to the correct format. This is done by considering each edge as undirected, removing any duplicates or self edges, and then generating vertex weights uniformly at random. There are some differences in previous studies on how weights are assigned. For instance, weights in the integer range [20, 100], [20, 120], or [1, 200] have all been used. We use weights drawn uniformly at random in the integer range [1, 200], which is similar to B & R [18] and HILS [23]. However, based on preliminary experiments, different weight ranges did not significantly impact the results.

Some of the other heuristics used for comparison contain built-in input size limitations. For instance, NuMWVC only accepts graphs with less than 9 million vertices. Therefore, Dataset 2 consist of the graphs from Dataset 1 that every heuristic accepted as input. The exact solver B & R [18], in its default configuration and with a time limit of 1200 seconds, was able to find exact solutions for 111 graphs from Dataset 1. These 111 graphs constitute Dataset 3. Lastly, six massive graphs with more than one billion edges are also included as Dataset 4.

[2] A complete list of graphs along with results and source code can be found at this repository: `https://github.com/KennethLangedal/MWVC-GNN-LS`

4.2 Training the GNN Model

When training a GNN model directly on vertex classification, one has to take into consideration that a minimum weighted vertex cover might not be unique. One idea that has already been successfully used for the MVC problem is to have multiple vertex covers as output and use a hindsight loss that only acts on the best output [21]. We propose another approach for MWVC, where one increases the weight range to lower the chance for multiple optimal solutions. Therefore, the weights assigned to the training data are drawn uniformly at random from the integer range [10, 2000] instead of [1, 200]. The input to the model is then linearly scaled down to the real interval [0, 1], to ensure that the data is in the same range when training as during inference. Roughly 1400 graphs from the SuiteSparse collection with sizes less than 40 MB were used as training data. The exact solver B & R [18] was able to find optimal solutions for 929 of these graphs, thus giving us over 8 million labeled vertices that we could then use for training.

Mean squared error (MSE) and stochastic gradient descent (SGD) with momentum were used to fit the model's parameters to the training data. The training data was divided into 90% for training and 10% for validation. Additional parameters used during the training include a 0.01 learning rate, 0.9 momentum, and a batch size of 250,000 vertices. The results from the training can be seen in Figure 1.

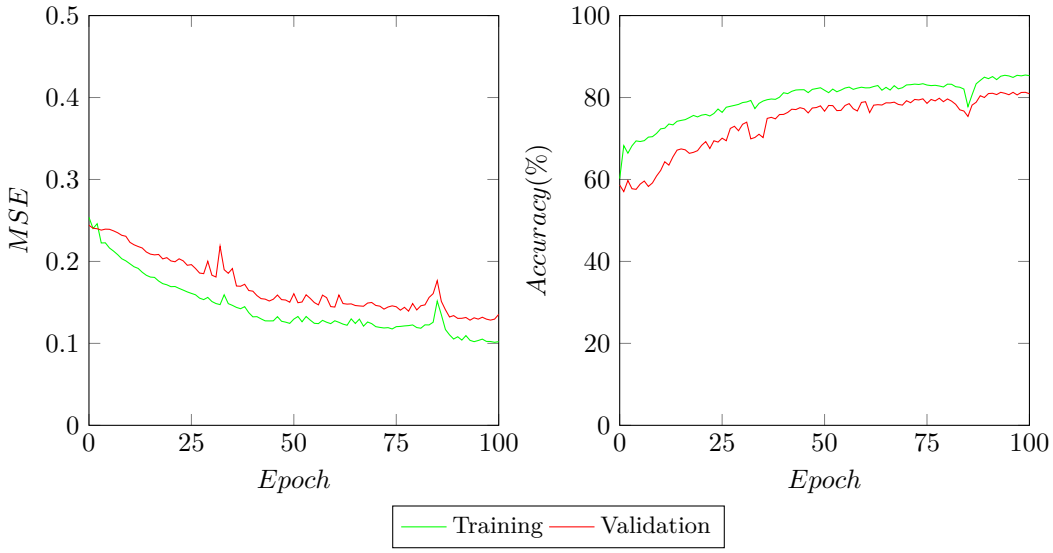


Figure 1 GNN training over 100 epochs, showing MSE and accuracy for training and validation sets.

5 Experimental Results

In this section, we report on a set of experiments to gauge the performance of our new GNN-based approach. We start with results from Dataset 2 compared to other state-of-the-art heuristics. Then, based on Dataset 1 and 3, we present a deeper analysis of the different components of our GNN-based approach. Finally, we include results on Dataset 4.

5.1 Comparison with State-of-the-art Heuristics

The first set of experiments compares GNN & LS with the following heuristics: DynWVC2, Hils, FastWVC, and NuMWVC. The source codes for all of these are available online, except NuMWVC, where the authors provided the code. There is one newer heuristic, MAE-HTS [27], but we were unable to obtain the code for it. Each heuristic ran for 1000 seconds, while recovering the best solution found. Dataset 2 was used here due to the built-in limitations on some implementations. Figure 2 shows the solution quality compared to the best solution found by any of the heuristics on each graph. The y-axis gives the percentage of graphs that a heuristic was able to solve with increasing distance to the best solution given by the x-axis. Here, the distance is measured as a percentage of the best solution. Thus the values initially show the percentage of graphs where each heuristic gave the best solution. The absolute numbers are also given in the first row of Table 1.

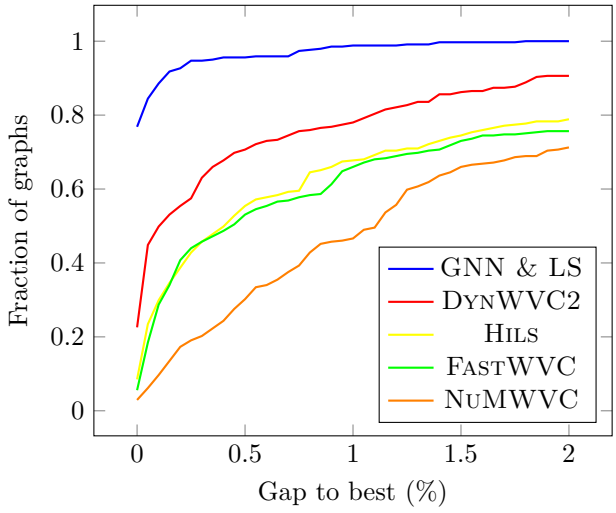


Figure 2 Solution quality on Dataset 2, based on the gap to the best solution found by one of the heuristics.

Table 1 Summary of results on Dataset 2.

	GNN & LS	DynWVC2	Hils	FastWVC	NuMWVC
Best solutions	262	77	29	19	10
Average gap to best	0.06%	0.84%	2.37%	1.44%	2.33%
Average time in sec.	480.79	735.83	907.9	826.75	669.91

As shown in Table 1, GNN & LS finds higher-quality solutions on significantly more test instances than the other heuristics. It is also, on average, closest to the best solution and uses the least time.

Table 1 gives the time it took to find the reported solution. Measuring the running time this way does not necessarily give meaningful insight into how fast the different methods are. A heuristic that finds a good solution quickly but then makes a small improvement after a long time would register as slow in this metric, while one that finds the same solution but does not make the improvement would register as fast, even though it never had a better solution than the first heuristic. In order to get a better understanding of the speed of the heuristics, we run the programs for a shorter duration and compare the results, as shown in

Figure 3. The GNN-based approach spends considerable time constructing the initial vertex cover, 27.7 seconds on average. Before that point, GNN & LS has nothing to report. Figure 3a shows the results when using the time the GNN spent constructing the initial solution as the time limit for the other programs, effectively testing the GNN construction part in isolation against the other heuristics.

Again, as can be seen from the results, GNN & LS also find higher-quality solutions with these lower time limits. Figure 3a shows that the advantage of the new heuristic is even greater when comparing the construction step alone against the other heuristics. A reason for the drop in performance from Figure 3a to 3b is due to graphs where GNN & LS did not finish constructing the initial vertex cover in the first 100 seconds.

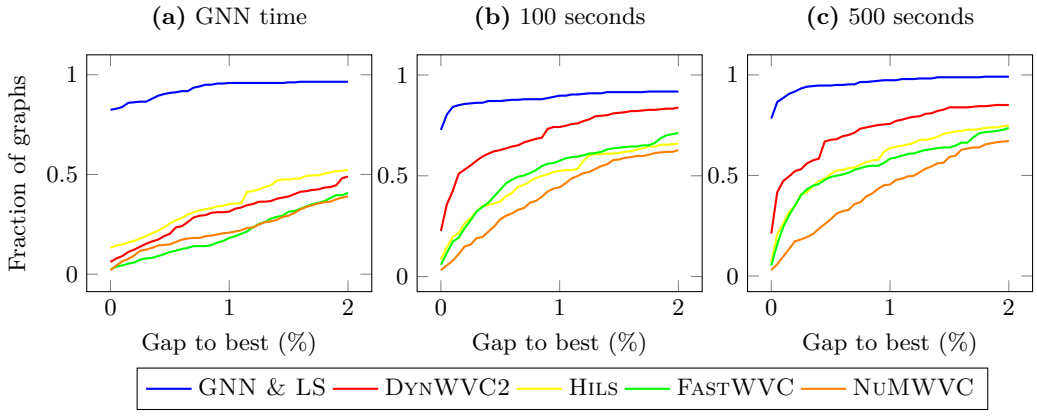


Figure 3 Solution quality on Dataset 2 using different time limits. Quality is measured based on the gap to the best solution found by one of the heuristics.

5.2 Evaluation of Different Configurations

In this section, we perform a deeper evaluation of the different components of our heuristic in isolation, with the main focus on the GNN component. The GNN's part is to label one vertex when reduction rules fail to make progress. One sensible alternative is to exclude the heaviest vertex, breaking ties based on degree, similar to how exact solvers pick a vertex to branch on. This modified version of GNN & LS will be referred to as Quick & LS. Additionally, to measure the importance of local search, both GNN and Quick without local search are also included. Lastly, a pure local search (LS) is used for comparison. These different configurations are evaluated w.r.t. running times and solution quality, including results on graphs where the optimal solution is known.

Table 2 Summary of results on Dataset 1.

	GNN	GNN & LS	Quick	Quick & LS	LS
Best solutions	65	242	57	122	77
Average gap to best	0.68%	0.01%	2.52%	0.22%	0.95%

When comparing different configurations of the proposed heuristic, GNN & LS wins on both the number of best solutions and the average gap to the best solution, as shown in Table 2. Taking away components from GNN & LS all lead to worse performance, but to different extents. Taking away the GNN, represented by Quick & LS, is the second-best

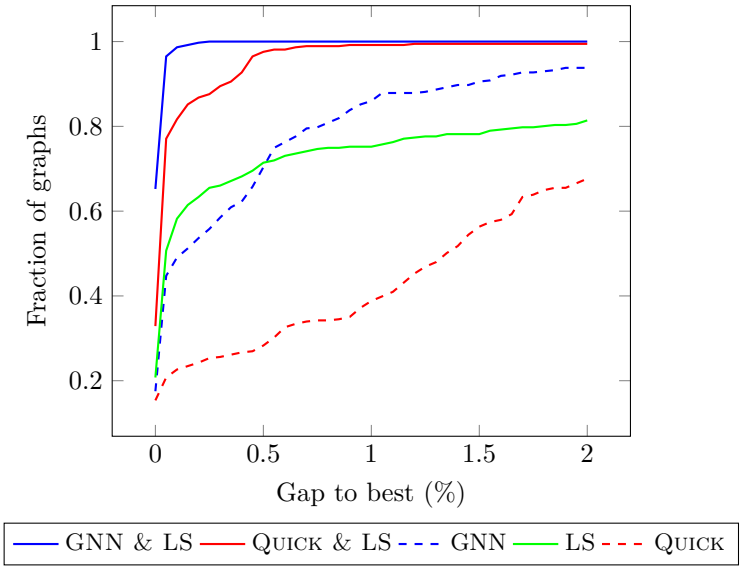


Figure 4 Solution quality on Dataset 1, based on the gap to the best solution found by one of the configurations.

configuration and shows that using reduction rules during the construction of the initial solution benefits the subsequent local search. Comparing the GNN and Quick results, both without local search, highlights the impact of the GNN model at this stage. Ultimately, the GNN, local search, and reduction rules are all responsible for significant parts of the quality of the final vertex cover.

Table 3 Summary of results on Dataset 3.

	GNN	GNN & LS	Quick	Quick & LS	LS
Optimal solutions	52	57	52	56	10
Average gap to optimal	1.12%	0.04%	3.27%	0.60%	1.77%

On Dataset 3, GNN & LS is on average closest to the optimal solutions, shown in Table 3, while the other configurations are noticeably worse. In order to find optimal vertex covers on graphs of these sizes, the graphs need to be especially amenable to reduction rules. Counting the number of vertices remaining after the initial reduction step confirms this, as 43 graphs were completely solved by reduction rules alone. On the graphs that had more than zero vertices left, the average on Dataset 3 was a reduction of 46.44%, while for Dataset 1, it was 26.7%. These numbers further demonstrate the power of reduction rules on the MWVC problem.

5.3 Running Time

So far, the focus has been on solution quality within a fixed time window. Since Dataset 1 contains several hundred graphs of various sizes, condensing this data to a single number or figure can oversimplify the results. For instance, on some graphs, the local search could quickly get stuck in a local minimum, while on others, an improved solution could be found after a long time. The interesting feature is how the solution quality changes over time. For example, the GNN-based approach takes considerable time to construct the initial solution.

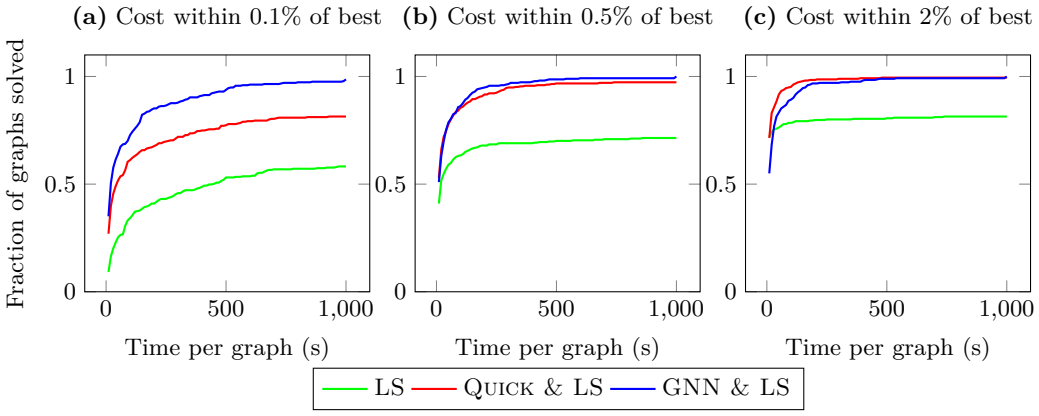


Figure 5 Performance profile on Dataset 1 showing the fraction of graphs solved over time, including different definitions of solved. Before GNN & LS and Quick & LS have solutions to report on a graph, it counts as not being solved regardless of defined threshold.

However, if that solution is better than what a pure local search could do in the same amount of time, it would likely be worthwhile. The same argument can be made for the Quick alternative as well. However, there is no guarantee that this is the case, and a worse initial solution can produce a better final solution after local search. To show how the solution quality changes over time, we ran each configuration for 1000 seconds, reporting the best solution at 10-second intervals.

The results, seen in Figure 5, show that there is a time/quality tradeoff that comes with the GNN component. When the best solution quality is desired, GNN & LS is the best choice (Figure 5a). However, since GNN & LS is slowest to produce a feasible solution, there comes the point where omitting the GNN component becomes beneficial, as shown in Figure 5c.

5.4 Large Instances

Table 4 Results from Dataset 4. Each configuration ran for 3000 seconds. The best solutions are indicated in bold.

	GNN & LS		Quick & LS		LS	
Graph	Cost	Time	Cost	Time	Cost	Time
webbase-2001	**3,440,309,297**	311.03	3,442,765,890	176.72	3,539,284,688	62.68
it-2004	**1,438,951,442**	749.59	1,439,091,400	490.23	1,480,088,254	76.98
GAP-twitter	**1,160,408,463**	724.04	1,160,512,688	712.48	1,201,023,583	30.18
twitter7	**1,160,187,362**	763.95	1,160,291,460	680.71	1,200,789,078	29.82
GAP-web	**1,861,729,481**	2,562.73	1,883,467,468	1,724.99	1,932,504,652	227.39
sk-2005	**1,861,502,940**	2,797.92	1,882,779,425	1,788.75	1,932,262,816	221.15

Dataset 4 contains six larger graphs to show that our GNN-based approach can scale beyond the problems sizes investigated in previous work. The results are shown in Table 4, with GNN & LS giving the best quality on all of these instances. The final solution cost of GNN & LS and Quick & LS is very similar. This is due to the effect the reduction rules has on these graphs, as shown in Table 5. They are particularly effective on GAP-twitter and twitter7, with less than 200,000 vertices left after the initial round of reductions. As a consequence, no matter how the remaining graphs are solved, the solution quality of the different methods is very close, but it is relevant that the GNN finds better solutions.

Table 5 Reduction rules on Dataset 4.

Graph	|V|	|E|	|V| after reduction
webbase-2001	118,142,155	854,809,760	4,936,598
it-2004	41,291,594	1,027,474,946	8,826,771
GAP-twitter	61,578,415	1,202,513,046	164,512
twitter7	41,652,230	1,202,513,046	165,489
GAP-web	50,636,151	1,810,063,329	16,803,173
sk-2005	50,636,154	1,810,063,329	16,756,003

6 Conclusion and Future Work

We have demonstrated that GNNs can boost the performance of heuristics for the MWVC problem. We have also introduced a local search implementation for large sparse graphs that avoids frequently occurring $O(|V|)$ steps. Our complete heuristic also incorporates previously established reduction rules and an exact solver. Extensive experiments on several hundred large graphs show that our heuristic significantly outperforms previous methods. We also demonstrate that every part of our strategy is needed to achieve these results and show that it can scale to larger graphs than previously considered, including graphs with more than 1 billion edges.

Despite our promising results, it is clear that each component of our strategy can be improved further based solely on existing work. For instance, the exact solver used on small connected components could also be significantly improved, evident by the success of solvers like B & R. The proposed improvements to local search are primarily focused on implementation, and there is undoubtedly room to include techniques from recent work in this area as well. Similarly, using a more sophisticated GNN architecture or increasing the amount of training data is likely to improve the performance. Naturally, it is also possible to apply the same strategy to new problems. This is something we intend to investigate in the future.

References

1 Ferhat Ay, Manolis Kellis, and Tamer Kahveci. Submap: aligning metabolic pathways with subnetwork mappings. *Journal of Computational Biology*, 18(3):219–235, 2011.

2 Ken Been, Eli Daiches, and Chee Yap. Dynamic map labeling. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):773–780, 2006.

3 Shaowei Cai, Wenying Hou, Jinkun Lin, and Yuanjie Li. Improving local search for minimum weight vertex cover by dynamic strategies. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1412–1418. International Joint Conferences on Artificial Intelligence Organization, July 2018. doi:10.24963/ijcai.2018/196.

4 Shaowei Cai, Kaile Su, and Abdul Sattar. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artificial Intelligence*, 175(9-10):1672–1696, 2011.

5 Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint*, 2021. arXiv:2102.09544.

6 Lijun Chang, Wei Li, and Wenjie Zhang. Computing a near-maximum independent set in linear time by reducing-peeling. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1181–1196, 2017.

7 Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1), December 2011. doi:10.1145/2049662.2049663.

8 Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdan Zavalnij. Boosting data reduction for the maximum weight independent set problem using increasing transformations. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 128–142. SIAM, 2021.

9 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

10 Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. WeGotYouCovered: The winning solver from the pace 2019 challenge, vertex cover track. In *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*, pages 1–11. SIAM, 2020.

11 Tomokazu Imamura and Kazuo Iwama. Approximating vertex cover on dense graphs. In *Symposium on Discrete Algorithms: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, volume 23, pages 582–589, 2005.

12 Raka Jovanovic and Milan Tuba. An ant colony optimization algorithm with improved pheromone correction strategy for the minimum weight vertex cover problem. *Applied Soft Computing*, 11(8):5360–5366, 2011.

13 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

14 Marek Karpinski and Alexander Zelikovsky. *Approximating dense cases of covering problems*. Citeseer, 1996.

15 Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53(8):5455–5516, December 2020. doi:10.1007/s10462-020-09825-6.

16 Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint*, 2016. arXiv:1609.02907.

17 Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F Werneck. Finding near-optimal independent sets at scale. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 138–150. SIAM, 2016.

18 Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 144–158. SIAM, 2019.

19 Ruizhi Li, Shuli Hu, Shaowei Cai, Jian Gao, Yiyuan Wang, and Minghao Yin. NuMWVC: A novel local search for minimum weighted vertex cover problem. *Journal of the Operational Research Society*, 71(9):1498–1509, 2020.

20 Yuanjie Li, Shaowei Cai, and Wenying Hou. An efficient local search algorithm for minimum weighted vertex cover on massive graphs. In *Asia-Pacific Conference on Simulated Evolution and Learning*, pages 145–157. Springer, 2017.

21 Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *arXiv preprint*, 2018. arXiv:1810.10659.

22 Sk Md Abu Nayeem and Madhumangal Pal. Genetic algorithmic approach to find the maximum weight independent set of a graph. *Journal of Applied Mathematics and Computing*, 25(1):217–229, 2007.

23 Bruno Nogueira, Rian G.S. Pinheiro, and Anand Subramanian. A hybrid iterated local search heuristic for the maximum weight independent set problem. *Optimization Letters*, 12(3):567–583, 2018.

24 Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386, 1958.

25 Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

26 Changbing Tang, Ang Li, and Xiang Li. Asymmetric game: A silver bullet to weighted vertex cover of networks. *IEEE Transactions on Cybernetics*, 48(10):2994–3005, 2017.

27 Yang Wang, Zhipeng Lu, and Abraham P Punnen. A fast and robust heuristic algorithm for the minimum weight vertex cover problem. *IEEE Access*, 9:31932–31945, 2021.

A Branch-And-Bound Algorithm for Cluster Editing

Thomas Bläsius
Karlsruhe Institute of Technology, Germany

Philipp Fischbeck
Hasso Plattner Institute, Potsdam, Germany

Lars Gottesbüren
Karlsruhe Institute of Technology, Germany

Michael Hamann
Karlsruhe Institute of Technology, Germany

Tobias Heuer
Karlsruhe Institute of Technology, Germany

Jonas Spinner
Karlsruhe Institute of Technology, Germany

Christopher Weyand
Karlsruhe Institute of Technology, Germany

Marcus Wilhelm
Karlsruhe Institute of Technology, Germany

Abstract

The cluster editing problem asks to transform a given graph into a disjoint union of cliques by inserting and deleting as few edges as possible. We describe and evaluate an exact branch-and-bound algorithm for cluster editing. For this, we introduce new reduction rules and adapt existing ones. Moreover, we generalize a known packing technique to obtain lower bounds and experimentally show that it contributes significantly to the performance of the solver. Our experiments further evaluate the effectiveness of the different reduction rules and examine the effects of structural properties of the input graph on solver performance. Our solver won the exact track of the 2021 PACE challenge.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases cluster editing

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.13

Supplementary Material
Software (Source Code): `https://github.com/kittobi1992/cluster_editing`
Software (Source Code): `https://doi.org/10.5281/zenodo.4892524`

Funding *Michael Hamann*: This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant WA 654/22-2.

Acknowledgements We thank Darren Strash for helpful discussions and literature research.

1 Introduction

In graph clustering, the goal is typically to partition the vertices into clusters such that there are many edges inside and few between clusters. The most clear-cut cases are so-called *cluster graphs* in which each connected component forms a clique. Thus, with one cluster for each connected component, there are no edges between clusters and all possible edges inside clusters exist. The *cluster editing problem* asks to use as few edge insertions and deletions as possible to transform a given graph into a cluster graph; thereby computing a clustering.

The cluster editing problem is NP-hard [18] and thus we cannot expect to solve it efficiently in general. Nonetheless there are algorithmic approaches using reduction rules [11, 12, 14] or search trees [8, 15]. The theoretically fastest known algorithm is by Böcker [7] with a running time of $O(1.62^k + n + m)$, where k is the number of edits (edge insertions plus deletions) and n, m are the number of vertices and edges of the graph, respectively. To encourage development and implementation of practical algorithms, the challenge of PACE 2021 [16] was to solve cluster editing. Our solvers won the exact [4] and heuristic [3] track.

In this paper, we describe the details of our exact solver [4] and present an in-depth evaluation. Roughly speaking our solver is a branch-and-bound algorithm: Whenever possible, we apply reduction rules to shrink the instance. When no reductions apply, we branch on the

decision whether to put a pair of vertices in the same or in different clusters. To reduce the size of the resulting search tree, we compute lower bounds on the optimal solution and prune subtrees where the lower bound exceeds the upper bound computed by our heuristic solver.

Our lower bounds are based on so-called *packings* of small substructures for which we know optimal solutions. This approach has been used before to solve related problems [13] and for cluster editing in particular by packing paths of length 3 [15]. We generalize this approach to support larger substructures and weighted[1] instances; see Section 3.2. For the reduction rules, we use various rules from the literature [2, 9, 11, 12, 13, 14] as well as newly developed ones; see Section 3.3. One type of reduction rule are so-called *forced choices* that essentially look ahead one branching step, e.g., if putting two vertices in different clusters would yield a lower bound exceeding the upper bound, one must put them in the same cluster. Thus, the lower bounds and the reduction rules are intertwined in the sense that better lower bounds lead to more applications of the forced choices rules. In Section 4 we evaluate how effective and efficient different reductions and lower bounds are, using the instances from the PACE challenge. Additionally, we evaluate our algorithm on geometric inhomogeneous random graphs [10], which lets us study scaling behavior of our solver and its efficiency depending on certain instance properties. Our main findings are summarized as follows.

- The instances we can solve are usually already solved by just the reduction rules, i.e., once we have to apply branching we usually do not find a solution within reasonable time.
- The forced choices reduction rules are by far the most effective rules. This identifies good lower bounds as the key ingredient of our algorithm.
- Using packings of stars instead of paths of length 3 is still computationally feasible and yields substantially better lower bounds.
- The solver performs better on graphs with low average degree and if the graph is well-clusterable.
- The upper bounds computed by our heuristic solver are exceptionally good. In fact, the heuristic solver found an optimal solution on all instances where we know it.

2 Preliminaries

Let $G = (V, E)$ be a simple, undirected graph. The *cluster editing problem* asks to transform G into a disjoint union of cliques with the least number of edge edit operations. An edit is the deletion of an existing edge or the insertion of a missing edge. As a graph is a disjoint union of cliques if and only if it does not contain an induced path on three vertices (a P_3), the problem can also be seen as P_3-free editing.

The *weighted cluster editing problem* replaces the set of edges with a symmetric cost function $s : V \times V \mapsto \mathbb{Z}$. If $s(uv) > 0$, the pair uv is considered an edge with a deletion cost of $s(uv)$. For $s(uv) < 0$ the pair uv is a non-edge with an insertion cost of $-s(uv)$. A vertex pair with $s(uv) = 0$ is called a zero-edge that can be inserted or deleted for free. A solution to the weighted cluster editing problem is a partition of vertices. We associate a solution K with its corresponding equivalence relation $\equiv_K$. The cost of K for the instance (V, s) is defined as the total cost of edges between clusters and non-edges inside clusters. That is,

$$\text{cost}(K, s) = \sum_{\substack{s(uv)>0 \\ u \not\equiv_K v}} |s(uv)| + \sum_{\substack{s(uv)<0 \\ u \equiv_K v}} |s(uv)|.$$

[1] Though the input is unweighted, our reduction rules as well as the branching lead to weighted instances.

3 A Branch-and-Bound Algorithm For Cluster Editing

Our algorithm uses branch-and-bound to solve the decision variant of cluster editing, which asks if there exists a solution with a cost of k or less. The optimization problem is solved by calling the decision variant with increasing values of k. At its core our algorithm is a simple recursive subroutine that computes a kernel by applying reductions, returns if the lower bound for the remaining instance is above k, and otherwise branches on the inclusion or exclusion of an edge in the solution, introducing *permanent* and *forbidden* edges into the instance. We select the edge to branch on by highest edit cost and tiebreak by the number of P_3s that overlap this edge. As outlined by Böcker et al. [7, 8], the endpoints of permanent edges can immediately be merged to obtain an equivalent weighted instance[2] of smaller size.

Since branching creates weighted instances, we initially apply a series of reductions that are only possible for unweighted instances before we run the recursive branch-and-bound algorithm. Furthermore, we split the initial instance into connected components and solve them separately because an optimal solution never connects them.

In the following we discuss the different parts of our algorithm. In Section 3.1, we mention our approach to obtain upper bounds. Section 3.2 introduces and generalizes the concept of lower bounds via conflict packings. Finally, we list the used reduction rules and explain their application in our algorithm in Section 3.3.

3.1 Upper Bounds

An upper bound for the optimal solution is crucial for any branch-and-bound algorithm to identify and prune branches that cannot lead to an optimal solution. For this, we use our heuristic solver that won the heuristic track of the 2021 PACE challenge [3]. The heuristic solutions are optimal on all 173 of the 200 test instances for the exact track we were able to solve (see Section 4). We refer to the solver description [3] for details about the algorithm.

3.2 Lower Bounds

For lower bounds, we use an idea from recent solvers for this and other similar problems [6, 13, 15]. The idea is to find a large set of vertex-pair disjoint P_3s. Recall that cluster editing can be seen as P_3-free editing. We call such a set a *conflict-packing* or *packing* for short. Since each P_3 in a packing needs at least one edit to resolve and no edit overlaps with more than one conflict, the size of the set is a lower bound on the required number of edits.

Finding a maximum disjoint set of conflicts is an independent set problem, which is hard to solve in general. We are not aware of complexity results for independent set on this specific kind of intersection graph. Hartung et al. [15] use the commonly known *small degree* heuristic and some random perturbation to find a maximal set of P_3s. Gottesbüren et al. [13] also use the concept of a conflict packing in their algorithm for the quasi-threshold editing problem, i.e., $\{C_4, P_4\}$-free editing. They propose to use a local search with random replacements and the 2-improvement heuristic for independent set to grow the packing [1].

With these heuristics, good P_3 packings can be found in reasonable time. However, P_3 packings have two major drawbacks. First, they are defined only for unweighted instances while the best known branch-and-bound algorithms for cluster-editing work on weighted instances [7, 8]. Second, each P_3 has two edges and one non-edge. Therefore a P_3 packing can never be larger than $|E|/2$ while many difficult instances require more than $|E|/2$ edits. We propose a framework to circumvent both these drawbacks by generalizing conflict packings.

[2] Forbidden edges have a weight of negative infinity.

We call two cost functions $a, b : V \times V \mapsto \mathbb{Z}$ *conflicting* if there is a vertex pair uv that is a non-edge in (V, a) and an edge in (V, b) or vice versa, i.e., if $|a(uv)+b(uv)| < |a(uv)|+|b(uv)|$. Let $a + b$ denote the element-wise addition of the functions a, b, that is, $(a + b)(uv) = a(uv) + b(uv)$. We call a set of cost functions P a *packing* for the instance (V, s) if (1) they are pairwise non-conflicting, (2) they are non-conflicting with s, and (3) they do not exceed s in any vertex pair, that is $\sum_{p\in P} |p(uv)| \leq |s(uv)|$. Note that $\sum_{p\in P} |p(uv)| = |\sum_{p\in P} p(uv)|$ because of property (1). Also note that property (1) actually follows from (2) and (3).

▶ **Theorem 1.** *For packing P of the instance (V, s), $\sum_{p\in P} opt(V, p) \leq opt(V, s)$.*

The theorem states that we can pack structures together and sum their lower bounds to obtain a lower bound for the initial instance. A P_3, for example, is represented by a cost function that is zero throughout except for its three (non-)edges. Therefore, the concept generalizes P_3 packings to weighted instances. Moreover, our formulation of a packing allows for other structures than P_3s. Recall that P_3 packings have the drawback that they cannot exceed $|E|/2$. To remedy this, we have to find other structures that have a better lower bound to edge ratio. Actually, a star S_k with k leaves (thus k edges and $\binom{k}{2}$ non-edges) cannot be solved with less than $k - 1$ edits. Coincidentally, a P_3 is a star with two leaves. One can even generalize from stars to complete bipartite graphs $K_{a,b}$ which cannot be solved in less than $a \cdot (b-1)$ edits. A star S_k is just a $K_{1,k}$. So there is a tradeoff between structures that are easy to find and pack and structures that have strong lower bounds.

We implemented a P_3 packing, a star packing, and a $K_{a,b}$ packing. In preliminary experiments, we observed that the quality of star and $K_{a,b}$ packings were similar while star packings were easier, and thus slightly quicker, to compute. We thus focus on star bounds in the following. Our implementation of the star packing builds upon the P_3 packing described by Gottesbüren et al. [13]. They go through all items in the packing and try to replace one with two currently not in the packing. To not get stuck in a local optimum, they also randomly replace an item with one other item with a small probability when it cannot be replaced by two new ones. We make three major changes. First, we introduce more mutations that change the lower bound by exactly one. For P_3s, the packing grows by removal of one P_3 and insertion of two new ones in its place. We never insert or remove stars with more than two leaves. Instead, we add the option to add/remove a leaf. Second, when possible we merge a star with another existing star instead of mutating it. The merge increases the lower bound of the packing by one. Third, we relax the termination condition for the local search. They stop if the packing does not grow for five iterations. In contrast, we continue while the average number of improving iterations is still above one in five, i.e., five times the number of improving iterations is greater or equal the number of total iterations. This leads to better packings for instances that benefit from longer local search while still being fast on instances that quickly hit a local maximum. Finally note that the packing is weighted but we only pack or modify unweighted structures. For performance, however, we associate an integer weight with each star to represent multiple identical overlapping stars.

3.3 Reduction Rules

There exist various reduction rules [2, 9, 11, 12, 13, 14] and we introduce additional ones (Forced Choices Single Merge and Clique-Like Subgraph). In the following, we discuss the reduction rules used by our solver and go into detail on how our solver applies the rules.

Twin Simple [14]. This rule merges vertices with identical neighborhoods and is part of the unweighted $4k$ kernel based on critical cliques [14]. The rule originally only works for unweighted instances. We generalize it to a pair of vertices in the weighted setting as follows.

We can merge u and v with $s(uv) \geq 0$ if their edit cost to every other vertex differs by the same constant positive factor, i.e., there exists a $c > 0$ such that $s(uw) = c \cdot s(vw)$ for every other vertex w. The correctness proof is analogous to the unweighted case and goes roughly as follows. If v being in a certain cluster produces cost X, then u being in this cluster produces cost cX. Thus, the cheapest cluster for v is also the cheapest cluster for u, though there could be multiple equally cheap clusters. In the latter case it is nonetheless still not worse to put u and v together as $s(uv) \geq 0$. We note that applying this rule repeatedly to an initially unweighted instance merges all critical cliques.

Twin Complex [9, Rule 5]. Let u, v be two nodes that are connected with an edge. The rule considers, for all possible ways to separate them into different cliques, the worst case cost of moving one into the clique of the other. If deleting the edge uv is at least as expensive as this worst case, then there is an optimal solution with u, v in the same clique and the edge can be contracted. The rule is checked with a dynamic programming (DP) approach [9].

Unfortunately, the DP degenerates when dealing with forbidden edges, i.e., edges with cost $-\infty$. In the following, we discuss why this problem exists and what we did to fix it. If u and v have a similar neighborhood, then there is no worst case where both, moving u into v's clique or vice versa, are expensive. Intuitively, the rule works because one of the two options is always cheap. Now consider a vertex w with non-edges to u and v. If another reduction finds the edge uw to be forbidden, i.e., $s(uw) = -\infty$, then two things happen to the DP. First, the solutions that put u and w in the same clique can be ignored, which is beneficial as it makes it more likely that v can be moved into the clique of u. Second, the worst case will put w and v together, which makes it impossible to move u into the cluster of v. Thus, due to the second implication, knowing that uw is forbidden can have a detrimental effect on the applicability of the reduction rule. In fact, the DP degenerates to the point that not even true twins (except for the forbidden edge to w) can be merged. To circumvent this problem, we remember the edit cost for edges that are marked forbidden throughout the whole algorithm. In the DP we then use the original weights (getting rid of the downside due to the second aspect) but still skip solutions that put forbidden node pairs in the same clique (still using the upside of knowing uw is forbidden for the first aspect).

Induced Cost Forbidden/Permanent (icf,icp) [9]. Let Δ denote the symmetric difference. The induced costs for setting a vertex pair to forbidden (icf) or permanent (icp) are

$$\text{icf}(uv) = \sum_{w \in N(u) \cap N(v)} \min\{s(uw), s(vw)\}$$
$$\text{icp}(uv) = \sum_{w \in N(u) \Delta N(v)} \min\{|s(uw)|, |s(vw)|\}.$$

If the induced cost of setting a vertex pair to forbidden (icf) exceeds the current budget, then the pair must be merged. If the induced cost of setting a vertex pair to permanent (icp) exceeds the current budget, then the pair must be forbidden.

Heavy Non-Edge [9, Rule 1]. If $s(uv) < 0$ and $|s(uv)| \geq \sum_{w \in N(u)} s(uw)$, i.e., inserting the edge uv is at least as expensive as isolating u by cutting all of its edges, one can set uv to forbidden, forcing u and v to be in different clusters.

Heavy Edge, Single End [9, Rule 2]. If $s(uv) \geq \sum_{w \in V \setminus \{u,v\}} |s(uw)|$, i.e., deleting uv is at least as expensive as editing all other pairs involving u, one can merge u and v.

Heavy Edge, Both Ends [9, Rule 3]. If $s(uv) \geq \sum_{w \in N(u) \setminus \{v\}} s(uw) + \sum_{w \in N(v) \setminus \{u\}} s(vw)$, i.e., deleting uv is at least as expensive as deleting all other edges adjacent to u and v, one can merge u and v, as it is always better to let u and v form their own cluster of size 2 than to separate them.

Distance Three Rule [2]. Two vertices with distance three or more cannot be in the same cluster in an optimal solution. Therefore, all vertex pairs with distance three or more are initially marked as forbidden. This does not apply to weighted instances.

Forced Choices, all Pairs [13]. If setting an edge to forbidden or permanent would raise the lower above the upper bound, then the opposite edit must be performed. In other words, we identify an edge where, if branched on it, one branch would be pruned immediately.

A naive implementation of this rule is too slow as it requires a quadratic number of lower bound (i.e. packing) computations [13]. However all these packings are similar. Given a packing lower bound for the instance, we locally modify the packing for each vertex pair to obtain the required bounds. Because a packing changes only locally, this can be done significantly faster than computing $\binom{n}{2}$ packing lower bounds from scratch.

Forced Choices, Single Merge. Updating the lower bounds as in the previous rule is usually worse than computing the bounds from scratch. Thus, we additionally identify a constant number of edits that are unlikely to be included in the optimal solution and compute lower bounds for them from scratch. Specifically, we choose five vertex pairs and test if editing them to non-edges would be too expensive so that the rule produces a merge when applicable. We do not test for the converse because merges are far more rewarding than finding a single forbidden edge. To choose the five pairs, a heuristic estimates in advance which pairs would produce the highest cost when set to non-edges. Criteria for this heuristic are the cost of the edit as well as the number of overlapping P_3s with the vertex pair before and after the edit.

Clique-Like Subgraph. Given the instance (V, s) and the subset $C \subset V$, we define the *C-subinstance* (V, s_C) by setting $s_C(u, v) = s(u, v)$ if $u \in C$ or $v \in C$ and $s_C(u, v) = 0$ otherwise, i.e., if $u, v \in V \setminus C$. With this, we prove the following theorem.

▶ **Theorem 2.** *Let (V, s) be an instance of Weighted Cluster Editing, and let $C \subset V$ be a set of vertices. If an optimal solution for the C-subinstance isolates C into its own cluster, then there is an optimal solution for (V, s) that does so as well.*

The rule can be checked by solving the C-subinstance. We note that the instance (V, s_C) is likely easier than (V, s) due to the following observation. When looking at the graph with vertex set V with an edge between u and v if $s(u, v) > 0$, then we expect the closed neighborhood $N[C]$ of C to be rather small. Moreover, for a vertex $u \in V \setminus N[C]$ and any other vertex $v \in V$, we have $s_C(u, v) = 0$. Thus, we know that there is an optimal solution of (V, s_C) that has u as singleton, which reduces the instance to only the vertices in $N[C]$.

Reduction Order

Reductions are checked sequentially in a certain order. If one reduction was applied, the process rechecks all reductions starting with the first one. Before the loop repeats, a new lower bound is computed to check if the current branch can be pruned. We chose the order of reductions by decreasing effectiveness. The forced choices reductions are the most effective

reductions and come first. Twin Complex is also very effective, but we do Twin Simple before that because it is faster and already catches some cases for Twin Complex. The remaining reductions run in $O(n^3)$ each with low constant factors so their order does not matter as much. The final order of reductions that are checked during the branching algorithm is:

1. Forced Choices, all Pairs (Star)
2. Forced Choices, all Pairs (P_3)
3. Twin Simple
4. Twin Complex
5. Induced Cost Forbidden/Permanent
6. Heavy Edge, Both Ends
7. Heavy Edge, Single End
8. Heavy Non-Edge

The initial instance is reduced differently. The Distance Three rule is applied once before the reduction loop since it is only applicable to unweighted instances. Then, the Clique-Like Subgraph reduction checks the clusters that are found by the heuristic solver. The optimal solution for the subinstances is computed with the exact solver itself. To keep the running time reasonable we skip subinstances with 50 vertices or more and run the solver with a timeout of 5 seconds. Finally, the other reductions are applied in a loop. During the loop, the order differs in three aspects from the order given in the list above. First, small connected components are brute-forced before the first item on the list. Second, Forced Choices Single Merge is added as a last reduction. Third, Force P_3 is applied before Force Star.

4 Experiments

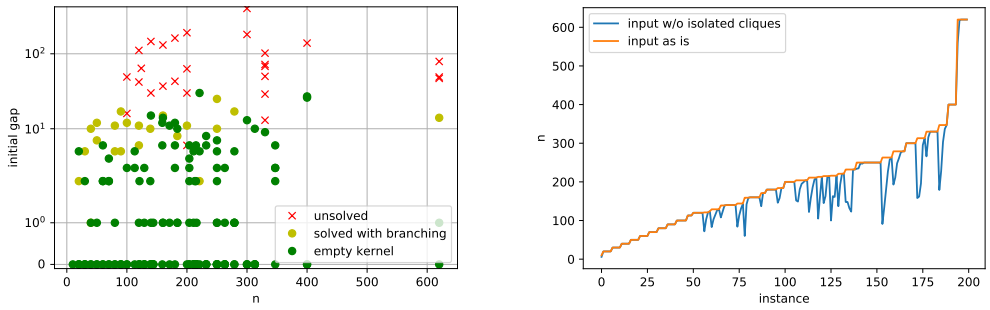
In Section 4.1, we discuss the performance of our solver, the efficiency and effectiveness of the reduction rules, and the quality of lower and upper bounds. In Section 4.2 we perform scaling experiments and determine how structural properties affect the solver performance on geometric inhomogeneous random graphs (GIRGs) [10], which are a generalization of hyperbolic random graphs [17]. As a generative network model, GIRGs can generate a series of similar instances that differ in a single property such as size, average degree, or clustering.

Setup. The experiments were run single threaded on a 4-Core Intel Xeon E5-1630v3 at 3.7GHz with 128GB DDR4 at 2133MHz. Each run has a soft timeout of one hour except for Figure 4a where it was 10 minutes per instance. Soft timeout means the current subroutine, is allowed to finish for the solver to terminate gracefully. To generate GIRGs, we use the efficient generator by Bläsius et al. [5]. The code for the experiments, raw data, execution logs, instances, as well as the plotting code can be found in a branch of our public repository[3].

4.1 PACE Instances

In the following, we use the public and hidden instances from the 2021 PACE challenge to evaluate our solver. They represent a well balanced selection of instances from bioinformatics and data mining as well as randomly generated ones. Moreover, they are publicly available

[3] `https://github.com/kittobi1992/cluster_editing/tree/experiments`



(a) The initial gap between lower and upper bound. **(b)** Vertices before/after removing isolated cliques.

Figure 1 For the 200 PACE instances, the gap between upper and initial lower bound (left) and the number of vertices per instance (right). In the left plot, the color indicates if an instance was solved by reductions only, needed branching, or remained unsolved in the given one-hour time limit.

at the PACE website[4]. We discuss the effectiveness and efficiency of individual reduction rules as well as their combination used in the solver. Then, we compare the quality of the greedy upper bound to the lower bounds obtained by the P_3 and star packing, respectively.

Solver Performance. In total, the algorithm solves 173 of the 200 instances from the PACE challenge with a timeout of one hour. Most instances finish significantly faster than that; 98 are solved in just one second and 160 finish in under a minute. Figure 1a shows for each instance if it was solved and whether reductions produce an empty kernel or branching was necessary. The axes correspond to the number of nodes and the initial gap between upper and lower bound. The gap is a good indicator of difficulty for our solver while the number of nodes seems unrelated to difficulty. All unsolved instances have a gap above 10. Surprisingly, 151 instances are solved with reductions alone. For a comparative evaluation of solver performance with other state-of-the-art algorithms, we refer to the official report of the 2021 PACE challenge [16]. On the hardware used in the actual challenge and a 30 min timeout, our algorithm solved 171 instances, while the second best submission solved 160.

Reduction Effectiveness. To evaluate the effectiveness of the reduction rules, we compute a kernel with each rule separately, i.e., apply the rule exhaustively with a soft timeout of one hour. This results in one kernel per combination of rule and instance. Before the kernel is computed we apply the Distance Three reduction rule which marks all vertex pairs in distance three or more as forbidden. Isolated cliques are removed from the input instance and once more from the final kernel. Figure 1b shows the size of the instances with and without the removal of isolated cliques. The results of the kernelization experiments can be seen in Figure 2. Each plot has a box for each reduction rule which represents the kernels made with this rule. There are two columns; the left one includes all instances while the right one only includes instances where the initial star bound does not match the upper bound. The instances with a gap between upper and lower bound represent more difficult instances for the solver thus making reductions more valuable on them. The rules *force p3* and *force star* refer to the Forced Choices, all Pairs reduction with the respective lower bound. The combination of the Induced Cost Forbidden and Permanent rules is labeled with *icx*. We

[4] `https://pacechallenge.org/`

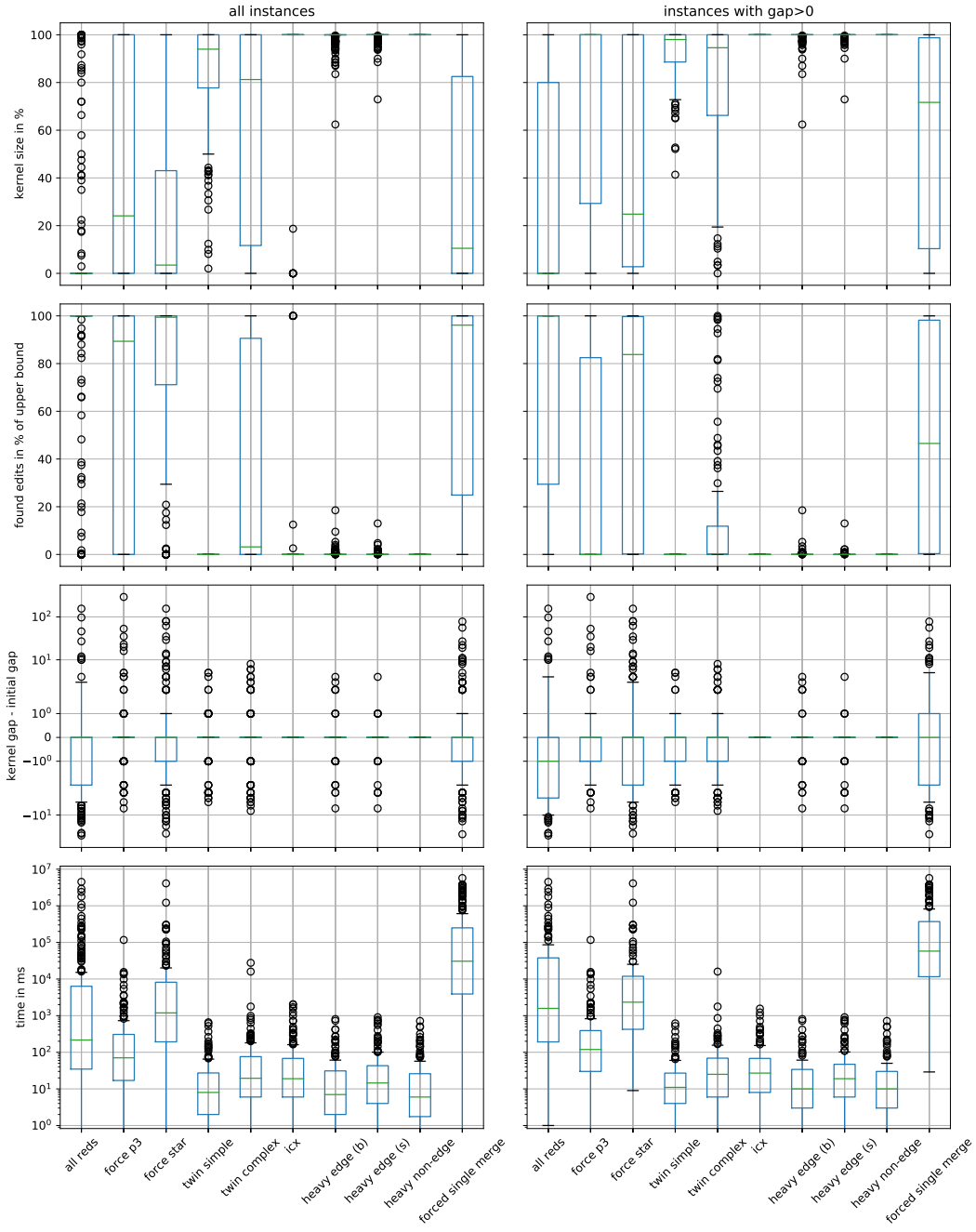


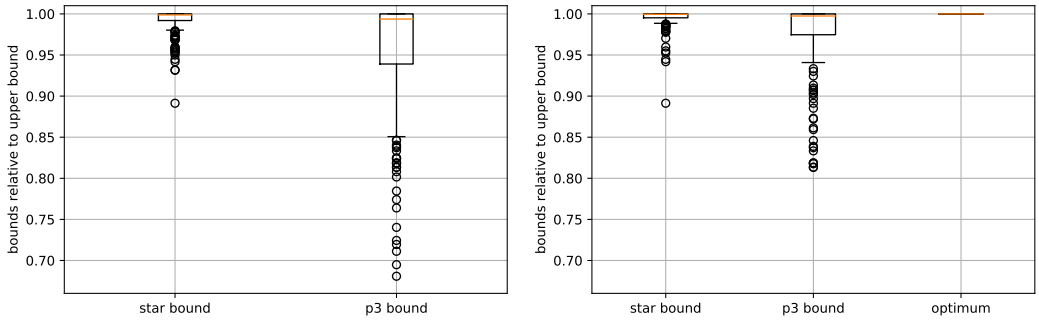
Figure 2 Kernels of PACE instances for each reduction rule. The rows show size, found edits, absolute gap change and time to compute the kernels. The left column includes all 200 instances while the right includes only the 121 instances with non-matching upper and lower bound. The label *all reds* refers to a combination of all other listed reduction rules.

also compute a kernel using all reduction rules (labeled as *all reds*) in the combination and order they are used by our solver to reduce the initial instance (see Section 3.3) excluding the brute-force of small components and the Clique-Like Subgraph reduction.

To evaluate the quality, we use three different measures. First, the number of vertices in the kernel, second, the number of edits that are already found, and third, whether the lower and upper bound get closer after computing the kernel. The number of vertices is a typical metric for kernel quality. For the second measure, the existence of an FPT algorithm for cluster editing indicates that the difficulty of the problem (the exponential part) is due to the size of the solution (the number of edits) rather than the size of the instance. In that sense, the percentage of edits that are already found during kernelization is a better indicator for progress towards a solution than instance size. The third measure, the gap between upper and lower bound, represents the difficulty of the kernel for any branch-and-bound solver using the lower bound that was used to compute the gap. With the change of the gap we estimate whether the kernel is easier or harder for our algorithm than the initial instance.

The first row shows the number of vertices in the produced kernels relative to the size of the initial instance without isolated cliques. The icx rules, both heavy edge rules and the heavy non-edge rule do not reduce the instance in the median. The heavy non-edge rule finds only forbidden edges. Therefore, the only way this rule could possibly reduce the number of vertices is by isolating a clique that is removed by our postprocessing. The simple twin and complex twin reductions are more effective with the complex twin producing smaller kernels. The non-zero gap instances prove to be harder for the twin reductions but the rules still find some application. The reductions based on forced choices produce the smallest kernels with an average size of 26% for force star, 44% for force P_3 and 36% for forced single merge. Best of all is the combination of all reduction rules that produces an empty kernel on more than 75% of instances and more than 50% of instances with a positive gap. On average, *all reds* reduces the instance to a size of 18%. While not explicitly shown, the instances with zero gap are interesting, too. The force star and the forced single merge reductions should produce an empty kernel in this case. While they indeed always apply initially, they do not always produce an empty kernel. This is because after the instance was reduced a few times it becomes weighted. Computing good packings for weighted instances becomes more difficult and might not be sufficient for the forced choices rules. In case of the forced single merge, the larger instances time out before the kernel is finished. Both these phenomenon happen rather rarely. On zero-gap instances the forced star produces a kernel size of 1.26% on average and 5.08% for forced single merge.

The second row shows the number of found edits that *must* be included in an optimal solution, i.e., the value that k is lowered by during kernelization. This value is normalized relative to the upper bound and represents another kind of progress towards solving the instance. The higher this value, the fewer choices remain to be fixed to solve the instance. Note that the values should be inverted when comparing with kernel size because for this plot 100% means solved while for instance size 0% means solved. Most reductions indicate similar results as for the kernel size. A notable exception is the simple twin rule. Since this rule only merges vertices with identical neighborhood, it never produces any cost but just reduces the instance. Interestingly, the progress made due to found edits is slightly better than the kernel size for the forced choices rules. For example, the force p3 kernel finds ca. 90% of edits in the median while the median kernel shrinks the instance by less than 80%. Since cluster editing is FPT in the number of edits, this is contrary to the expectation that the number of edits instead of the size of the instance should be responsible for the instance difficulty.

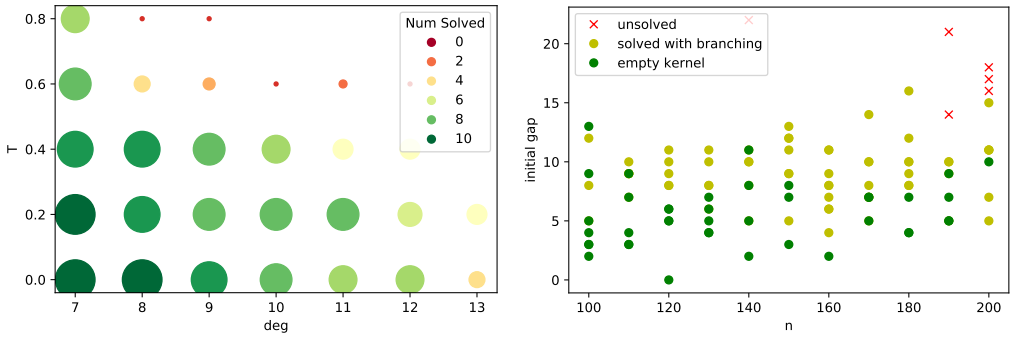


(a) Lower bounds via P_3, star packing (all instances). **(b)** Lower bounds and optimum for solved instances.

Figure 3 Packing lower bounds via P_3 and star packings on all instances (left) and on solved instances (right). The right plot additionally contains a column for the optimum. All values are relative to the respective upper bound for the instance. Note that the y-axis ranges from 0.7 to 1.0.

The third row shows the absolute change in gap after the kernel was computed, i.e., how much the difference between upper and lower bound has changed due to the kernelization. The y-axis uses symmetric log-scaling. A positive value means the gap grew and a negative value means the gap shrank. It might seem unintuitive that the gap can grow as previous lower bounds (before kernelization) still hold for the kernel. To explain this, consider two types of progress. The number of performed edits is *hard* progress, the lower bound represents *soft* progress. Once the total progress reaches the upper bound, the instance is solved. Hard progress is final but soft progress is temporary in the sense that, when actually solving the remaining instance, each reduction or branching step produces a new instance for which it might be more difficult to find good lower bounds. In general, there is no clear tendency for any reduction rule to only grow or only shrink the gap. The variance is very high to both sides. Thus, the inaccessibility of the kernel for soft progress sometimes outweighs the hard progress. In other cases it is the other way round or soft progress is even easier to achieve on the kernel. This is, e.g., the case for the simple twin rule, which makes no hard progress at all but has outliers to both sides. The median gap change is zero when looking at all instances. This is not surprising since 79 of the 200 instances already start with a gap of zero. For instances with a non-zero initial gap (the right column), the combination of all reductions actually reduces the gap by one for the median instance.

Reduction Efficiency. To evaluate the kernels by performance, we measure the time it takes to compute them. The last row of Figure 2 shows the results. The y-axis is logarithmic. Note that the highest outliers are approximately at $3.6 \cdot 10^6$ms which equals the soft timeout of one hour. All but the forced choices rules have a comparable run time with less than 100ms for more than 75% of the instances. Of these rules, just the twin complex and the icx rule have outliers over 1s and are in general the slowest of the non-forced choices rules. Note that the icx rule can be exhaustively applied in time $O(n^3)$ which is the same time one execution of the rule takes [8]. We instead apply the rule repeatedly since its run time is dominated by the forced choices rules. In the median, force p3 is slightly below 100ms, force star takes just above 1s, and forced single merge approximately one minute. All reductions combined are faster than force star but slower than force p3. Since all reductions produce by far the best kernels, this speaks for the order in which they are applied.



(a) Number of solved instances by T and degree. **(b)** The initial gap between lower and upper bound.

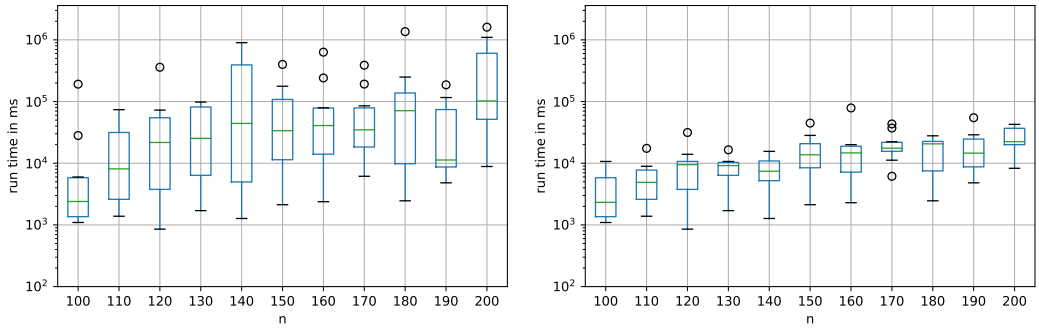
Figure 4 Solved instances by degree and T (left) and the initial gap on growing GIRGs (right). The colors in the right plot indicate if an instance was solved by reductions only, needed branching, or remained unsolved in the given time limit. The left plot maps color and size to solved instances.

Bound Quality. Figure 3 compares the P_3 bound, the star bound, and the optimum solution. The values are given relative to the upper bound computed for the instance. Figure 3a aggregates this over all instances while Figure 3b contains only solved instances and additionally shows the optimum solution. Note that the y-axis begins at 0.7 which means that even the worst outlier is already fairly good. The first observation is that the optimum is at 1.0 relative to the upper bound with no variance between instances. In fact, the upper bound from our heuristic solver matches the optimum on all 173 instance we can solve. Therefore, the lower bounds can be considered relative to the optimum; at least for the right plot. In total, the star bound is significantly better than the P_3 bound with all but one instances having a bound at more than 90% of the upper bound. The orange line for the median is only slightly lower for P_3 compared to the star bound but in this context this is huge. The number of edits in an optimal solution is approximately 1800 on average and goes as high as 27000. In contrast to this, we did not solve any instance with an initial gap of more than 30 (see Figure 1a), which is less than 2% of the 1800 edits needed on average. The average gap over all instances is 123 for P_3 and 15 for the star bound. Of course the average is heavily biased by the huge number of edits for the larger instances. Nevertheless, the 75th percentile ordered by absolute gap represents a gap of 43 for P_3 and 9 for the star bound, which makes the difference between solvable and unsolvable in this context.

4.2 Scaling Experiments

We use geometric inhomogeneous random graphs [10] to benchmark the solver for a growing number of vertices, temperature, and average degree. Unless noted otherwise, the number of vertices is 150, the average degree is ten, the power-law exponent describing the degree distribution is 2.9, the temperature parameter, which controls the degree of clustering, is zero (meaning high clustering) and the dimension of the ground space torus is two.

Temperature and Average Degree. Figure 4a shows the effect of clustering and average degree on solver performance. For each combination of temperature and average degree, the plot shows how many of ten instances were solved in less than ten minutes. There is a clear threshold behavior that instances with both, high temperature and high average degree, are rarely solved. High average degree (13) and low temperature (0.0) is manageable with four



(a) Run time in milliseconds on solved instances. **(b)** Run time in milliseconds for the initial kernel.

Figure 5 The run time of the solver in total (left) and to compute the initial kernel (right).

of ten instances solved; high temperature (0.8) and low average degree (7) even more so with seven of ten instances solved. In contrast, the algorithm solved only 4 of the 80 instances with temperature at least 0.6 and average degree at least 10.

Graph Size. Figure 4b gives an overview of which instances could be solved with or without branching. The axes are the size of the graph and the initial gap between lower and upper bound. As expected, the instances that could not be solved have a higher gap. Compared to the PACE instances, fewer can be solved without branching and only one has matching initial upper and lower bounds. Nevertheless, 41 of these 110 instances are solved by reductions alone. An instance with only 140 vertices was not solved and has a substantially higher gap than the others of the same size. GIRGs with the same configuration can vary greatly in difficulty for our solver. Two instances with 190 nodes and three with 200 nodes are unsolved.

Figure 5a and 5b show the total run time of the solver and the time spent with initial reductions, respectively. Although the run time of the solver differs up to three orders of magnitude between instances of the same size, the median time to solve an instance grows from 100 to 140 vertices. After that, the growth becomes less pronounced such that the variance makes it hard to estimate a clear trend. Also the last two columns are biased because of the instances that timed out. We explain the high variance by the comparatively small range of n that can reliably be solved and the low number of samples. Thus, in the range from 150 to 200 vertices the random sampling of position and degree distribution affects the difficulty of the GIRG for our solver more than the size of the graph. Nevertheless, the time to compute the initial kernel grows steadily with increasing number of nodes (see Figure 5b).

5 Conclusion

We present an exact branch-and-bound algorithm for the cluster editing problem. Moreover, we propose new reduction rules as well as formalize an improved technique to obtain lower bounds via subgraph packings, which contributes significantly to the success of the solver. We evaluate the lower bounds as well as various reductions rules on the instances of the 2021 PACE challenge. The lower bounds match the optimum on 79 of the 173 instances we were able to solve. For the reduction rules, by far the most effective ones are the rules that depend on lower bounds to identify forced choices, i.e., edge pairs that must or must not be edited in any optimal solution. They produce kernels with a small number of vertices and reduce k (the number of allowed edits) to an even greater extent. Combining all reductions

used by the solver produces an empty kernel on more than 75% of all instances. We also investigate the effect of size, clustering, and density on our algorithm in a scale-free network model. While the size of the graph has a small effect on performance, the combination of high density and low clustering produces remarkably hard instances.

References

1 Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. Fast local search for the maximum independent set problem. *Journal of Heuristics*, 18(4):525–547, 2012. doi:10.1007/s10732-012-9196-4.

2 Lucas Bastos, Luiz Satoru Ochi, Fábio Protti, Anand Subramanian, Ivan César Martins, and Rian Gabriel S Pinheiro. Efficient algorithms for cluster editing. *Journal of Combinatorial Optimization*, 31(1):347–371, 2016. doi:10.1007/s10878-014-9756-7.

3 Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner, Christopher Weyand, and Marcus Wilhelm. PACE solver description: KaPoCE: A heuristic cluster editing algorithm. In *16th International Symposium on Parameterized and Exact Computation (IPEC 2021)*, pages 31:1–31:4, 2021. doi:10.4230/LIPIcs.IPEC.2021.31.

4 Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner, Christopher Weyand, and Marcus Wilhelm. PACE solver description: The KaPoCE exact cluster editing algorithm. In *16th International Symposium on Parameterized and Exact Computation (IPEC 2021)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 27:1–27:3, 2021. doi:10.4230/LIPIcs.IPEC.2021.27.

5 Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. Efficiently generating geometric inhomogeneous and hyperbolic random graphs. In *27th Annual European Symposium on Algorithms (ESA 2019)*, pages 21:1–21:14, 2019. doi:10.4230/LIPIcs.ESA.2019.21.

6 Thomas Bläsius, Tobias Friedrich, David Stangl, and Christopher Weyand. An efficient branch-and-bound solver for hitting set. In *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 209–220, 2022. doi:10.1137/1.9781611977042.17.

7 Sebastian Böcker. A golden ratio parameterized algorithm for cluster editing. *Journal of Discrete Algorithms*, 16:79–89, 2012. doi:10.1016/j.jda.2012.04.005.

8 Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truß. Going weighted: Parameterized algorithms for cluster editing. *Theoretical Computer Science*, 410(52):5467–5480, 2009. doi:10.1016/j.tcs.2009.05.006.

9 Sebastian Böcker, Sebastian Briesemeister, and Gunnar W Klau. Exact algorithms for cluster editing: Evaluation and experiments. *Algorithmica*, 60(2):316–334, 2011. doi:10.1007/s00453-009-9339-7.

10 Karl Bringmann, Ralph Keusch, and Johannes Lengler. Geometric inhomogeneous random graphs. *Theoretical Computer Science*, 760:35–54, 2019. doi:10.1016/j.tcs.2018.08.014.

11 Yixin Cao and Jianer Chen. Cluster editing: Kernelization based on edge cuts. *Algorithmica*, 64(1):152–169, 2012. doi:10.1007/s00453-011-9595-1.

12 Jianer Chen and Jie Meng. A 2k kernel for the cluster editing problem. *Journal of Computer and System Sciences*, 78(1):211–220, 2012. doi:10.1016/j.jcss.2011.04.001.

13 Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. Engineering Exact Quasi-Threshold Editing. In *18th International Symposium on Experimental Algorithms (SEA 2020)*, volume 160, pages 10:1–10:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.SEA.2020.10.

14 Jiong Guo. A more effective linear kernelization for cluster editing. *Theoretical Computer Science*, 410(8-10):718–726, 2009. doi:10.1016/j.tcs.2008.10.021.

15 Sepp Hartung and Holger H. Hoos. Programming by optimisation meets parameterised algorithmics: A case study for cluster editing. In *Learning and Intelligent Optimization*, pages 43–58. Springer, 2015. doi:10.1007/978-3-319-19084-6_5.

16 Leon Kellerhals, Tomohiro Koana, André Nichterlein, and Philipp Zschoche. The PACE 2021 Parameterized Algorithms and Computational Experiments challenge: Cluster editing. In *16th International Symposium on Parameterized and Exact Computation (IPEC 2021)*, pages 26:1–26:18, 2021. doi:10.4230/LIPIcs.IPEC.2021.26.

17 Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3), 2010. doi:10.1103/physreve.82.036106.

18 Mirko Křivánek and Jaroslav Morávek. NP-hard problems in hierarchical-tree clustering. *Acta informatica*, 23(3):311–323, 1986. doi:10.1007/BF00289116.

A Missing Proofs

▶ **Lemma 3.** *Let a, b, c be three cost functions. If they are pairwise non-conflicting, then $(a + b)$ and c are non-conflicting.*

Proof. Let uv be a vertex pair with $c(uv) > 0$. If $c(uv) > 0$, then $a(uv) \geq 0$ and $b(uv) \geq 0$ since both are non-conflicting with c. Thus $(a+b)(uv) = a(uv)+b(uv) \geq 0$ which means that $(a+b)$ is non-conflicting with c on this vertex pair. The case for $c(uv) < 0$ works analogously. The case where $c(uv) = 0$ cannot lead to a conflict for c with any other function. ◀

▶ **Lemma 4.** *Let a, b be two non-conflicting cost functions. If for all vertex pairs uv, $|a(uv)| \leq |b(uv)|$, then $\mathrm{opt}(V, a) \leq \mathrm{opt}(V, b)$.*

Proof. Let K be a solution for (V, a). Lemma 4 follows from the fact that all vertex pairs that are edited in $\mathrm{cost}(K, a)$ are present in $\mathrm{cost}(K, b)$ with greater or equal absolute value. ◀

▶ **Lemma 5.** *For non-conflicting cost functions a, b, $\mathrm{opt}(V, a) + \mathrm{opt}(V, b) \leq \mathrm{opt}(V, a + b)$.*

Proof. Let K be an optimal solution of $(V, a + b)$. Since a, b and $a + b$ are all non-conflicting (due to Lemma 3), there is never a non-edge in one instance that is an edge in the other. Thus we get,

$$\begin{aligned}
\mathrm{opt}(V,a) + \mathrm{opt}(V,b) &\leq \mathrm{cost}(K,a) + \mathrm{cost}(K,b) \\
&= \sum_{\substack{a(uv)<0 \\ u \equiv_K v}} |a(uv)| + \sum_{\substack{a(uv)>0 \\ u \not\equiv_K v}} |a(uv)| + \sum_{\substack{b(uv)<0 \\ u \equiv_K v}} |b(uv)| + \sum_{\substack{b(uv)>0 \\ u \not\equiv_K v}} |b(uv)| \\
&= \sum_{\substack{(a+b)(uv)<0 \\ u \equiv_K v}} |(a+b)(uv)| + \sum_{\substack{(a+b)(uv)>0 \\ u \not\equiv_K v}} |(a+b)(uv)| \\
&= \mathrm{cost}(K, a+b) = \mathrm{opt}(V, a+b).
\end{aligned}$$

◀

▶ **Theorem 1.** *For packing P of the instance (V, s), $\sum_{p \in P} opt(V, p) \leq opt(V, s)$.*

Proof. Let $c : V \times V \mapsto \mathbb{Z}$ be the element-wise addition of all functions in P which is non-conflicting with s by Lemma 3. Moreover, Lemma 5 implies that $\sum_{p \in P} \mathrm{opt}(V, p) \leq \mathrm{opt}(V, c)$. Since P is a packing for (V, s) the third property of packings states that for all vertex pairs uv: $|c(uv)| \leq |s(uv)|$. Therefore, Lemma 4 results in $\mathrm{opt}(V, c) \leq \mathrm{opt}(V, s)$. ◀

▶ **Theorem 2.** *Let (V, s) be an instance of Weighted Cluster Editing, and let $C \subset V$ be a set of vertices. If an optimal solution for the C-subinstance isolates C into its own cluster, then there is an optimal solution for (V, s) that does so as well.*

Proof. Let $\mathcal{P}$ be any solution of (V, s). We construct a new partition $\mathcal{P}^\star$ that isolates C such that $\text{cost}(\mathcal{P}^\star, s) \leq \text{cost}(\mathcal{P}, s)$. For every cluster $A \in \mathcal{P}$, the partition $\mathcal{P}^\star$ contains $A \setminus C$. Additionally $\mathcal{P}^\star$ contains C.

For a pair $u, v \in V$, we say that $\mathcal{P}$ *splits* u and v if u and v are in different clusters of $\mathcal{P}$; otherwise $\mathcal{P}$ *joins* u and v. Similarly, for $C \subseteq V$, we say that $\mathcal{P}$ *splits* C if $\mathcal{P}$ splits at least one pair of vertices in C. Otherwise, if C is contained in a cluster of $\mathcal{P}$, then $\mathcal{P}$ *joins* C. We regularly need to sum over only negative or only positive cost. To simplify notation in these cases, let $s^+, s^- \colon V \times V \to \mathbb{N}$ be defined as $s^+(u, v) = \max(0, s(u, v))$ and $s^-(u, v) = \max(0, -s(u, v))$. The cost of the solution $\mathcal{P}$ is then defined as

$$\text{cost}(\mathcal{P}, s) = \sum_{\substack{u,v \in V \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) + \sum_{\substack{u,v \in V \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v).$$

For the subset $C \subset V$ and its complement $T = V \setminus C$ it will be useful to split the sum in $\text{cost}(\mathcal{P}, s)$ by vertex pairs within C, pairs between C and T, and pairs within T. We have

$$\begin{aligned}\text{cost}(\mathcal{P}, s) = &\sum_{\substack{u,v \in C \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \; + \sum_{\substack{u,v \in C \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v) \; + \\ &\sum_{\substack{(u,v) \in C \times T \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \; + \sum_{\substack{(u,v) \in C \times T \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v) \; + \\ &\sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \; + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v). \end{aligned} \tag{1}$$

Now we can bound the cost of $\mathcal{P}^\star$ in the original instance. Consider the six sums of $\text{cost}(\mathcal{P}^\star, s)$ as in Equation (1). The first sum evaluates to 0 as $\mathcal{P}^\star$ does not split pairs in C. Similarly, the fourth sum evaluates to 0, as $\mathcal{P}^\star$ does not join vertices from C with vertices from T. For the second and third sum, we can drop the condition that $\mathcal{P}$ joins and splits u, v, respectively, as $\mathcal{P}$ joins all pairs in C and splits all pairs between C and T. Finally, $\mathcal{P}^\star$ splits a vertex pair $u, v \in T$ if and only if $\mathcal{P}$ does, i.e., we can exchange $\mathcal{P}^\star$ with $\mathcal{P}$ in the fifth and sixth sum. Thus, writing the remaining sums in order $5, 6, 2, 3$, we obtain

$$\begin{aligned}\text{cost}(\mathcal{P}^\star, s) &= \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \; + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v) \; + \sum_{u,v \in C} s^-(u, v) \; + \sum_{(u,v) \in C \times T} s^+(u, v) \\ &= \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \; + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v) \; + \; \text{opt}(V, s_C) \\ &\leq \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \; + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v) \; + \; \text{cost}(\mathcal{P}, s_C) \\ &= \text{cost}(\mathcal{P}, s).\end{aligned}$$

The first equality follows from the premise of the theorem that there is an optimal solution for the C-subinstance that isolates C. Any solution for the subinstance that isolates C has to pay for all non-edges in C as well as all *edges* from C to $V \setminus C$. Moreover the solution that keeps all other vertices as singletons incurs no additional cost beyond this. Therefore the premise can alternatively be stated as

$$\sum_{u,v \in C} s^-(u, v) \; + \sum_{(u,v) \in C \times T} s^+(u, v) = \text{opt}(V, s_C). \tag{2}$$

For the inequality, we have $\text{opt}(V, s_C) \leq \text{cost}(\mathcal{P}, s_C)$ because $\mathcal{P}$ is also a solution for (V, s_C). Regarding the last equality, note that $\text{cost}(\mathcal{P}, s_C)$ coincides with $\text{cost}(\mathcal{P}, s)$ except that the last two terms of Equation (1) evaluate to 0 for $\text{cost}_S(\mathcal{P}, s_C)$, i.e.,

$$\text{cost}(\mathcal{P}, s) = \text{cost}(\mathcal{P}, s_C) \; + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} \text{cost}^+(u, v) \; + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} \text{cost}^-(u, v).$$

In conclusion, we obtain $\text{cost}(\mathcal{P}^\star) \leq \text{cost}(\mathcal{P})$, which proves the claim. ◀

B Unused Reductions

There are other reduction rules in the literature, which did not make it into our solver for different reasons, which we briefly discuss in the following.

The Clique-Like Subgraph rule is similar to Rule 4 by Böcker et al. [9], which is based on min-cuts. Since the min-cut rule can be computed more efficiently it could be useful in a solver, but we found it to be ineffective during preliminary testing.

We implemented the reduction rules by Cao and Chen [11] leading to a kernel of size $2k$, which is the smallest known kernel for cluster editing. However, our preliminary experiments showed that these reduction rules were dominated by other rules. Moreover, the rules explicitly exclude zero-edges, which we can get due to edge contractions, and it is not obvious how to adapt the rules to this setting.

Böcker et al. [9] suggest the improvement to the Induced Cost Forbidden/Permanent rules to add a lower bound for the graph without u and v to the induced costs to obtain an even stronger rule. The P_3-packing bound exactly captures this idea, even if it does not explicitly compute icf / icp, because all triangles formed with uv have a combined cost of icf / icp and can all be included in the packing because they share only the vertex pair uv. Moreover, the P_3-packing bound is strictly stronger in a weighted setting because it can additionally use the residual cost of edges uw and vw for all $w \in V \setminus \{u, v\}$ after each P_3 constituting the icf / icp is removed. Although the Forced Choices All Pairs rule with P_3 packings as bound dominates this improved version of the icp/icf rules, we keep the icp/icf rules in the solver as a failsafe to detect possible errors in the involved implementation of the forced choices rule.

Finally, there are the rules used in the unweighted $4k$ and $2k$ kernels [12, 14]. We have not implemented or adapted them to a weighted setting except for our generalization of the Simple Twin rule. The kernels are based on critical cliques, i.e., a clique containing nodes with identical closed neighborhood and the Simple Twin rule merges all critical cliques when applied repeatedly. Moreover, some other rules from these kernels are captured by our implemented rules. E.g., the Induced Cost Forbidden rule dominates rule 1 from the unweighted $2k$-kernel [12]. Nevertheless, the rules could be useful in future work.

C Reproducibility

The random components such as the generation of GIRGs or the local search to find a packing bound use the Mersenne Twister algorithm of the C++ standard template library. They are seeded as to produce deterministic results. In fact, each lower bound computation uses the same seed therefore producing the same output when given identical inputs. For each set of input parameters for the GIRG model we generate 10 instances with different seeds.

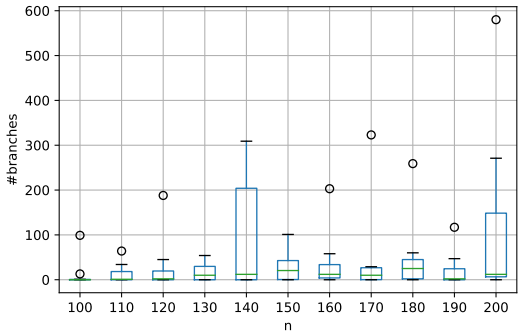
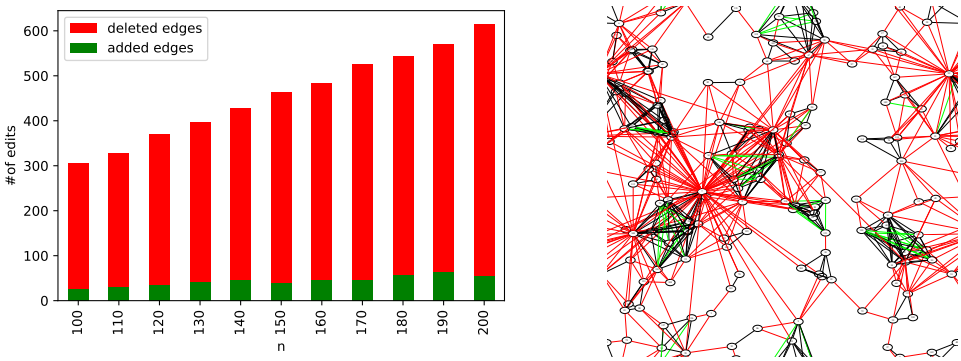


Figure 6 The number of branching decisions of the solver.



(a) Average number of edits in an optimal solution. **(b)** Visualization of an optimal solution for a GIRG.

Figure 7 The number of edits in an optimal solution (left) and a possible optimal solution (right). The left plot shows the average over ten instances. The right GIRG was generated with default parameters and edits are indicated by color. Thus, the green edges are not in the generated GIRG.

D Search Space on Growing GIRGs

Figure 6 shows the number of branching decisions while solving GIRGs of different size. The plot includes only the solved instances. The results confirm that reductions contribute the most to solver performance. The number of branches is six on average. More than half of all instances are solved with less than ten branches. There is no clear trend for growing number of vertices. All graph sizes have outliers that need far more branching decisions. The instance with the highest number of branches that was still solved has 580.

E Solution Structure on GIRGs

Figure 7b shows the edits of an optimal solution for a GIRG with the default parameters listed above. Note that the positions for the vertices are sampled in a unit torus where opposite borders are identified. Due to the scale-free degree distribution, some vertices have very high degree in the input instance. Most of those edges are deleted and the high-degree node is placed in the largest clique in close proximity. There are many small cliques in the resulting cluster graph and more edges are deleted than inserted. Figure 7a confirms this observation. The plot shows the number of deleted or added edges in an optimal solution over growing graph size. Each bar represents the average over all solved instances for this

size. Due to the average degree of ten, the number of edges in the input is between 500 and 1000. The number of total edits grows approximately linear in the size of the graph and deletes about half of all edges. In fact, between 44 and 64 percent of the edges are deleted which seems to be independent of graph size. The number of inserted edges is comparatively low but also grows with growing number of vertices.

An Experimental Study of Algorithms for Packing Arborescences

Loukas Georgiadis
Department of Computer Science & Engineering, University of Ioannina, Greece

Dionysios Kefallinos
Department of Computer Science & Engineering, University of Ioannina, Greece

Anna Mpanti
Department of Computer Science & Engineering, University of Ioannina, Greece

Stavros D. Nikolopoulos
Department of Computer Science & Engineering, University of Ioannina, Greece

Abstract

A classic result of Edmonds states that the maximum number of edge-disjoint arborescences of a directed graph G, rooted at a designated vertex s, equals the minimum cardinality $c_G(s)$ of an s-cut of G. This concept is related to the *edge connectivity* $\lambda(G)$ of a strongly connected directed graph G, defined as the minimum number of edges whose deletion leaves a graph that is not strongly connected. In this paper, we address the question of how efficiently we can compute a maximum packing of edge-disjoint arborescences in practice, compared to the time required to determine the edge connectivity of a graph. To that end, we explore the design space of efficient algorithms for packing arborescences of a directed graph in practice and conduct a thorough empirical study to highlight the merits and weaknesses of each technique. In particular, we present an efficient implementation of Gabow's arborescence packing algorithm and provide a simple but efficient heuristic that significantly improves its running time in practice.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Mathematics of computing → Graph algorithms

Keywords and phrases Arborescences, Edge Connectivity, Graph Algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.14

Supplementary Material *Software (Source Code and Sample Input Instances)*:
`https://github.com/sakiskef/PackingArborescencesAlgorithms`
archived at `swh:1:dir:de7880aa38dd66bdb3661030d905a224bd23c8b9`

Funding Research supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant", Project FANTA (eFficient Algorithms for NeTwork Analysis), number HFRI-FM17-431.

Acknowledgements We would like to thank the anonymous referees for several useful comments.

1 Introduction

Let $G = (V, E)$ be a directed graph (digraph), with m edges and n vertices. Digraph G is *strongly connected* if there is a directed path from each vertex to every other vertex. Throughout the paper we let s be a fixed but arbitrary start vertex of G. If G is strongly connected, then all vertices are reachable from s and reach s. The *edge connectivity* $\lambda(G)$ of G is the minimum number of edges whose deletion leaves a digraph that is not strongly connected. Computing the edge connectivity of a graph is a classical subject in graph theory, as it is an important notion in several application areas, such as in the reliability

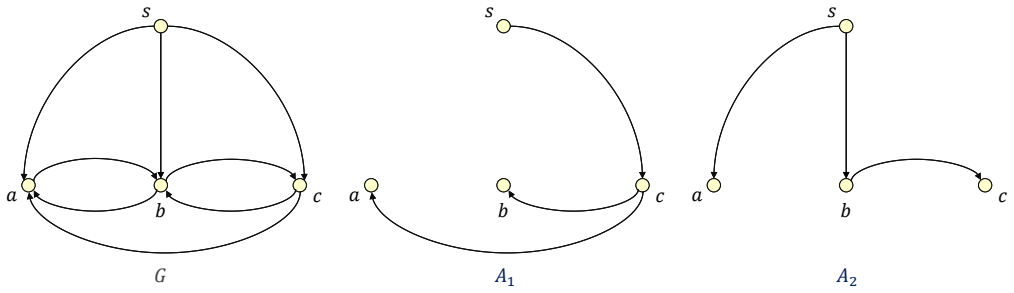


Figure 1 A directed graph G with start vertex s and minimum s-cut value $c_G(s) = 2$. Subgraphs A_1 and A_2 are two edge-disjoint s-arborescences of G.

of transportation and communication networks, and in production, scheduling, and power engineering [23]. The *reverse digraph* of G, denoted by $G^R = (V, E^R)$, is the digraph that results from G by reversing the direction of all edges. An *s-cut* is the set of edges directed from S to $V \setminus S$, where S is any vertex set that contains s such that $S \subset V$. We let $c_G(s)$ denote the minimum cardinality of an s-cut of G. Then, the edge connectivity of G is the minimum cardinality of an s-cut of G or G^R, i.e., $\lambda(G) = \min\{c_G(s), c_{G^R}(s)\}$. This observation also holds for undirected graphs, since the edge-connectivity of an undirected graph is equal to the edge-connectivity of the corresponding directed graph where each edge is oriented in both directions.

A *spanning tree* of an undirected connected graph G is a connected acyclic spanning subgraph of G. We extend this definition to directed graphs by ignoring the edge orientations. Let T be a *spanning tree* of a directed graph G rooted at s; T is an *s-arborescence* of G if s has in-degree zero and every other vertex has in-degree one. (Thus, any s-arborescence is a spanning tree of G but not vice versa.) The *arborescence packing problem* for vertex s is to construct the greatest possible number of edge-disjoint s-arborescences. See Figure 1. These concepts are useful in applications such as modeling broadcasting and evacuation [10].

Currently, the state of the art algorithm for computing the edge-connectivity λ of a digraph is the algorithm of Gabow [11] which runs in $O(\lambda m \log n^2/m)$ time. Gabow's algorithm is inspired by matroid intersection and is based on the idea of packing spanning trees. Moreover, in [11], Gabow shows how to extend his edge-connectivity algorithm so that it also computes a maximum packing of edge-disjoint s-arborescences of G in $O(k^2n^2)$ time, where $k = c_G(s)$. The edge connectivity of a simple undirected graph can be computed in $\tilde{O}(m)$ time[1], randomized [14, 17] or deterministic [15, 18]. In particular, the deterministic algorithm of Kawarabayashi and Thorup [18], as well as its improvement by Henzinger et al. [15], apply Gabow's algorithm on a contracted graph, for which the latter runs in $\tilde{O}(m)$ time.

In this paper we consider the arborescence packing problem from a practical perspective. Our starting point is the following fundamental theorem of Edmonds:

▶ **Theorem 1** (Edmonds [7]). *The maximum number of edge-disjoint s-arborescences of G equals the minimum cardinality of an s-cut.*

Edmonds gave an algorithmic proof, but the algorithm is complicated and seems to require exponential time in the worst-case [11]. Later, Lovasz [21] gave an elegant proof of Edmonds' theorem. Tarjan [25] presented an $O(k^2m^2)$-time algorithm to compute a maximum packing

[1] The notation $\tilde{O}(\cdot)$ hides poly-logarithmic factors.

of edge-disjoint s-arborescences, where $k = c_G(s)$. Schiloach [24], presented later an $O(k^2mn)$-time algorithm. The same bound was achieved by Tong and Lowler [26], who also claimed that Schiloach's algorithm is flawed. See also [9, 16] for recent interesting generalizations of Edmonds' theorem.

Currently, the best bound for the arborescence packing problem is $O(mk \log n + nk^4 \log^2 n)$, which was achieved by Bhalgat et al. [2] using the concept of edge splitting [1, 12]. Let G be a digraph with start vertex s. Define the *s-v edge-connectivity* $c_G(s, v)$, for any vertex $v \neq s$, as the cardinality of the minimum s-v cut. We say that a vertex v is *eligible* if *indegree*$(v) \geq$ *outdegree*(v). For such a vertex v, we can assume that *indegree*(v) = *outdegree*(v) since we can add multiple edges from v to s without affecting the s-w edge-connectivity $c_G(s, w)$ of any $w \neq s$. Splitting off two edges (x, y) and (y, z) means deleting these two edges, and adding a new edge (x, z). This operation can be done so that the s-v edge-connectivity $c_G(s, v)$ is preserved for any $v \neq y$. Splitting off of an eligible vertex v means to split off pairs of edges incident on v so that each pair consists of an edge entering v and an edge leaving v, without affecting the connectivity of the remaining graph until v has no outgoing edge. Now, v can be removed from the graph without affecting the connectivity of the remaining graph [2]. Bhalgat et al. describe a procedure that removes any specified set S of eligible vertices while maintaining the s-v edge-connectivity of all $v \notin S$. Their algorithm extends the edge-connectivity algorithm of Gabow so that it can compute a splitting of all vertices in S. Then, it recursively computes a maximum arborescence packing of the resulting graph, which can then be used to recover a maximum arborescence packing of the original graph by putting back the vertices in S.

Here we explore the design space of efficient algorithms that compute a maximum packing of edge-disjoint arborescences. In particular, we present an efficient implementation of Gabow's arborescence packing algorithm, and provide a simple but efficient heuristic that significantly improves its running time in practice. Then, we conduct a thorough empirical study to highlight the merits and weaknesses of each technique. To the best of our knowledge, we present the first efficient implementations of algorithms for packing arborescences. Hence, we also complement the work of Georgiadis et al. [13] that studies the practical efficiency of algorithms for computing the edge-connectivity.

2 Preliminaries

Let G be a directed graph, which may have multiple edges, with a distinguished start vertex s. We denote the vertex and edge sets of G by $V(G)$ and $E(G)$, respectively. A vertex v is *reachable* in G if there is a path from s to v; v is *unreachable* if no such path exists. An *s-cut* is the set of edges directed from S to $V(G) \setminus S$, where S is any vertex set that contains s such that $S \subset V(G)$. We let $c_G(s)$ denote the minimum cardinality of an s-cut of G. Then, $c_G(s) > 0$ if and only if all vertices are reachable.

Let $S \subseteq V(G)$. The out-degree (resp., in-degree) of S, denoted by $\delta^+_G(S)$ (resp., $\delta^-_G(S)$), is the number of edges directed from S to $V(G) \setminus S$ (resp., from $V(G) \setminus S$ to S). For a vertex $v \in V(G)$, $\delta^+_G(v)$ (resp., $\delta^-_G(v)$) denotes its out-degree (resp., in-degree) in G. We let $\delta_G = \min_{v \in V(G)}\{\delta^+_G(v), \delta^-_G(v)\}$ denote the minimum degree of the graph. We let $E^+_G(v) = \{(v, w) \in E(G)\}$ (resp., $E^-_G(v) = \{(u, v) \in E(G)\}$), i.e., the set of edges directed from v (resp., to v), and refer to $E_G(v) = E^+_G(v) \cup E^-_G(v)$ as the set of edges adjacent to v. We omit the subscript G if the graph is clear from the context.

Let T be a spanning tree of G and let e be an edge that is not contained in T. The *fundamental cycle of e in T*, denoted by $C(e, T)$, is the cycle that is formed by adding e into T.

An *s-arborescence* A is a directed graph such that all vertices in $V(A)$ are reachable from s, $\delta^-_A(s) = 0$ and $\delta^-_A(v) = 1$ for all $v \in V(G) - s$. I.e., there is exactly one directed path in A from s to any other vertex. We say that A is an *s-arborescence of* G if A is a spanning subgraph of G, i.e., $V(A) = V(G)$. If $V(A) \subset V(G)$, then we say that A is a *partial s-arborescence of* G.

A *k-intersection* of G is a collection T of k spanning forests $T_1, \ldots, T_k$ of G that contains at most k edges directed to each vertex $v \in V(G) - s$, and none to s, i.e., $\delta^-_T(s) = 0$ and $\delta^-_T(v) \leq k$ for $v \in V(G) - s$. A k-intersection $T = \{T_1, \ldots, T_k\}$ is *complete* if each T_j is a spanning tree, so that $\delta^-_T(v) = k$ for all $v \neq s$. Edmonds [6] also proved the following *Matroid Characterization* of s-cuts:

▶ **Theorem 2** (Edmonds [6]). *The edges of a directed graph can be partitioned into k s-arborescences if and only if they can be partitioned into k spanning trees and every vertex except s has in-degree k.*

Referring to this result as the Matroid Characterization of minimum-cut is justified by the fact that the spanning trees with the above property are formed by the intersection of two matroids.

For a graph G and a subgraph H of G, we let $G - H$ denote the subgraph of G with vertex set $V(G)$ and edge set $E(G) - E(H)$. Also, for an edge e, we let $G \cup \{e\}$ denote the graph after adding e to G, and let $G - e$ denote the graph after deleting e from G.

3 Algorithms

In this section we provide an overview of the algorithms that we consider in our experimental study. Let G be the input digraph with n vertices, m edges, and start vertex s. Throughout this section, we let $k = c_G(s)$. We assume that all vertices are reachable (from s) in G, so $k > 0$, since otherwise there is nothing to do.

Let A be a partial s-arborescence of G. We say that A is *good* if $c_{G-A}(s) \geq k - 1$. All the algorithms we consider try to enlarge a partial s-arborescence A of G by adding one edge at a time.

We note that we can combine any packing arborescences algorithm with Gabow's edge connectivity algorithm as follows. First, we run Gabow's edge connectivity algorithm on G, and compute a complete k-intersection T of G in $O(km \log n^2/m)$ time. Then, we keep in G only the edges of $E(T)$ and run the packing arborescences algorithm on the reduced graph. (This is valid by the Matroid Characterization of s-cuts.) If the packing algorithm runs in $O(f(m, n))$ time on the original graph, then the combined algorithm runs in $O(km \log n^2/m + f(nk, n))$ total time.

3.1 The algorithm of Tarjan

Tarjan [25] computes a maximum packing of arborescences $\mathcal{A} = \{A_1, \ldots, A_k\}$ by executing k iterations of the following procedure. During the j-th iteration (for $j = 1, 2, \ldots, k$), it computes an arborescence A_j of G such that A_j is pairwise edge-disjoint with the arborescences in $\mathcal{A}^{(j-1)} = \{A_1, \ldots, A_{j-1}\}$, and $G^{(j)} = G \setminus \mathcal{A}^{(j)}$ has $c_{G^{(j)}}(s) = k - j$. (I.e., A_j is good for $G^{(j-1)}$.) To compute A_j we work as follows. We initialize a vertex set $S = \{s\}$, the set of edges $E(A_j) = \emptyset$ and mark all edges of $G^{(j-1)}$ as usable. Then, we perform the following step until $S = V(G)$. We find a usable edge $e = (u, v)$ such that $u \in S$ and $v \in V(G) \setminus S$, mark e as unusable and compute $c_{G'}(S)$, where $G' = G^{(j-1)} \setminus (A_j \cup \{e\})$. If $c_{G'}(S) \geq k - j$ then we add v to S, and add e to A_j.

To test if $c_{G'}(S) \geq k - j$, it suffices to determine if we can send at least $k - j$ units of flow from S to v (where we assign unit edge capacities). This can be done in $O(km)$ time by executing at most $k - j$ iterations of the Ford-Fulkerson method [8]. Since we perform k iterations, and in each iteration we test at most m edges, the total running time is $O(k^2m^2)$. This bound is reduced to $O(km \log n^2/m + k^4n^2) = O(k^4n^2)$ if we first compute a complete k-intersection of G by Gabow's edge connectivity algorithm, and then run Tarjan's algorithm on the reduced graph that contains only the edges of $E(T)$.

Implementation details

In our experiments, we noticed that the order in which we examine the usable edges may affect the running time of the algorithm significantly. Hence, we considered two versions of the algorithm: In the first version, we maintain the vertices of S in a stack, and examine the usable edges $e = (u, v)$ starting from the most recently added vertices $u \in S$. In the second version, we maintain S in a FIFO queue, and examine the usable edges $e = (u, v)$ starting from the least recently added vertices $u \in S$. As in turns out, in our experiments the stack version performed significantly better on most instances. (See Appendix A.)

3.2 The algorithm of Tong and Lawler

The algorithm of Tong and Lawler [26] applies a divide and conquer approach. Similarly to Tarjan's algorithm, it grows a partial arborescence A of G by trying to add one edge at a time. Initially $V(A) = \{s\}$ and $E(A) = \emptyset$. A candidate edge $e = (u, v)$, such that $u \in V(A)$ and $v \in V(G) \setminus V(A)$, is selected according to the following rule: $u \neq s$, unless there is no other candidate edge. Then, we compute $c_{G'}(s)$, where $G' = G - (A \cup \{e\})$, and consider the following two cases. (a) If $c_{G'}(s) \geq k - 1$ then the examination of e is *successful*. In this case, we add v to $V(A)$, and add e to $E(A)$. If A is now an arborescence, that is $V(A) = V(G)$, then we set $G = G - A$ and recursively compute $k - 1$ edge-disjoint arborescences in G. (b) Otherwise, we have $c_{G'}(S) = k - 2$, and the examination of e is *unsuccessful*. In this case, the algorithm of Tong and Lawler applies divide and conquer as follows. Let $(S, V(G) \setminus S)$ be a minimum s-cut of G, where $s \in S$. It is easy to observe that $e \in E(S, V(G) \setminus S)$, hence $v \in V(G) \setminus S$, and moreover, that there must be an edge $e' \in E(A)$ such that $e' \in E(S, V(G) \setminus S)$. Next, we split G into two auxiliary graphs G_1 and G_2, with corresponding partial arborescences A_1 and A_2 as follows. To construct G_1, we contract the vertices of S into s and delete all edges directed to s. Then, A_1 consists of the edges of A that were not deleted and is a partial arborescence of G_1 (but it may not be a full arborescence yet). Similarly, we construct G_2 by contracting the vertices of $V(G) \setminus S$ into v and delete self-loops. To form a corresponding partial arborescence A_2 of A in G_2, we delete all the edges of A that are directed from S to $V(G) \setminus S$ except for the first edge (x, y) on a path from S to $V(G) \setminus S$ in A. That is, x is reachable from s through a path that contains only vertices in $S \cap V(A)$. So, A_2 consists of the edges of A that were not deleted and is a partial arborescence of G_2 (but it may not be a full arborescence yet). We recursively compute k edge-disjoint arborescences $A_1^1, A_2^1, \ldots, A_k^1$ of G_1 and $A_1^2, A_2^2, \ldots, A_k^2$ of G_2, where $E(A_1^1) \subseteq E(A_1)$ and $E(A_1^2) \subseteq E(A_2)$. Then, we combine these arborescences to form k edge-disjoint arborescences of G. This combination is easy to perform because each arborescence of G_1 is edge-disjoint from exactly $k - 1$ arborescences of G_2 and vice versa. Hence, to form the desired arborescences of G, we combine each pair of non-disjoint arborescences.

As in Tarjan's algorithm, we can test if an edge e can be included in the partial arborescence A in $O(km)$ time by executing k iterations of the Ford-Fulkerson method. Since at most kn edges are added in the packing $\mathcal{A}$, the total time spent for successful edge additions is $O(k^2mn)$. On the other hand, since we can split G at most n times, there are at most n unsuccessful edge examinations. Thus, the total time spent for unsuccessful edge examinations is $O(kmn)$, which results to a total running time of $O(k^2mn)$.

Tong and Lawler also observed that the running time of their algorithm can be improved to $O(kmn + k^3n^2)$ after some preprocessing. The preprocessing phase computes a flow of value k from s to each other vertex $t \neq s$. After we have computed an s-t flow, we delete the edges entering t with zero flow. By repeating this process for all vertices $t \neq s$, after $O(kmn)$ time we are left with a subgraph H of G with $O(kn)$ edges and $c_H(s) = k$. Thus, we can compute k edge-disjoint arborescences of H in $O(k^3n^2)$ time. If we use Gabow's edge connectivity algorithm to compute a complete k-intersection of G instead of H, then we obtain an $O(km \log n^2/m + k^3n^2) = O(k^3n^2)$ time bound.

Implementation details

As in our implementation of Tarjan's algorithm, we examine candidate edges (to be included in the partial arborescence A) using a stack or a FIFO queue. As with Tarjan's algorithm, the running time of the Tong-Lawler algorithm depends on the order in which we examine candidate edges. In our experiments the stack version of the Tong-Lawler algorithm outperformed the queue version, but not consistently. (See Appendix A.) When we split G, we create two new graph instances for G_1 and G_2, and assign new vertex ids so that they are in the ranges $[1, |V(G_1)|]$ and $[1, |V(G_2)|]$ respectively. To restore the original vertex ids, we maintain mappings $h_i : V(G_i) \mapsto V(G)$, $i = 1, 2$. Moreover, for each edge in G_1 and G_2 that has exactly one endpoint in a contracted part of the graph (i.e., for each edge (x, y) such that $x \in S$ and $y \in V(G) \setminus S$, or vice versa), we associate it with the corresponding original edge of G. This information suffices to combine the arborescences in G_1 and G_2 and form the arborescences of G. Thus, after a split, we no longer need to keep the initial graph in memory.

3.3 The algorithm of Gabow

Gabow [11] presented an $O(k^2n^2)$-time algorithm to compute a maximum arborescence packing. First, it computes a complete k-intersection T of the input digraph for s. We let G be the subgraph with edges $E(T)$. Then, it repeats the following procedure, until $k = 0$:

1. Compute a complete $(k-1)$-intersection T of G.
2. Find a good s-arborescence A of G, using the algorithm described below in Section 3.3.2.
3. Decrease k by one and repeat the procedure on $G - A$.

Similarly to the algorithms of Tarjan, and of Tong and Lawler, in Step 2 Gabow's algorithm tries to enlarge a partial arborescence by adding one edge at a time. Unlike the these other algorithms, however, Gabow's algorithm does not perform flow computations, but relies on the framework of his edge-connectivity algorithm. Hence, we first provide an overview of how Gabow computes the value $k = c_s(G)$ of a minimum s-cut, together with a complete k-intersection T of G, in $O(km \log n^2/m)$ time.

3.3.1 Computing a complete k-intersection T of G

Recall that a complete k-intersection T of G is a collection of k edge-disjoint spanning trees $T_1, \ldots, T_k$, such that each vertex $v \neq s$ has in-degree k and s has in-degree zero. Gabow's algorithm computes T in k iterations, where in the k'-th iteration ($k' = 1, \ldots, k$), it begins

with a complete $(k'-1)$-intersection and tries to enlarge it so that it becomes a complete k'-intersection. To that end, it executes a *round robin* algorithm that maintains a forest $T_{k'}$ and tries to locate "joining" edges that will make $T_{k'}$ a spanning tree of G, while satisfying the invariant that $\delta^-_T(s) = 0$ and $\delta^-_T(v) \leq k'$ for all $v \neq s$. In the following, we call a vertex v *deficient* if $\delta^-_T(v) < k'$.

In more detail, during the k'-th iteration $T_{k'}$ is a forest of rooted trees, referred to as *f-trees*, where each f-tree F_z is rooted at its unique deficient vertex z. For $z \neq s$, $\delta^-_T(z) = k' - 1$. An edge $e = (x, y)$ is *joining* if x and y are in different f-trees of $T_{k'}$. The round robin algorithm looks to enlarge $T_{k'}$ by one edge at a time, and simultaneously to increase the in-degree of a deficient vertex $z \neq s$ by one. To that end, it examines an edge e_1 in $E^-(z) \setminus E(T)$. If e_1 is joining then we are done. Otherwise, it adds e_1 in some T_i and looks for a joining edge in the fundamental cycle $C(e, T_i)$. To break the cycle, we can remove from T_i an edge $e_2 \in C(e, T_i)$. Then, we can add $e_2 = (u, v)$ to some other tree of T, or replace e_2 with a edge in $E^-(v) \setminus E(T)$. This pattern continues until a joining edge is found. The sequence of edges that leads to a joining edge is called an *augmenting path*. We define this notion formally below.

An ordered pair of edges e, f is called a *swap* if $f \in C(e, T_i)$ for some $T_i \in T$. To *execute the swap* is to replace f in T_i by e. A *partial augmenting path P from z* is a sequence of distinct edges $e_1, \ldots, e_l$, such that:

1. $e_1 \in E^-(z) \setminus E(T)$.
2. For each $i < l$ either
   a. $e_{i+1} \in C(e_i, T_j)$, where T_j contains e_{i+1} but not e_i, or
   b. $e_i, e_{i+1} \in E^-(v)$, for some vertex v, where T contains e_i but not e_{i+1}.
3. Executing all swaps of P (i.e., the pairs e_i, e_{i+1} of (2a)) gives a new collection of forests.

An *augmenting path P from z* is a partial augmenting path from z whose last edge e_l is joining for z. To *augment T along P* is to execute each swap of P and add e_l to $T_{k'}$. This increases the in-degree of z by one (so z is no longer deficient), while no other in-degree changes.

Each iteration is organized as a sequence of at most $\lceil \log n \rceil$ rounds. At the start of each round all f-trees are active except F_s. Then, we repeatedly choose an active f-tree F_z and search for an augmenting path from z. If no such path is found, then the algorithm terminates and reports an s-cut of cardinality $k'-1$. Otherwise, we have found an augmenting path P from z and we augment T along P. Thus, we enlarge T by one edge and F_z is joined to another f-tree F_w. The resulting f-tree of $T_{k'}$ is rooted at w (since z is no longer deficient), and becomes inactive for the rest of the current round. Gabow showed that with an appropriate implementation, each round runs in $O(m)$ time. Furthermore, he showed that by organizing the search for augmenting paths carefully, all augmentations can be executed at the end of a round.

3.3.2 Computing a good s-arborescence

We now give an overview of how Gabow computes a good s-arborescence A of G in Step 2 of his arborescence packing algorithm. The algorithm maintains the following subgraphs of G: a partial s-arborescence A of G, a working graph $H = G - A$, and a complete $(k-1)$-intersection T for s on G. It uses the following key concept. An *enlarging path* consists of an edge $e \in E^+(A)$, and if $e \in T$, an augmenting path P for the $(k-1)$-intersection $T - e$ on $H - e$. Gabow shows that if $V(A) \neq V(G)$, then there is always an enlarging path.

The algorithm marks the edges that are known to belong in any complete $(k-1)$-intersection contained in H. It also maintains a set X of vertices such that each $u \in X$ has all edges of $E^+(u) \cap E^+(V(A))$ marked. The algorithm is divided into "periods", where each period enlarges either A or X. Initially, A contains only the start vertex s, X is empty, and all edges are unmarked. Then, we repeatedly apply the following procedure that locates an enlarging path, until it halts:

Period Step. If A is an arborescence, that is $V(A) = V(G)$ then halt. Otherwise, choose a vertex $u \in V(A) \setminus X$ and execute the *Edge Step*.

Edge Step. If all edges in $E^+(u) \cap E^+(V(A))$ are marked then add u to X and continue with the next *Period Step*. Otherwise, choose an unmarked edge $e \in E^+(u) \cap E^+(V(A))$. If $e \notin E(T)$, then add e to A and continue with the next *Period Step*.

Search Step. At this point e belongs in T. Search for an augmenting path for the $(k-1)$-intersection $T - e$ in $H - e$. If the search is successful, then use the augmenting path to enlarge A and continue with the next *Period Step*. If the search is unsuccessful then mark e and go to *Edge Step*.

The correctness of this procedure is based on the following facts. Suppose $e \in E^+(u) \cap E^+(V(A))$ has no enlarging path, i.e., the *Search Step* was unsuccessful. Let L be the set of edges labelled in the search, and let e' be any edge in $E^+(u)$. Then, e belongs to any complete $(k-1)$-intersection contained in H, and no edge of L is in an enlarging path for e', with respect to the current A and T. This implies that for any $v \in X$, no edge of $E^+(v)$ has an enlarging path.

To make the search for enlarging paths fast, each unsuccessful search in a period contracts the edges in L that become labelled during an unsuccessful search for an edge $e \in E^+(u)$. The contraction is valid since, for each tree $T_i \in T$, $i = 1, 2 \ldots, k-1$, the edges in $L \cap (T_i - e)$ form a tree. Contracting $V(L)$ into a single vertex v results in a graph H' that has a complete $(k-1)$-intersection that contains all edges in $E^-_{H'}(v)$. Then, for any edges $e' \in E^+_{H'}(v)$, graphs H and H' have the same enlarging paths for e', and unsuccessful searches in $H - e'$ and $H - e$ label the same edges not in L.

To implement the above procedure efficiently, Gabow's algorithm performs the contractions implicitly. To that end, it maintains a partition of $V(G)$ into disjoint sets $S_1, \ldots, S_l$, such that each set S_j induces a tree in each $T_i \in T$. Each vertex is labelled with the name of the set that contains it and also, for each $i = 1, \ldots, k-1$, each set S_j is labelled with its root vertex in T_i. The *Search Step* for an edge $e = (u, w) \in T_i$ removes e from T and H, and searches for an augmenting path P from w. During this search, when a vertex v is reached, if $v \in S_j$ then the search continues from the root of S_j in T_i. If the search is successful, then we augment along P. Otherwise, when the search is unsuccessful, we add e back to H and T, and update the vertex partition $\{S_j\}$ by merging together all sets S_j that contain an end of an edge that was labelled during the search.

Gabow shows that this algorithm constructs an s-arborescence A in $O(kn^2)$ time; there are at most kn searches (at most one per edge), and at most $2n$ periods. The latter follows from the fact that a period enlarges A or X, and each set can be enlarged less than n times. Moreover, each period can be implemented to run in $O(kn)$ time. Since we compute k arborescences, the total running time is $O(k^2n^2)$.

Practical speedup

In order to speedup Gabow's algorithm in practice, we implemented the following simple heuristic. Let G be the current graph. We compute an s-arborescence A of G, e.g., by executing a depth-first search (DFS) from s, and test if A is good. To do that, it suffices to

Table 1 An overview of the algorithms considered in our experimental study. The bounds refer to a digraph G with minimum s-cut value $k = c_s(G)$, n vertices and m edges; $m \leq kn$ if G contains only the edges of a complete k-intersection.

Algorithm	Technique	Complexity	Ref.
Tarjan (Tar)	Test if a usable edge can be added via max-flow	$O(k^2m^2)$	[25]
	Run on a complete k-intersection	$O(k^4n^2)$	
Tong-Lawler (TL)	Graph splitting via min-cut	$O(k^2mn)$	[26]
	Run on a complete k-intersection	$O(k^3n^2)$	
Gabow (Gab)	Compute complete k'-intersections ($k' \leq k$) and enlarging paths	$O(k^2n^2)$	[11]

test if $c_{G-T}(s) = k - 1$. If this is the case, then we can keep A in the packing. Otherwise, we simply discard A, and compute a good s-arborescence of G using Gabow's algorithm. In either case, after we have computed a good s-arborescence A of G, we decrease k by one and repeat the procedure on $G - A$.

Despite its simplicity, the above modification provides significant speedups in practice, as the experimental results of Section 4 suggest. Moreover, we can immediately observe that the overall $O(k^2n^2)$ running time still holds for this variant of Gabow's algorithm.

4 Empirical Analysis

We implemented our algorithms in `C++`, using `g++ 7.5.0` with full optimization (flag -O4) to compile the code. The reported running times were measured on a GNU/Linux machine, with Ubuntu (18.04.6 LTS): a Dell Precision Tower 7820 server 64-bit NUMA machine with an Intel(R) Xeon(R) Gold 5220R processor and 192GB of RAM memory. The processor has 24.75MB of cache memory and 18 cores. In our experiments we did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the `high_resolution_clock` class of the standard library `chrono`, averaged over ten different runs.

Table 1 gives an overview of the algorithms we consider in our experimental study. We did not include the algorithm of Bhalgat et al. because some important details are omitted from the extended abstract of [2].[2]

We base our implementation of Gabow's arborescence packing algorithm on efficient implementations of Gabow's edge connectivity algorithm presented in [13]. Also, for the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), we report the running time of their stack-based implementations. We compare the stack-based against the queue-based implementations in Appendix A. For the experimental evaluation, we considered two types of graphs: (i) real-world directed graphs, augmented with additional edges in order to increase their edge-connectivity, and (ii) k-cores of undirected graphs.

Augmented graphs

In our first experiment, we consider how the running time of each algorithm is affected as the minimum s-cut value $k = c_s(G)$ increases. To that end, we augment some real-word graphs as follows. Let G be an input strongly connected digraph. For a given parameter

[2] We are unaware of a full version of [2].

β, we create an augmented instance G_β of G by executing the following procedure. We go through the vertices of G and, for each vertex v, we add $\beta - \delta^-(v)$ edges directed to v if $\delta^-(v) < \beta$, where each added edge originates from a randomly chosen vertex. Then, we make a second pass over the vertices and, for each vertex v, we add $\beta - \delta^+(v)$ edges directed away from v if $\delta^+(v) < \beta$, where each added edge is directed to a randomly chosen vertex. Notice that the resulting graph has minimum degree $\delta \geq \beta$.

Table 2 reports the characteristics of the augmented graphs G_β produced by the above procedure for $\beta \in \{2, 4, 8, 16\}$. Here, we also give the number of edges (m') in a complete k intersection T of G (with respect to the start vertex s). In Table 3 we report the corresponding running times of each algorithm. The execution of an algorithm was terminated if it exceeded one hour. We also report the running times of two versions of Gabow's algorithm that computes a complete k intersection T of G: the standard version (Gab-EC), and a version that uses DFS to do a fast initialization of the forest $T_{k'}$ at the beginning of the k'-th iteration (Gab-EC-DFS). Both implementations are taken from [13]. For the algorithms of Tarjan (Tar), and of Tong and Lawler (TL), we report both the running time when the input is the original graph G (above) and a complete k intersection of G (below). In the latter case, the algorithms receive a complete k intersection T of G as input, and we do not account for the time required to compute T.

Table 2 Characteristics of augmented graphs, resulting from some real-world graphs after inserting some edges; n is the number of vertices, m the number of edges; δ denotes the minimum vertex (in or out) degree, and $c_s(G)$ denotes the cardinality of the minimum s-cut; m' is the number of edges in a complete k intersection of G (for the start vertex s).

Graph	n	m	δ	$c_s(G)$	m'	type and source
enron-EC2	8271	151651	2	2	8270	email network [19]
enron-EC4	8271	162999	4	4	24810	
enron-EC8	8271	190618	8	8	57890	
enron-EC16	8271	253345	16	16	124050	
p2p-Gnutella25-EC2	5152	19765	2	2	10302	peer2peer network [19]
p2p-Gnutella25-EC4	5152	27565	4	4	20604	
p2p-Gnutella25-EC8	5152	50076	8	8	41208	
p2p-Gnutella25-EC16	5152	100669	16	16	82416	
rome99-EC2	3352	9869	2	2	6702	road network [5]
rome99-EC4	3352	15468	4	4	13404	
rome99-EC8	3352	31952	8	8	26808	
rome99-EC16	3352	65542	16	16	53616	
s38584-EC2	16310	42128	2	2	32618	VLSI circuit [4]
s38584-EC4	16310	80250	4	4	65236	
s38584-EC8	16310	160297	8	8	130472	
s38584-EC16	16310	323963	16	16	260944	
web-Stanford-EC2	150475	2334929	2	2	300948	web graph [19]
web-Stanford-EC4	150475	3307506	4	4	601896	
web-Stanford-EC8	150475	2379878	8	8	1203792	
web-Stanford-EC16	150475	3643794	16	16	2407584	

From the results, we observe that TL has overall the worst performance, even compared to Tar despite the inferior upper bound of the latter. Indeed, on average Tar runs twice as fast compared to TL. This is due to the overhead incurred in TL for splitting a graph G into

Table 3 Running times in seconds of the algorithms for the augmented graphs of Table 2. For the algorithms of Tarjan (Tar), and of Tong and Lawler (TL), we report the running time when the input is the original graph G (above) and a complete k intersection of G (below). The execution of an algorithm was terminated if it exceeded 1 hour.

Graph	Gab-EC	Gab-EC-DFS	Tar	TL	Gab	Gab-DFS
enron-EC2	0.01	0.01	0.01 0.01	1.09 0.04	0.15	0.01
enron-EC4	0.02	0.01	36.34 7.43	118.17 7.59	9.81	0.05
enron-EC8	0.06	0.01	304.29 95.58	614.18 131.91	38.42	0.28
enron-EC16	0.17	0.02	1840.76 951.43	2782.78 1319.77	130.58	1.50
p2p-Gnutella25-EC2	0.01	0.01	1.19 0.79	3.90 1.12	1.39	0.02
p2p-Gnutella25-EC4	0.02	0.01	8.46 7.05	23.76 14.34	6.57	0.04
p2p-Gnutella25-EC8	0.04	0.01	55.21 53.41	124.36 91.05	21.63	0.17
p2p-Gnutella25-EC16	0.09	0.01	403.92 450.80	760.74 629.79	61.32	0.80
rome99-EC2	0.01	0.01	0.45 0.35	0.37 0.24	0.36	0.34
rome99-EC4	0.02	0.01	3.20 3.19	7.26 5.56	2.63	0.02
rome99-EC8	0.03	0.01	21.67 21.62	50.86 37.24	8.65	0.10
rome99-EC16	0.06	0.01	160.01 156.29	312.30 241.95	25.32	0.49
s38584-EC2	0.03	0.01	14.12 8.88	37.02 23.45	14.73	12.76
s38584-EC4	0.06	0.01	131.86 117.79	262.07 98.85	75.42	0.14
s38584-EC8	0.14	0.02	1012.85 878.79	2007.45 1258.60	245.01	0.68
s38584-EC16	0.34	0.04	>1h >1h	>1h >1h	717.20	3.29
web-Stanford-EC2	0.60	0.29	>1h 2307.26	>1h 2350.42	520.22	1.39
web-Stanford-EC4	1.45	0.69	>1h >1h	>1h >1h	>1h	5.40
web-Stanford-EC8	3.26	1.03	>1h >1h	>1h >1h	>1h	14.32
web-Stanford-EC16	7.73	3.28	>1h >1h	>1h >1h	>1h	70.98

two auxiliary graphs G_1 and G_2, and manipulating mappings from the vertex ids of G_1 and G_2 to those in G. Furthermore, we observe that TL runs consistently faster on the complete k intersection T of G compared to the original graph G. While this is expected since T has fewer edges, on the other hand we note that it may be easier to find good candidate edges to augment a partial arborescence if the graph contains some additional edges. The same

observation holds for Tar as well, but here we see that in one instance (p2p-Gnutella25-EC16) the algorithm runs faster on G rather than on T. Moreover, we note that the executions of TL and Tar on T outperform Gab in some instances.

Next, we turn to the algorithms of Gabow. First, we verify that the edge-connectivity algorithms Gab-EC and Gab-EC-DFS are very effective. Regarding the arborescence packing algorithms, we first note that Tar and TL perform close to Gab when the edge-connectivity $c_G(s)$ is small (EC2 instances), but quickly become uncompetitive when the edge-connectivity increases. Overall, in our experiment, Gab was 50% faster than Tar on average. Finally, we note that Gab-DFS is faster than Gab by two orders of magnitude on most instances. This is due to the fact that our simple heuristic very often manages to construct a good arborescence by a simple DFS traversal.

k-cores

A k-core of an undirected graph G is a maximal subgraph of G such that $\delta(v) \geq k$ for all $v \in V(H)$. This concept is useful in the analysis of social networks [3] as well as in several other applications [22]. In this experiment, we use the k-core decomposition algorithm of the SNAP software library and tools [20], and use subgraphs of this decomposition as inputs, for various values of k. We transform each such undirected graph to a directed graph by orienting each edge in both directions. Table 4 reports the characteristics of the resulting graphs. In Table 5 we report the corresponding running times of each algorithm. Again, we terminated the execution of an algorithm if that exceeded one hour.

Here too, we observe that Tar outperforms TL on most instances, but unlike the augmented graphs, their difference is marginal. Both TL and Tar run consistently faster on the complete k intersection T of G compared to the original graph G. Again, the executions of TL and Tar on T outperform Gab in some instances, but overall Gab is 50% faster. Also, our heuristic was very effective in this experiment as well, since Gab-DFS ran faster than Gab by two orders of magnitude.

Table 4 Characteristics of k-core graphs, extracted from real-world graphs in [19]; n is the number of vertices, m the number of edges; δ denotes the minimum vertex degree, and $c_s(G)$ denotes the cardinality of the minimum s-cut (which equal the edge-connectivity since the graphs are undirected); m' is the number of edges in a complete k intersection.

Graph	n	m	δ	$c_s(G)$	m'	type and source
facebook_combined-core02	3964	176318	2	2	7926	social circles from facebook
facebook_combined-core04	3754	175332	4	4	11258	
facebook_combined-core25	1366	118810	25	25	6824	
facebook_combined-core50	616	75246	50	30	19064	
Email-Enron-core09	5088	206472	9	9	45783	email network
Email-Enron-core10	4513	196594	10	10	45120	
Email-Enron-core16	2873	157506	16	16	45952	
Email-Enron-core18	2561	147332	18	18	46080	
CA-AstroPh-core18	5049	244004	18	18	90864	collaboration network
CA-AstroPh-core25	3202	175520	4	4	12804	
CA-AstroPh-core29	2441	139070	29	2	4880	
CA-AstroPh-core32	1926	112830	32	32	61600	
Gowalla_edges-core11	22742	851196	11	6	136443	social network
Gowalla_edges-core12	19938	791666	12	5	99684	
Gowalla_edges-core15	13833	639244	15	4	55328	
Gowalla_edges-core20	8161	456014	20	8	65280	

Table 5 Running times in seconds of the algorithms for the k-core graphs of Table 4. For the algorithms of Tarjan (Tar), and of Tong and Lawler (TL), we report the running time when the input is the original graph G (above) and a complete k intersection of G (below). The execution of an algorithm was terminated if it exceeded 1 hour.

Graph	Gab-EC	Gab-EC-DFS	Tar	TL	Gab	Gab-DFS
facebook_combined-core02	0.01	0.01	2.47 0.22	8.71 0.43	2.00	0.01
facebook_combined-core04	0.01	0.01	6.88 0.29	20.32 0.86	2.88	0.02
facebook_combined-core25	0.01	0.01	3.97 0.29	6.82 0.53	0.95	0.02
facebook_combined-core50	0.04	0.01	43.41 29.38	47.76 22.59	4.59	0.55
Email-Enron-core09	0.03	0.01	109.84 44.41	156.29 55.41	30.06	0.17
Email-Enron-core10	0.03	0.01	107.00 47.09	101.79 54.95	28.60	0.17
Email-Enron-core16	0.04	0.01	136.26 106.58	170.74 74.56	20.65	0.35
Email-Enron-core18	0.04	0.01	151.68 119.37	177.93 77.12	19.80	0.42
CA-AstroPh-core18	0.25	0.01	894.66 716.73	1029.08 412.42	63.18	1.86
CA-AstroPh-core25	0.02	0.01	6.74 1.83	16.29 1.16	2.03	0.04
CA-AstroPh-core29	0.01	0.01	0.72 0.08	2.68 0.21	0.53	0.01
CA-AstroPh-core32	0.15	0.01	556.99 537.45	545.93 270.58	28.09	1.61
Gowalla_edges-core11	0.10	0.06	2086.57 567.03	2559.07 394.37	404.29	0.49
Gowalla_edges-core12	0.07	0.04	915.08 80.10	1094.83 166.56	219.33	0.31
Gowalla_edges-core15	0.04	0.02	197.84 28.64	321.91 49.67	71.65	0.15
Gowalla_edges-core20	0.05	0.02	327.64 118.93	412.93 109.56	68.57	0.32

References

1 András A Benczúr and David R Karger. Augmenting undirected edge connectivity in Õ(n2) time. *Journal of Algorithms*, 37(1):2–36, 2000. doi:10.1006/jagm.2000.1093.

2 Anand Bhalgat, Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. Fast edge splitting and edmonds' arborescence construction for unweighted graphs. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 455–464, USA, 2008. Society for Industrial and Applied Mathematics.

3 Kshipra Bhawalkar, Jon Kleinberg, Kevin Lewi, Tim Roughgarden, and Aneesh Sharma. Preventing unraveling in social networks: The anchored k-core problem. *SIAM Journal on Discrete Mathematics*, 29(3):1452–1475, 2015.

4 CAD Benchmarking Lab. ISCAS'89 benchmark information. http://www.cbl.ncsu.edu/www/CBL_Docs/iscas89.html.

5 C. Demetrescu, A.V. Goldberg, and D.S. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/~challenge9/, 2007.

6 J. Edmonds. Submodular functions, matroids, and certain polyhedra. *Combinatorial Structures and their Applications*, pages 69–81, 1970.

7 J. Edmonds. Edge-disjoint branchings. *Combinatorial Algorithms*, pages 91–96, 1972.

8 D. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, USA, 2010.

9 S. Fujishige. A note on disjoint arborescences. *Combinatorica*, 30(2):247–252, 2010. doi:10.1007/s00493-010-2518-y.

10 Satoru Fujishige and Naoyuki Kamiyama. The root location problem for arc-disjoint arborescences. *Discrete Applied Mathematics*, 160(13):1964–1970, 2012. doi:10.1016/j.dam.2012.04.013.

11 H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50:259–273, 1995.

12 Harold N. Gabow. Efficient splitting off algorithms for graphs. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 696–705, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/195058.195436.

13 Loukas Georgiadis, Dionysios Kefallinos, Luigi Laura, and Nikos Parotsidis. An experimental study of algorithms for computing the edge connectivity of a directed graph. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 85–97. SIAM, 2021. doi:10.1137/1.9781611976472.7.

14 M. Ghaffari, K. Nowicki, and M. Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, pages 1260–1279, USA, 2020. Society for Industrial and Applied Mathematics.

15 M. Henzinger, S. Rao, and D. Wang. Local flow partitioning for faster edge connectivity. *SIAM Journal on Computing*, 49(1):1–36, 2020.

16 N. Kamiyama, N. Katoh, and A. Takizawa. Arc-disjoint in-trees in directed graphs. *Combinatorica*, 29:197–214, 2009. doi:10.1007/s00493-009-2428-z.

17 D. R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, January 2000. doi:10.1145/331605.331608.

18 K.-I. Kawarabayashi and M. Thorup. Deterministic edge connectivity in near-linear time. *Journal of the ACM*, 66(1), December 2018. doi:10.1145/3274663.

19 J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

20 Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

21 László Lovász. On two minimax theorems in graph. *Journal of Combinatorial Theory, Series B*, 21(2):96–103, 1976. doi:10.1016/0095-8956(76)90049-6.

22 F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis. The core decomposition of networks: theory, algorithms and applications. *The VLDB Journal*, 29(1):61–92, 2020. doi:10.1007/s00778-019-00587-4.

23 H. Nagamochi and T. Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Cambridge University Press, 2008. 1st edition.

24 Yossi Shiloach. Edge-disjoint branching in directed multigraphs. *Information Processing Letters*, 8(1):24–27, 1979. doi:10.1016/0020-0190(79)90086-3.

25 Robert Endre Tarjan. A good algorithm for edge-disjoint branching. *Information Processing Letters*, 3(2):51–53, 1974. doi:10.1016/0020-0190(74)90024-6.

26 Po Tong and E.L. Lawler. A faster algorithm for finding edge-disjoint branchings. *Information Processing Letters*, 17(2):73–76, 1983. doi:10.1016/0020-0190(83)90073-X.

A Stack-based vs queue-based implementations

Tables 6 and 4 compare the stack-based against the queue-based implementation of the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), for augmented and k-core graphs, respectively.

Table 6 Running times in seconds of the stack-based and queue-based implementations of the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), for the augmented graphs of Table 2. We report the running time when the input is the original graph G (above) and a complete k intersection of G (below). The execution of an algorithm was terminated if it exceeded 1 hour.

Graph	Tar		TL	
	stack	queue	stack	queue
enron-EC2	0.01	0.01	1.09	2.41
	0.01	0.01	0.04	0.48
enron-EC4	36.34	111.44	118.17	131.24
	7.23	8.18	7.59	6.73
enron-EC8	304.29	591.30	614.18	786.56
	95.58	136.25	131.91	118.14
enron-EC16	1840.76	2662.05	2782.78	3340.18
	951.43	1298.16	1319.77	1343.09
p2p-Gnutella25-EC2	1.19	3.66	3.90	4.41
	0.79	1.13	1.12	1.59
p2p-Gnutella25-EC4	8.46	22.00	23.76	23.61
	7.05	14.67	14.34	14.73
p2p-Gnutella25-EC8	55.21	123.56	124.36	119.05
	53.41	99.95	91.005	88.94
p2p-Gnutella25-EC16	403.92	744.87	760.74	768.10
	450.80	662.15	629.79	620.41
rome99-EC2	0.45	0.40	0.37	0.45
	0.35	0.32	0.24	0.21
rome99-EC4	3.20	8.12	7.26	6.98
	3.19	6.08	5.56	3.88
rome99-EC8	21.67	49.48	50.86	47.44
	21.62	37.11	37.24	35.71
rome99-EC16	160.01	306.58	312.30	320.36
	156.29	239.31	241.95	243.20
s38584-EC2	14.12	39.62	37.02	44.71
	8.88	25.96	23.45	26.93
s38584-EC4	131.86	286.66	262.07	282.07
	117.79	102.47	98.85	114.64
s38584-EC8	1012.85	2040.86	2007.45	1867.42
	878.22	1541.89	1258.60	1265.24
s38584-EC16	>1h	>1h	>1h	>1h
	>1h	>1h	>1h	>1h
web-Stanford-EC2	>1h	>1h	>1h	>1h
	2307.26	3100.36	2350.42	>1h
web-Stanford-EC4	>1h	>1h	>1h	>1h
	>1h	>1h	>1h	>1h

Table 7 Running times in seconds of the stack-based and queue-based implementations of the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), for the k-core graphs of Table 4. We report the running time when the input is the original graph G (above) and a complete k intersection of G (below). The execution of an algorithm was terminated if it exceeded 1 hour.

Graph	Tar		TL	
	stack	queue	stack	queue
facebook_combined-core02	2.47	8.04	8.71	6.63
	0.22	0.37	0.43	0.36
facebook_combined-core04	6.88	18.83	20.32	11.51
	0.29	0.83	0.86	0.95
facebook_combined-core25	3.97	6.32	6.82	3.97
	0.29	0.51	0.53	0.31
facebook_combined-core50	43.41	44.34	47.76	8.54
	29.38	21.88	22.59	3.11
Email-Enron-core09	109.84	146.24	156.29	221.64
	44.41	55.64	55.41	47.98
Email-Enron-core10	107.00	94.40	101.79	116.07
	47.09	53.62	54.95	41.25
Email-Enron-core16	136.26	161.54	170.74	226.42
	106.58	90.43	74.56	94.21
Email-Enron-core18	151.68	168.80	177.93	240.51
	119.37	90.58	77.12	104.76
CA-AstroPh-core18	894.66	991.31	1029.08	619.59
	716.73	99.58	412.42	213.50
CA-AstroPh-core25	6.74	15.42	16.29	19.77
	1.83	1.66	1.16	0.75
CA-AstroPh-core29	0.72	2.50	2.68	3.02
	0.08	0.19	0.21	0.10
CA-AstroPh-core32	556.99	526.15	545.93	991.77
	537.45	427.66	270.58	349.73
Gowalla_edges-core11	2086.57	2468.99	2559.07	>1h
	567.03	455.23	394.37	403.34
Gowalla_edges-core12	915.08	1425.66	1094.83	2482.81
	80.10	187.11	166.56	169.45
Gowalla_edges-core15	197.84	498.99	321.91	780.18
	28.64	53.81	49.67	47.25
Gowalla_edges-core20	327.64	552.16	412.93	847.61
	118.93	122.30	109.56	105.48

Stochastic Route Planning for Electric Vehicles

Payas Rajan
Department of Computer Science, University of California, Riverside, CA, USA

Chinya V. Ravishankar
Department of Computer Science, University of California, Riverside, CA, USA

Abstract

Electric Vehicle routing is often modeled as a generalization of the energy-constrained shortest path problem, taking travel times and energy consumptions on road network edges to be deterministic. In practice, however, energy consumption and travel times are stochastic distributions, typically estimated from real-world data. Consequently, real-world routing algorithms can make only probabilistic feasibility guarantees. Current stochastic route planning methods either fail to ensure that routes are energy-feasible, or if they do, have not been shown to scale well to large graphs. Our work bridges this gap by finding routes to maximize on-time arrival probability and the set of non-dominated routes under two criteria for stochastic route feasibility: $\mathbb{E}$-feasibility and p-feasibility. Our $\mathbb{E}$-feasibility criterion ensures energy-feasibility in expectation, using expected energy values along network edges. Our p-feasibility criterion accounts for the actual distribution along edges, and keeps the stranding probability along the route below a user-specified threshold p. We generalize the charging function propagation algorithm to accept stochastic edge weights to find routes that maximize the probability of on-time arrival, while maintaining $\mathbb{E}$- or p-feasibility. We also extend multi-criteria Contraction Hierarchies to accept stochastic edge weights and offer heuristics to speed up queries. Our experiments on a real-world road network instance of the Los Angeles area show that our methods answer stochastic queries in reasonable time, that the two criteria produce similar routes for longer deadlines, but that $\mathbb{E}$-feasibility queries can be much faster than p-feasibility queries.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Shortest paths; Theory of computation → Stochastic control and optimization

Keywords and phrases Stochastic Routing, Electric Vehicles, Route Planning Algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.15

Acknowledgements The authors would like to thank the Mapbox Community team for access to the Mapbox Traffic Data used in our experiments.

1 Introduction

Routing methods for Electric Vehicles (EVs) cannot just minimize travel time, but must also address driver *range anxiety*. EVs have limited battery capacity, charging times are long, and the charging infrastructure remains relatively sparse, so a major concern is *stranding*, which occurs when the battery's State of Charge (SoC) reaches zero en route. A route for an EV is hence considered *feasible* only if the SoC along the route never reaches zero.

Merely trying to minimize travel time greatly increases the risk of stranding, since energy consumption is typically quadratic in vehicle speed. Standard formulations such as [9, 14, 30] model EV routing as a generalization of the NP-hard Constrained Shortest Path problem [26, 58], and seek to minimize travel time while maintaining a non-zero SoC along the route. Some recent work [7, 36] even tries to exercise direct control over travel time and route feasibility, by pre-determining and assigning optimal EV travel speeds for each road segment.

Most existing problem formulations also assume that travel times and energy consumption values on road network edges are deterministic. In practice, both travel time and energy consumption are stochastic, and difficult to estimate reliably [2, 17, 46]. In such a context, even routing algorithms offering strong feasibility guarantees are of limited value. Approaches that pre-determine and assign vehicle speeds for each edge are not practical, since speed is not always a variable under driver control, but rather a *result* of prevalent traffic conditions.

Consequently, travel times and route feasibility may only be defined *probabilistically*. Stochastic routing algorithms [12, 20, 40, 39, 38, 42, 43, 44, 27, 59], model travel times along network edges as random variables with given probability distributions, and allow richer query semantics, such as finding paths to maximize the probability of arrival before a deadline [15], or finding the latest departure time and path to guarantee a certain probability of arrival before a deadline [35]. Despite recent improvements [47], stochastic routing is typically several orders of magnitude slower than deterministic routing, since obtaining travel time distributions along a path requires very expensive convolutions of its edge distributions.

Limited work exists on stochastic route planning for EVs. Chen et al. [13] assume lognormal travel-time and Gaussian energy-consumption distributions, and uses bicriteria search to find the Pareto-optimal routes optimizing energy consumption and travel time reliability. Jafari et al. [25] allow arbitrary distributions of travel times on edges and charging stations, and uses multicriteria search to minimize the cost of charging and travel time, subject to a minimum reliability threshold, on small synthetic graphs with randomly generated edge weights and charging station placements. Shen et al. [51] allow correlated travel time distributions between edges, and use bicriteria search on travel times and energy consumptions. However, they assume deterministic energy consumptions, and run experiments on a network of only a few hundred vertices.

1.1 Our Contributions

We study EV routing when both travel times and energy consumptions are stochastic. The travel time on each edge $e \in E$ of a road network $G = \langle V, E \rangle$ is always a random variable $\boldsymbol{T}_e$ with a known distribution (estimated from data, say). The energy consumption along e is a function ε_e of EV speed and distance. We introduce two probabilistic definitions of route feasibility: We say that a route is $\mathbb{E}$-feasible if the SoC of the EV is always maintained above zero *in expectation*, and p-feasible if the probability of route feasibility is *at least* p. We show how to enhance stochastic routing queries for travel times with these feasibility criteria to find non-dominated feasible routes and probabilistic budget feasible routes. Our work addresses the four types of stochastic routing queries in the cells of the following table:

	$\mathbb{E}$-Feasibility	p-Feasibility
Non-Dominated Routes	✓	✓
Probabilistic Budget Routes	✓	✓

We address these queries by generalizing the Charging Function Propagation algorithm of [8, 9] to accommodate stochastic edge weights. We evaluate our methods experimentally using a realistic road network instance with travel time distributions derived from traffic speeds observed over four and a half months in the Los Angeles area, and real-world elevations and charging station locations. Further, we apply an uncertain variant of Contraction Hierarchies [21] to speed up our queries and present results. Our results indicate that in general, $\mathbb{E}$-feasible routing queries can be computed much faster than p-feasible queries, and produce similar routes for longer routes with higher time budgets.

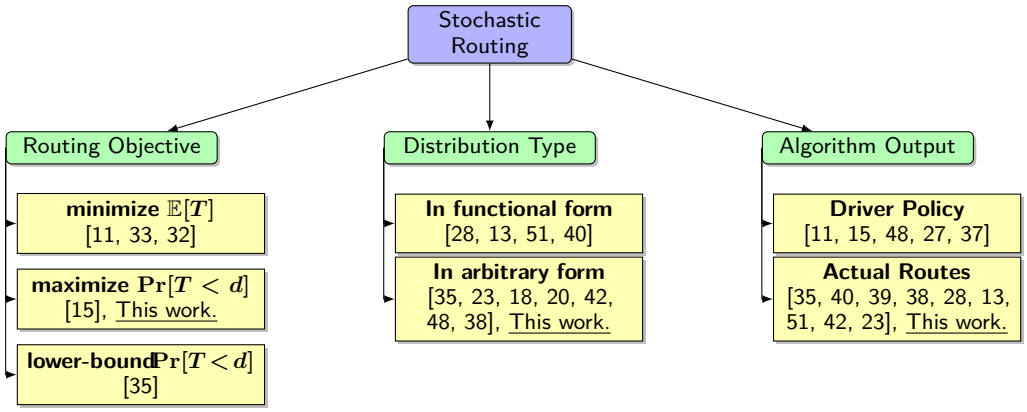


Figure 1 Stochastic route planning, classified by routing objective, edge distribution, and result. Our work finds energy-feasible routes that maximize probability of arrival before deadline.

2 Related Work

EV routing has been typically modeled as energy-aware routing, with objective functions ranging from minimizing the total energy consumption [14, 49], to minimizing travel time while maintaining route *feasibility* [3, 9, 36, 54], to multicriteria search on both travel time and energy consumption [22]. In contrast, most prior work on routing Internal Combustion-based vehicles merely minimizes the total travel time [4, 53].

More attention is now being paid to real-world issues. Examples include allowing battery-swapping stations [56], partial recharges at stations [9, 30, 8, 57] and maintaining battery buffer to relieve range anxiety [45, 24, 18]. Many challenges remain, however. The energy consumption models are imperfect, and factors such as battery wear, driver aggressiveness [31], or traffic conditions are hard to model, but can have a significant impact. Data also suggests that drivers may prefer familiar paths to shortest paths [60, 29].

2.1 Stochastic Route Planning

Stochastic route planning goes back to [20], which attempted an exact solution for the shortest path problem in stochastic graphs, using Monte Carlo simulations to derive path weights. It is now known that driver behavior changes if travel time is stochastic [19, 52], so problem variants have been explored. Existing work can be categorized in three ways: by objective function, the forms assumed for edge probability distributions, and by targeted outcome. For conciseness, we discuss our categorization here, and show references in Figure 1.

By routing objective. Routing objectives can be quite varied, such as minimizing expected time [11, 33, 32], maximizing the on-time-arrival probability [15], maintaining on-time-arrival probability above a given threshold [35]. Some works [51, 13, 25] apply stochastic routing algorithms to EVs, while others [1] route multiple EVs collaboratively, on-line.

By distribution. The edge distributions assumed can have a functional form, or be arbitrary without a closed form. This choice also affects the edge weight representations used. For functional forms, storing the distributional parameters suffices, but arbitrary distributions require more space-intensive representations such as histograms. Further, with functional forms, a small number of observations can suffice to capture real-world behaviour, but histograms require much more data. Edge weight representations have been shown to significantly affect the runtime performance of stochastic shortest path queries [44, 47].

Output. Adaptive methods [41, 35] output policies for drivers to make routing decisions on-line, as they reach vertices or edges during the drive. In contrast, a-priori approaches produce routes before travel begins. [35] showed that adaptive approaches can produce strictly better solutions than the a-priori approaches, but are much more computationally expensive. Recently, performance improvements to policy-based approaches, such as the Stochastic On-Time Arrival problem have also been proposed [48, 37, 27].

3 Problem Setup

A road network is a directed graph $G = \langle V, E \rangle$ where V is the set of vertices and $E : V \times V$ is the set of edges. An s-t path $P = [s = v_1, v_2 \cdots, v_n = t]$ is a sequence of adjacent vertices in the road network G. A set $C \subseteq V$ is marked as *charging stations*.

▶ **Definition 1** (State of Charge). *The State of Charge (SoC) of an EV is the charge status of the EV's battery, lying between 0 and the battery capacity M. We denote the SoC on arrival at a vertex v by ${}_v\beta$ and the SoC at departure from v by β_v. We have $\beta_v \geq {}_v\beta$ if the EV charges its batteries at node v, and $\beta_v = {}_v\beta$ otherwise.*

Each $c \in C$ has a monotonically increasing, piecewise-linear charging function Φ_c such that $\Phi_c({}_c\beta, t_c) \mapsto \beta_c$ where t_c is charging time. We require ${}_c\beta \geq 0$, and $\beta_c \leq M$ [45].

▶ **Definition 2** (Leg and Prefix). *A subpath $L = [c_1, \ldots, v, \ldots, c_2]$ is a* leg *of path P iff c_1, c_2 are successive charging stations along P. Each $\lambda_v = [c_1, \ldots, v]$, $v \neq c_2$ is a* prefix *of L.*

3.1 Travel Times and Energy Depletion

The travel time along each edge e is a random variable $\boldsymbol{T}_e$ with a known probability distribution. For problem tractability, we assume that the EV travels on e at a uniform speed drawn from the distribution $\boldsymbol{T}_e$. This is reasonable, since variable travel time on an edge can be easily modeled by splitting an edge into several smaller edges.

Let $e_1, e_2, \ldots, e_{n-1}$ be the edges along path P, and let e_k have travel time distribution T_k. The aggregate travel time distribution for the path P is $\boldsymbol{T}_P = \boldsymbol{T}_1 * \boldsymbol{T}_2 * \cdots * \boldsymbol{T}_{n-1}$, where $*$ denotes linear convolution. Let $\boldsymbol{T}_\emptyset$ be the Dirac "delta" distribution defined so that $\boldsymbol{T}_\emptyset(0) = 1$ and $\boldsymbol{T}_\emptyset(x) = 0$ at $x \neq 0$. Now, $\boldsymbol{T}_\emptyset$ functions as a convolution identity, so $\boldsymbol{T}_\emptyset * \boldsymbol{T}_P = \boldsymbol{T}_P$.

We assign to each edge e a function $\varepsilon_e : \mathbb{R}^+ \to \mathbb{R}$, which maps a travel time to the battery energy depleted by travel along e. The total energy depletion is the sum of the work done along e by the EV against air resistance, rolling resistance, and against gravity. The wind resistance grows quadratically with speed. If t is the travel time along edge e, these three terms cause $\varepsilon_e(t)$ to assume the form

$$\varepsilon_e(t) = \frac{a_e}{t^2 - b_e} + \frac{c_e}{t} + d_e. \tag{1}$$

where a_e, b_e, c_e, d_e are fixed coefficients for each edge e. We can derive the edge *energy depletion distribution* $\boldsymbol{D}_e$ from the travel time distribution $\boldsymbol{T}_e$ using Equation 1, thereby associating probabilities with energy depletions. A path may have negative energy depletion; EVs have regenerative brakes, and can accumulate charge, say, when going down a slope.

We can also aggregate energy depletion distributions using convolutions. If $e_1, e_2, \ldots, e_{n-1}$ are the edges along a path P, and edge e_i has the depletion distribution $\boldsymbol{D}_i$, the aggregate energy depletion distribution for P is $\boldsymbol{D}_P = \boldsymbol{D}_1 * \boldsymbol{D}_2 * \cdots * \boldsymbol{D}_{n-1}$. By analogy with $\boldsymbol{T}_\emptyset$, we define $\boldsymbol{D}_\emptyset$ to be the Dirac "delta" function corresponding to energy depletion, so that $\boldsymbol{D}_\emptyset * \boldsymbol{D}_P = \boldsymbol{D}_P$. Sometimes, as with expected-feasibility queries, it suffices to add expectations directly, since $\mathbb{E}[\boldsymbol{D}_1 * \boldsymbol{D}_2] = \mathbb{E}[\boldsymbol{D}_1] + \mathbb{E}[\boldsymbol{D}_2]$.

Table 1 Symbols used in this paper.

Sym	Meaning	Sym	Meaning
$\boldsymbol{T}_P$	Travel time distribution on path P	$\boldsymbol{T}_\emptyset$	Convolution identity for $\boldsymbol{T}$
$\boldsymbol{D}_P$	Energy depletion distribution on path P	$\boldsymbol{D}_\emptyset$	Convolution identity for $\boldsymbol{D}$
δ_λ	Depletion function for leg prefix λ	$\delta_\emptyset$	Depletion function for null path
${}_u\beta$	SoC on arrival at vertex u	β_u	SoC at departure from vertex u
Φ_c	Charging function at charging station c	ε_e	Energy depletion function on edge e

3.2 $\mathbb{E}$-Feasible Routing

In this class of queries, we assume that the travel times are stochastic, but define feasibility in terms of the expectations for the energy depletion distributions. Say that an EV starts from vertex s with State of Charge (SoC) $\beta_s \in [0, M]$ and wishes to travel to vertex t along the s-t path P. Let leg $L = [c, \ldots, c']$ of P lie between charging stations c and c' along P.

▶ **Definition 3** ($\mathbb{E}$-Feasible Path). *Leg L is* expected-feasible *(or* $\mathbb{E}$-feasible*) iff $\mathbb{E}[\boldsymbol{D}_\lambda] \leq \beta_c$, where $\boldsymbol{D}_\lambda$ is the depletion distribution for all prefixes λ of L, and β_c is the EV's SoC when it departs c. A path $P = [L_1, L_2, \ldots, L_n]$ is $\mathbb{E}$-feasible iff each of its legs L_i is $\mathbb{E}$-feasible.*

We consider two $\mathbb{E}$-feasible queries:

▶ **Query 4** (Non-Dominated $\mathbb{E}$-feasible Paths). *Find the set of $\mathbb{E}$-feasible s-t paths such that their travel time distributions are not dominated by any other path.*

▶ **Query 5** (Probabilistic Budget $\mathbb{E}$-feasible Path). *Find an $\mathbb{E}$-feasible s-t path that maximizes the probability of reaching t before a given deadline d.*

3.3 p-Feasible Routing

▶ **Definition 6** (p-Feasible Path). *A path P with legs $L_1, L_2, \ldots, L_n$ is* p-feasible *iff the probability of the EV not being stranded along P is at least p, the non-stranding probability.*

The non-stranding probability of P is given by the product of non-stranding probabilities of P's legs. For P to be p-feasible, each of its legs must have a non-stranding probability of at least p. We consider two p-feasible queries:

▶ **Query 7** (Non-Dominated p-Feasible Paths). *Find the set of s-t paths whose travel time distributions are not dominated by any other path, and which ensure that probability not being stranded is at least p.*

▶ **Query 8** (Probabilistic Budget p-Feasible Paths). *Find an s-t path which maximizes the probability of reaching t before a given deadline d, while keeping the probability of not being stranded is at least p.*

4 Charging Function Propagation for $\mathbb{E}$-Feasible Routing

The CFP algorithm of [9] uses only deterministic edge weights, but we show how to extend it to answer expected-feasible stochastic shortest path queries. As in [9], we ensure that the SoC on departing a charging station suffices to complete the ensuing leg. Our Dijkstra's search labels maintain the set of all possible tradeoffs between charging time and the resulting SoC.

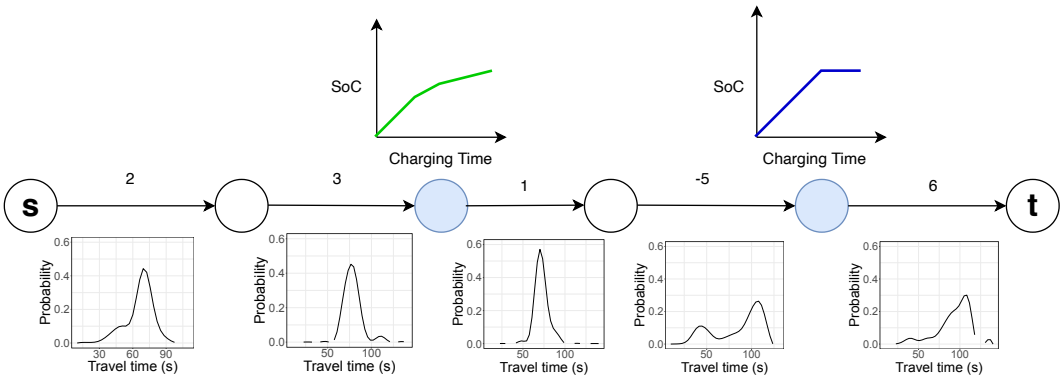


Figure 2 $\mathbb{E}$-feasible queries. Edges have two weights: a travel time distribution (below), and an expected energy depletion (above). Shaded nodes are charging stations, with piecewise-linear charging functions. The CFP search propagates travel time distributions using convolutions.

However, complexities arise since we use stochastic travel times. The deterministic case can simply use a min-priority queue ordered by travel times, but distributions can be ordered in different ways. For simplicity, we will use *usual stochastic ordering* [50] to order the travel time distributions in the priority queue, under which two random variables X and Y obey $X \preceq Y$ iff $\Pr[X > x] \leq \Pr[Y > x], \forall x \in (-\infty, \infty)$. Other stochastic orderings, such as the hazard rate or likelihood ratio ordering, may result in interesting tradeoffs for the EV, but are beyond the scope of this paper. Also, deterministic travel times can be simply added along a path, but travel time distributions must be convolved to aggregate travel time distributions.

For expected-feasible routes, we will use stochastic travel times, but expected values for energy depletion. That is, let $e_1, e_2, \ldots, e_n$ be the edges comprising a path P, and let edge e_i have travel time and energy depletion distributions $\boldsymbol{T}_i$ and $\boldsymbol{D}_i$. For expected-feasible routing, the aggregate travel time distribution $\boldsymbol{T}_P = \boldsymbol{T}_1 * \boldsymbol{T}_2 * \cdots * \boldsymbol{T}_n$, and the aggregate energy depletion value is $\mathbb{E}[\boldsymbol{D}_P] = \mathbb{E}[\boldsymbol{D}_1] + \mathbb{E}[\boldsymbol{D}_2] + \ldots + \mathbb{E}[\boldsymbol{D}_n]$.

4.1 The Depletion Function Along Route Legs

Even if the energy depletion over leg $L = [c_1, \ldots, v, \ldots, c_2]$ is deterministic with value E_L, departing c_1 with an SoC of $\beta_{c_1} = E_L$ may not suffice to complete L. For instance, L may go up a hill, climbing which requires more energy than β_{c_1}. Similarly, ${}_{c_2}\beta$, the arrival SoC at c_2, may not equal $\beta_{c_1} + E_L$ when $E_L < 0$, since the SoC can never exceed M.

Consider a prefix $\lambda = [c, \ldots, v]$ of some leg that starts with charging station c. Let S_λ be the minimum starting SoC required to traverse λ, E_λ be the maximum ending SoC possible at v, and let $\mathrm{C}_\lambda = \mathbb{E}[\boldsymbol{D}_\lambda]$. The *depletion function* δ_λ (similar to *SoC profiles* in [8, 9]) for prefix λ maps the SoC at the start of λ to the SoC at the end of λ, and is defined as

$$\delta_\lambda(\beta_c) = {}_v\beta = \begin{cases} -\infty, & \text{if } \beta_c < \mathrm{S}_\lambda, \\ \mathrm{E}_\lambda, & \text{if } \beta_c - \mathrm{C}_\lambda > \mathrm{E}_\lambda, \\ \beta_c - \mathrm{C}_\lambda, & \text{otherwise.} \end{cases} \tag{2}$$

The depletion function for a null path comprising a single vertex s is the *identity* depletion function $\delta_\emptyset : \beta_s \mapsto \beta_s$. Let $P_1 = [v_i, v_{i+1}, \ldots, v_j]$ and $P_2 = [v_{j+1}, v_{j+2}, \ldots, v_k]$ be contiguous segments, and $P = P_1P_2 = [v_i, \ldots, v_k]$ be their concatenation. In this case, we have $\mathrm{S}_P = \max\{\mathrm{S}_{P_1}, \mathrm{C}_{P_1} + \mathrm{S}_{P_2}\}$, $\mathrm{E}_P = \min\{\mathrm{E}_{P_2}, \mathrm{E}_{P_1} - \mathrm{C}_{P_2}\}$ and $\mathrm{C}_P = \mathrm{C}_{P_1} + \mathrm{C}_{P_2}$.

4.2 Dijkstra Search for $\mathbb{E}$-feasible Routes

We find expected-feasible paths via Dijkstra search using two types of priority queues: the global queue Q_G holds the travel time distributions from s to all other vertices in the road network G, and per-vertex queues $L_u(v)$ and $L_s(v)$. $L_u(v)$ and $L_s(v)$ hold the unsettled and settled search labels at vertex v respectively. All priority queues are ordered by $\boldsymbol{T}_{[s\ldots v]}$ using the usual stochastic ordering $\preceq$ defined above. Each label in $L_s(v)$ corresponds to an s-v path already known to be feasible, and gives the required charging time at the last charging station. Consequently, as in [9], we maintain the invariant that the minimum element in $L_u(v)$ is not dominated by any label in $L_s(v)$.

The EV leaves s having acquired an SoC of β_s at s, so we treat s as a charging station, by default. One of our major challenges in the search will be to determine at which stations to charge, and for how long. Our search hence remembers the last charging station c along the route in the search labels, since dropping to an SoC below a permissible threshold signals the need to include a charging time at c, and update route times accordingly.

4.2.1 The Search Algorithm

When the search reaches vertex v, the label at v is a four-tuple $\langle \boldsymbol{T}_{[s\ldots v]}, {}_c\beta, c, \delta_{[c\ldots v]}\rangle$, where $\boldsymbol{T}_{[s\ldots v]}$ is the travel time distribution for the subpath $[s\ldots v]$, c is the last charging station en route from s to v, ${}_c\beta$ is the arrival SoC at c, and $\delta_{[c\ldots v]}$ is the depletion function of the subpath $[c\ldots v]$. We note that the charging times at some charging stations may be zero.

A label is extracted from $L_u(v)$ on each search iteration, where v is the minimum-travel time vertex in Q_G. It is then settled, and added to $L_s(v)$. A label in $L_s(v)$ represents a path from s to v that we know to be feasible, along with the exact charging time at the last charging station. A label in $L_u(v)$ represents a potentially feasible path that we haven't checked for feasibility. If an unsettled label in $L_u(v)$ is dominated by a label in $L_s(v)$, we can discontinue search along that path and discard that label, because we already know a better feasible path. The search proceeds as follows:

1. *At s*: Mark s as a charging station. Add the label $\langle \boldsymbol{T}_\emptyset, {}_s\beta, s, \delta_\emptyset\rangle$ to $L_u(s)$.
2. *At a non-charging vertex v*: Let $\ell = \langle \boldsymbol{T}_{[s\ldots v]}, {}_c\beta, c, \delta_{[c\ldots v]}\rangle$ be the label extracted from $L_u(v)$. Since ℓ indicates that c is the last charging station encountered, add label $\langle \boldsymbol{T}_{[s\ldots v]}, \delta_{[c\ldots v]}({}_c\beta), c, \delta_{[c\ldots v]}\rangle$ to $L_u(v)$ and update the travel times for v in Q_G.
3. *At a charging vertex v*: Let label $\ell = \langle \boldsymbol{T}_{[s\ldots v]}, {}_c\beta, c, \delta_{[c\ldots v]}\rangle$ be the minimum element extracted from $L_u(v)$. Let t_c be the charging time at the last charging station c, so that $\beta_c = \Phi_c({}_c\beta, t_c)$ is the SoC when the EV departs c.
   The CFP algorithm of [8, 9] uses only deterministic travel times, but our travel times are distributions. As [8] shows, however, the charging times corresponding to the breakpoints of the charging function $\Phi_c(\cdot)$ capture the information required to make the required tradeoffs between charging times and travel times. To see how we approach the problem, let τ represent some value for the travel time from s to v, and compute

$$b_\ell(t_c, \tau) := \begin{cases} \delta_{[c\ldots v]}(\beta_c) & \text{if } t_c > 0 \text{ and } \boldsymbol{T}_{[s\ldots v]}(\tau) > 0 \\ -\infty & \text{otherwise} \end{cases}$$

   Since the charging function $\Phi_c(\cdot)$ is assumed to be piecewise linear, its breakpoints induce breakpoints for b_ℓ. For a given value of τ we need to create one label per breakpoint of b_ℓ [8]. For a fixed τ and each breakpoint $B = (t_B, \mathrm{SoC}_B)$ of b_ℓ, we add to $L_u(v)$ the label $\langle t_B, \mathrm{SoC}_B, v, \boldsymbol{T}_\emptyset\rangle$, and update the travel times to v in Q_G.

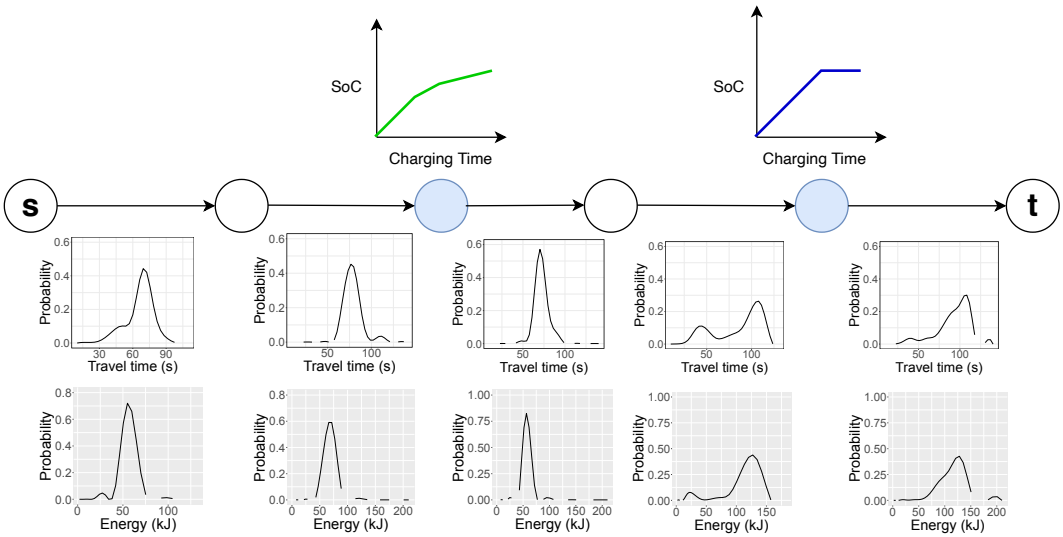


Figure 3 p-Feasible queries. Travel time and energy depletion are both distributions., propagated by the CFP search using convolutions. While the non-dominated search stops only when Q_G becomes empty, the probabilistic budget route search can stop when $\boldsymbol{T}_P(t)$ drops to 0.

In principle, τ can take an infinite number of values. We handle this difficulty by discretizing the domain of $\boldsymbol{T}_e$. We use histograms to represent $\boldsymbol{T}_e$ in our implementation, and generate only one set of breakpoints per histogram bin.

4. *At the destination* t: End search, backtrack using parent pointers to extract an s-t path.

A label ℓ is said to dominate another label ℓ' if $b_\ell(t,\tau) \geq b_{\ell'}(t,\tau)$ for all $t > 0$ and all $\tau > 0$.

If we end the search only when Q_G is empty, not simply when t is reached, we obtain the $\mathbb{E}$-feasible non-dominated paths. For $\mathbb{E}$-feasible probabilistic budget paths, we end the search when $\boldsymbol{T}_P(d) = 0$, i.e. the probability of reaching t within the time budget d drops to 0.

5 Charging Function Propagation for p-Feasible Routing

For p-feasible routing, we must consider the actual depletion distribution $\boldsymbol{D}_P$ for a path P, not simply $\mathbb{E}[\boldsymbol{D}_P]$, which sufficed for expected-feasible paths. As with expected-feasible paths, we must also deal with the travel time distribution $\boldsymbol{T}_P$. If path P has the edges $e_1, e_2, \ldots, e_n$, then $\boldsymbol{T}_P = \boldsymbol{T}_1 * \boldsymbol{T}_2 * \cdots * \boldsymbol{T}_n$ and $\boldsymbol{D}_P = \boldsymbol{D}_1 * \boldsymbol{D}_2 * \cdots * \boldsymbol{D}_n$. We can use p to place a bound on the maximum energy depletion we can accommodate over a path P. Let

$$\mathrm{c}_P(p) = \arg\max_x \{\boldsymbol{D}_P(x) \leq p\},$$

so that $\mathrm{c}_P(p)$ is the highest energy depletion that could occur along P with a probability of no more than p, that is, to ensure a non-stranding probability of p.

We define the *stochastic depletion function* analogously to Equation 2. Let $\mathrm{S}_P(p)$ be the minimum starting SoC at s required to traverse P with non-stranding probability p. Similarly, let $\mathrm{E}_P(p)$ be the maximum SoC possible on arriving at vertex t with at least probability p, and $\mathrm{C}_P = \boldsymbol{D}_P$. The *stochastic depletion function* for P is

$$\sigma_P(\beta_s, p) = {}_t\beta = \begin{cases} -\infty, & \text{if } \beta_s < \mathrm{S}_P(p), \\ \mathrm{E}_P(p), & \text{if } \beta_s - \mathrm{C}_P(p) > \mathrm{E}_P(p), \\ \beta_s - \mathrm{C}_P(p), & \text{otherwise.} \end{cases} \tag{3}$$

Let $\sigma_\emptyset$ be the *identity* stochastic depletion profile for a null path, so that $\sigma_\emptyset(\beta_s, p) = \beta_s$. If $P_1 = [v_i, v_{i+1}, \ldots, v_j]$ and $P_2 = [v_{j+1}, v_{j+2}, \ldots, v_k]$, the depletion profile of the concatenated path $P = P_1P_2 = [v_i \ldots v_k]$ is given by $\mathrm{S}_P(p) = \max\{\mathrm{S}_{P_1}(p), \mathrm{C}_{P_1}(p) + \mathrm{S}_{P_2}(p)\}$, $\mathrm{E}_P(p) = \min\{\mathrm{E}_{P_2}(p), \mathrm{E}_{P_1}(p) - \mathrm{C}_{P_2}(p)\}$ and $\mathrm{C}_P = \mathrm{C}_{P_1} * \mathrm{C}_{P_2}$.

5.1 Dijkstra Search for p-feasible Routes

The label at vertex v is a four-tuple $\langle \boldsymbol{T}_{[s\ldots v]}, {}_c\beta, c, \sigma_{[c\ldots v]} \rangle$, where $\boldsymbol{T}_{[s\ldots v]}$ is the travel time distribution for the subpath $[s \ldots v]$, c is the last charging station enroute from s to v, ${}_c\beta$ is the arrival SoC at c, and $\sigma_{[c\ldots v]}$ is the stochastic depletion function of the subpath $[c \ldots v]$.

As for $\mathbb{E}$-feasible routes, we maintain a global priority queue Q_G storing the travel time distributions from s, and queues $L_u(v)$ and $L_s(v)$ to store the unsettled and settled labels at vertex v respectively. All queues use the usual stochastic ordering. On each search iteration, a label is extracted from $L_u(v)$, where v is the minimum-travel time vertex in Q_G, settled, and added to $L_s(v)$. Each label in $L_s(v)$ represents a feasible path from s to v, including the charging time at the last charging station. Each label in $L_u(v)$ represents a potentially feasible path whose feasibility is yet unverified. If a label $\ell \in L_u(v)$ is dominated by $\ell' \in L_s(v)$, we can prune the search along that path and discard ℓ, because a faster feasible path is already known. p-feasible queries have four parameters: the source vertex s, the destination vertex t, the β_s, and the given p. The search proceeds as follows:

1. *At s*: Mark s as a charging station. Add the label $\langle \boldsymbol{T}_\emptyset, {}_s\beta, s, \sigma_\emptyset \rangle$ to $L_u(s)$.
2. *At a non-charging vertex v*: Let $\ell = \langle \boldsymbol{T}_{[s\ldots v]}, {}_c\beta, c, \sigma_{[c\ldots v]} \rangle$ be the label extracted from $L_u(v)$. Since c is the last charging station encountered on the route represented by ℓ, add label $\langle \boldsymbol{T}_{[s\ldots v]}, \sigma_{[c\ldots v]}({}_c\beta, p), c, \sigma_{[c\ldots v]} \rangle$ to $L_u(v)$ and update the travel times for v in Q_G.
3. *At a charging vertex v*: Let $\ell = \langle \boldsymbol{T}_{[s\ldots v]}, {}_c\beta, c, \sigma_{[c\ldots v]} \rangle$ be the label extracted from $L_u(v)$. Let t_c be the charging time at the last charging station c, so $\beta_c = \Phi_c({}_c\beta, t_c)$. As with $\mathbb{E}$-feasible routes, the charging times corresponding to breakpoints of $\Phi_c(\cdot)$ suffice to make the required trade-off between charging and travel times. Let τ represent some value for travel time from s to v, and compute

$$b'_\ell(t_c, \tau, p) := \begin{cases} \sigma_{[c\ldots v]}(\beta_c, p) & \text{if } t_c > 0 \text{ and } \boldsymbol{T}_{[s\ldots v]}(\tau) > 0 \\ -\infty & \text{otherwise} \end{cases}$$

Since $\Phi_c(\cdot)$ is piecewise linear, its breakpoints induce breakpoints for b_ℓ. Moreover, p is already known at query time, so for a given value of τ, we only need to create one label per breakpoint of b'_ℓ [8]. For a fixed τ and each breakpoint $B = (t_B, \mathrm{SoC}_B)$ of b'_ℓ, we add to $L_u(v)$ the label $\langle t_B, \mathrm{SoC}_B, v, \boldsymbol{T}_\emptyset \rangle$, and update the travel times to v in Q_G.
As with $\mathbb{E}$-feasible routes, τ can take an infinite number of values, but we use histograms to represent $\boldsymbol{T}_e$, and we need to generate only one set of breakpoints per histogram bin. Lastly, we verify p-feasibility of path $[s \ldots v]$, by maintaining the running product of the non-stranding probabilities of all legs over this path. If this product falls below p, the path $[s \ldots v]$ is no longer p-feasible. The search is pruned and labels for v are dropped.

4. *At the destination t*: End search, backtrack using parent pointers to extract an s-t path. For a given p, a label ℓ dominates another label ℓ' if $b'_\ell(t,\tau,p) \geq b'_\ell(t,\tau,p)$ for all $t > 0$ and $\tau > 0$. If the search terminates only when Q_G is empty, the resulting s-t paths are the p-feasible Non-Dominated Paths. For Probabilistic Budget queries, we end the search only when it reaches far enough for the probability of reaching t within the time budget d is 0.

6 Stochastic Contraction Hierarchies

For deterministic queries, Contraction Hierarchies (CHs) [21] are widely used for speed up. Graph vertices are ranked, and *contracted* in ranked order. If u-v-w is a shortest path from u to w, vertex v is contracted by adding an edge u-w, and removing v from the graph. Such shortcuts significantly speed up the query-time Dijkstra search. The vertex ranks and the edge-weight hierarchy significantly affect preprocessing and query times [6]. A multicriteria CH variant is used in [55] for constrained shortest paths with positive weights. The CHArge algorithm [9, 8] combines a partial multicriteria CH with A* search. It contracts most graph vertices, creating a partial multicriteria CH but keeps an uncontracted *core* with charging stations. A* search using potential functions is used in the core to find routes at query time.

CHs have also been applied recently to stochastic route planning [42, 47]. However, we are interested in finding *feasible* routes that satisfy the energy bounds on EVs. Our queries are stochastic, and in fact doubly so. Travel time is always stochastic, and energy depletion is also stochastic for p-feasible queries. The stochastic dominance criterion is known to be too restrictive in practice [61], so it is hard to find dominating paths for most shortest paths in the network. Since the added shortcuts in CHs must not violate correctness, we can only avoid adding a shortcut covering a shortest path P only if we can find another witness path that dominates P [21, 55].

We solve this problem by relaxing our definition of dominance as follows. For distributions $\boldsymbol{T}_P$ and $\boldsymbol{D}_P$, we use the restricted-dominance criterion of [10], which checks if the CDF of one distribution is greater than that of the other within a fixed interval I, which we set to two standard deviations on each side of $\mathbb{E}[\boldsymbol{T}_P]$ or $\mathbb{E}[\boldsymbol{D}_P]$. For search labels, we use a definition of ϵ-dominance similar to that of [5, 45]. We say that a label ℓ_1 dominates another label ℓ_2 if all breakpoints of b_{ℓ_1} or b'_{ℓ_1} have SoC_B values within ϵ of b_{ℓ_2} or b'_{ℓ_2}. We set $\epsilon = 2\%$ of battery capacity in our experiments.

7 Experiments

Our algorithms were implemented in Rust 1.60.0-nightly with full optimizations and run on an Intel core i5-8600K processor with 3.6GHz base clock, 192KB of L1, 1.5 MB of L2, and 9 MB of L3 cache and equipped with 64GB of dual-channel 3200MHz DDR4 RAM.

7.1 Preparing a realistic routing instance

We extracted traffic speeds from Mapbox Traffic Data[1] for Tile 0230123,[2] between 15$^{\text{th}}$ July and 30$^{\text{th}}$ November, 2019. Tile 0230123 covers Los Angeles county between Long Beach and Oxnard, and yielded a graph with 559,271 vertices and 1,058,450 edges. The dataset contained speed updates for an edge subset at 5-minute intervals, which we aggregated

[1] `https://www.mapbox.com/traffic-data`
[2] `https://labs.mapbox.com/what-the-tile`

into weekday and weekend speed histograms. We discarded the weekend histograms due to sparsity, and used only the weekday speeds for our experiments. We added latitudes and longitudes for each vertex from the OSM dataset taken from GeoFabrik,[3] contracted the degree-2 vertices, and extracted the largest connected component. This step resulted in the final routing graph of 244,728 vertices and 453,942 edges.

We added elevation data from the NASADEM dataset [34] at 30M resolution to each vertex, using bilinear interpolation to estimate elevations at vertex locations. Lastly, we obtained charging stations from the Alternative Fuels Data Center,[4] marking the vertex closest to each charging station as the charging vertex. The charging function Φ_c on each vertex c was linear, and either (1) a *slow*, charging to 100% in 100 minutes, or (2) *fast*, charging up to 80% in 30 minutes, and up to 100% in 60 minutes. We randomly assigned the slow charging function to 70% of charging stations, the fast charging function to the rest.

Energy consumption parameters for ε_e on all edges e were derived using the vertex elevations and the values a_e, b_e, c_e, d_e used for Nissan Leaf 2013 in [16]. To force the search to require charging en route for feasibility, we assumed that the EV had a 12 kWh battery.

Choice of edge weight representation. Histograms capture arbitrary $\boldsymbol{T}_e$ and $\boldsymbol{D}_e$ distributions, but take more space. Functions may be less faithful to real-world distributions, but are compact and may lead to faster queries in some cases [47]. We used histograms to represent the travel time and energy consumption distributions on edges since our dataset had enough data for most edges. This allows us to represent arbitrary distributions while keeping the implementation simple.

Applying Contraction Hierarchies. Building a full CH by contracting all vertices of the graph can be prohibitively expensive due to the high cost of contracting the highest ranked vertices [9]. So, we build a only partial CH by contracting 97% of the vertices, keeping an uncontracted *core* containing all the charging stations on the network. Queries are run in three stages—from s to a vertex in the core restricted to using only (upward) edges from lower to higher ranked vertices, backward search from t to a vertex in the core using downward edges, and a simple bidirectional search within the core of the network.

7.2 Results

Using stochastic edge weights raises many challenges that do not arise for deterministic weights. Two obvious issues are maintaining route feasibility, and aggregating edge distributions $\boldsymbol{T}_e$ and $\boldsymbol{D}_e$ into path distributions $\boldsymbol{T}_P$ or $\boldsymbol{D}_P$, which requires expensive convolutions. Several other issues also arise, two of which we will discuss.

Number of histogram bins. The time and energy value ranges in the path distributions $\boldsymbol{T}_P, \boldsymbol{D}_P$ increases linearly with the number of edges in P, so more histogram bins are needed to maintain accuracy. As in the deterministic case, the Dijkstra search labels track the travel time-charging time tradeoff. The labels represent histograms, so the label sizes increase with the number bins used for $\boldsymbol{T}_P$ and $\boldsymbol{D}_P$. Labels become progressively larger for longer routes, raising the cost of all operations on the distributions, (convolution, dominance checks, etc.).

[3] `https://download.geofabrik.de/north-america/us/california/socal.html`
[4] `https://afdc.energy.gov/fuels/electricity_locations.html`

At charging stations, moreover, we must create a set of breakpoints per bin of the energy depletion histogram. More breakpoints are created for charging stations further along the route, increasing costs and making label dominance checks labels more difficult.

We also note that the CH shortcuts represent longer routes, whose histograms have more bins than the original graph edges. Shortcut edges are hence more expensive to handle than original graph edges, decreasing the utility of shortcuts in speeding up route planning queries.

Ensuring stochastic feasibility. Standard probabilistic budget routes use a single criterion, such as travel time [42, 47]. In contrast, our queries must handle search with two criteria to maintain feasibility. Further, the number of breakpoints in the charging functions along a route determines the number of labels generated.

For deterministic edge weights, path costs are just sums of edge costs, so routing takes just microseconds even on continent-sized road networks [4]. Routing with stochastic edge weights is far slower, since the convolutions needed to get path costs are very expensive. Prior work [42, 47] deals only with stochasticity in time, ignoring energy feasibility, but we consider both aspects. Our methods take tens of seconds, which is comparable to these prior methods. In preliminary experiments, our use of stochastic, multicriteria CH yielded a 2–2.4 factor speedup over queries not using CH. In deterministic settings, similar methods have been reported to achieve speedups of two to three orders of magnitude [21]. This lower gain can be attributed to the weaker "hierarchy" with stochastic edge weights, causing far more shortcuts to be added to the original graph. This forces the Dijkstra search to scan many more edges on settling each vertex.

Table 2 Single-criterion probabilistic budget routing queries [47] vs. our $\mathbb{E}$-feasible and p-feasible queries on the Tile 0230123 graph. Query times (seconds) are averages over 100 random vertex pairs. The EV is a Nissan Leaf 2013 with 12 kWh battery and 50% starting SoC.

d	**Feasibility Ignored [47]**	**$\mathbb{E}$-feasible Routes**	**p-feasible Routes**		
			$p = 0.8$	$p = 0.85$	$p = 0.9$
5 min.	6.192	10.662	12.993	11.99	10.31
15 min.	19.999	24.711	38.71	38.9	36.2
25 min.	45.384	38.123	75.05	73.8	71.34

Table 2 quantifies the overhead of maintaining feasibility of routes in stochastic settings, and compares the query times for single-criteria probabilistic budget routes (time only, feasibility ignored) with those of our two-criteria feasible probabilistic budget routes. Single-criteria routing is fastest, followed by $\mathbb{E}$-feasible routing, and p-feasible routing. The anomaly for $d = 25$ minutes can be understood as follows. Multicriteria search must explore a larger set of routes from the source than single-criteria queries, because it needs to return the pareto frontier of routes, rather than a single route. The $\mathbb{E}$-feasible and p-feasible queries must also carry and update per-vertex labels, and maintain more information in each label to capture the travel time-charging time tradeoffs. However, we use the restricted dominance criterion for $\mathbb{E}$-feasible and p-feasible routes but not for the single-criteria routes, making the cost per convolution is slightly lower for the two feasible-path queries. This suffices to make $\mathbb{E}$-feasible routing slightly faster for longer routes than even single-criterion queries.

Table 3 compares $\mathbb{E}$-feasible and p-feasible queries, for longer deadlines. $\mathbb{E}$-feasible queries are generally faster than p-feasible queries because they must convolve only $\boldsymbol{T}_P$, but p-feasible queries convolve both $\boldsymbol{T}_P$ and $\boldsymbol{D}_P$. p-feasible queries with higher p thresholds tend to run slightly faster, as they can prune the search quicker than searches run with lower p.

Table 3 $\mathbb{E}$-feasible and p-feasible query performance on the Tile 0230123 graph, with real-world charging station and elevation data. Query times (seconds) are over 500 random vertex pairs. EV used is a Nissan Leaf 2013 fitted with a 12 kWh battery and 50% starting SoC.

Query Type	Feasibility Threshold	Time Budget (d)			
		10 min.	**20 min.**	**30 min.**	**40 min.**
$\mathbb{E}$-feasible	—	18.01	34.975	48.662	72.198
p-feasible	$p = 0.8$	33.1	52.895	81.94	91.04
	$p = 0.85$	27.1	49.58	80.35	98.312
	$p = 0.9$	26.43	47.901	78.419	96.51

Table 4 Average Jaccard Index for 500 random $\mathbb{E}$-feasible and p-feasible routes, with $p = 0.85$. The index is 0 when the routes are edge-disjoint, and 1 when they are identical.

Queries Compared	d	Avg. Jaccard Index
$\mathbb{E}$-feasible and p-feasible, for $p = 0.85$	10 min.	0.73
	20 min.	0.74
	30 min.	0.87
	40 min.	0.94

Table 4 shows how similar the $\mathbb{E}$-feasible and p-feasible routes are, using the average Jaccard Similarity between the set edges of a route chosen by each of them. The Jaccard similarity for two routes P_1 and P_2 is the number of edges common to both divided by the number of edges in their union. That is,

$$J(P_1, P_2) = \frac{|\{e \in P_1\} \cap \{e \in P_2\}|}{|\{e \in P_1\} \cup \{e \in P_2\}|}$$

The Jaccard index clearly increases with the time budget, so the $\mathbb{E}$-feasible and p-feasible routes are more similar when the routes are longer. This is because longer routes require more convolutions, making $\boldsymbol{D}_P$ closer to the Gaussian, which is more concentrated near its mean. In such cases, the pruning of edges forced by the feasibility criterion brings the set of edges of $\mathbb{E}$-feasible routes closer to the set of edges for p-feasible routing. For shorter routes, however, the difference between the two types of queries is higher. Hence, if stronger feasibility guarantees are desired for shorter routes, p-feasible queries may be better.

8 Conclusion and Future Work

EV routing methods usually model the problem as a generalized constrained shortest-path problem, with deterministic travel times and energy consumptions. This is unrealistic since these are really stochastic parameters. Current stochastic route planning methods either fail to ensure that routes are energy-feasible, or when they do, have not been shown to scale well to large graphs. In this work, we address this shortcoming by making travel time and energy consumption stochastic, and requiring paths to be energy-feasible. We defined two energy-feasibility criteria, namely, $\mathbb{E}$-feasibility and p-feasibility. We showed how to generalize the standard Charging Function Propagation algorithm of [8, 9] to accept

stochastic edge weights, while allowing recharging stations. We also applied a multicriteria variant of stochastic Contraction Hierarchies to speed up our queries, using the restricted stochastic dominance criterion of [10] and the ϵ-dominance among labels. We demonstrated that our techniques were feasible in the real world by running experiments on a realistic routing instance, using real-world travel speeds in the Los Angeles area collected over four and a half months. The similarity between $\mathbb{E}$-feasible and p-feasible routes indicates the potential applicability of a tiered-hierarchy style approach [47] to help speed up stochastic feasible routing queries even further, and could be an interesting avenue for further work.

References

1 Niklas Akerblöm, Yuxin Chen, and Morteza Haghir Chehreghani. An online learning framework for Energy-Efficient navigation of electric vehicles. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20)*, pages 2051–2057, 2020.

2 Yazan Al-Wreikat, Clara Serrano, and José Ricardo Sodré. Driving behaviour and trip condition effects on the energy consumption of an electric vehicle under real-world driving. *Appl. Energy*, 297:117096, September 2021.

3 Andreas Artmeier, Julian Haselmayr, Martin Leucker, and Martin Sachenbacher. The shortest path problem revisited: Optimal routing for electric vehicles. In *KI 2010: Advances in Artificial Intelligence*, Lecture Notes in Computer Science, pages 309–316. Springer, Berlin, Heidelberg, September 2010.

4 Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route planning in transportation networks. In *Algorithm Engineering*, Lecture Notes in Computer Science, pages 19–80. Springer, Cham, 2016.

5 Lucas S Batista, Felipe Campelo, Frederico G Guimarães, and Jaime A Ramírez. A comparison of dominance criteria in many-objective optimization problems. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 2359–2366, June 2011.

6 Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in contraction hierarchies. *Theor. Comput. Sci.*, 645:112–127, September 2016.

7 Moritz Baum. *Engineering Route Planning Algorithms for Battery Electric Vehicles*. PhD thesis, Karlsruhe Institute of Technology, 2018.

8 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 SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 44. ACM, November 2015.

9 Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. Shortest feasible paths with charging stops for battery electric vehicles. *Transportation Science*, 53(6):1627–1655, November 2019.

10 Roger L Berger. A nonparametric, intersection-union test for stochastic order. Technical report, North Carolina State University. Dept. of Statistics, 1986.

11 Dimitri P Bertsekas and John N Tsitsiklis. An analysis of stochastic shortest path problems. *Math. Oper. Res.*, 16(3):580–595, August 1991.

12 Anthony Chen and Zhaowang Ji. Path finding under uncertainty. *J. Adv. Transp.*, 39(1):19–37, September 2005.

13 Xiao-Wei Chen, Bi Yu Chen, William H K Lam, Mei Lam Tam, and Wei Ma. A bi-objective reliable path-finding algorithm for battery electric vehicle routing. *Expert Syst. Appl.*, 182:115228, November 2021.

14 Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal route planning for electric vehicles in large networks. In *AAAI*, 2011.

15 Y Y Fan, R E Kalaba, and J E Moore. Arriving on time. *J. Optim. Theory Appl.*, 127(3):497–513, December 2005.

16 Chiara Fiori, Kyoungho Ahn, and Hesham A Rakha. Power-based electric vehicle energy consumption model: Model development and validation. *Appl. Energy*, 168:257–268, April 2016.

17 Chiara Fiori, Vittorio Marzano, Vincenzo Punzo, and Marcello Montanino. Energy consumption modeling in presence of uncertainty. *IEEE Trans. Intell. Transp. Syst.*, pages 1–12, 2020.

18 Matthew William Fontana. *Optimal routes for electric vehicles facing uncertainty, congestion, and energy constraints*. PhD thesis, Massachusetts Institute of Technology, 2013.

19 Mogens Fosgerau. The valuation of travel time variability. Technical report, International Transport Forum, 2016.

20 H Frank. Shortest paths in probabilistic graphs. *Oper. Res.*, 17(4):583–599, 1969.

21 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms*, pages 319–333. Springer, Berlin, Heidelberg, May 2008.

22 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*, pages 193–202. ACM, November 2014.

23 Ming Hua and Jian Pei. Probabilistic path queries in road networks: traffic uncertainty aware path selection. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 347–358, New York, NY, USA, March 2010. Association for Computing Machinery.

24 Gerhard Huber, Klaus Bogenberger, and Hans van Lint. Optimization of charging strategies for battery electric vehicles under uncertainty. *IEEE Trans. Intell. Transp. Syst.*, pages 1–17, 2020.

25 Ehsan Jafari and Stephen D Boyles. Multicriteria stochastic shortest path problem for electric vehicles. *Networks Spat. Econ.*, 17(3):1043–1070, September 2017.

26 H C Joksch. The shortest route problem with constraints. *J. Math. Anal. Appl.*, 14(2):191–197, May 1966.

27 Moritz Kobitzsch, Samitha Samaranayake, and Dennis Schieferdecker. Pruning techniques for the stochastic on-time arrival problem- an experimental study. *arXiv*, July 2014. `arXiv:1407.8295`.

28 Sejoon Lim, Christian Sommer, Evdokia Nikolova, and Daniela Rus. Practical route planning under delay uncertainty: Stochastic shortest path queries. In *Robotics: Science and Systems*, volume 8, pages 249–256. books.google.com, 2013.

29 Antonio Lima, Rade Stanojevic, Dina Papagiannaki, Pablo Rodriguez, and Marta C González. Understanding individual routing behaviour. *J. R. Soc. Interface*, 13(116), March 2016.

30 Sören Merting, Christian Schwan, and Martin Strehler. Routing of electric vehicles: Constrained shortest path problems with resource recovering nodes. In *OASIcs-OpenAccess Series in Informatics*, volume 48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2015.

31 Sebastiano Milardo, Punit Rathore, Marco Amorim, Umberto Fugiglando, Paolo Santi, and Carlo Ratti. Understanding drivers' stress and interactions with vehicle systems through naturalistic data analysis. *IEEE Trans. Intell. Transp. Syst.*, pages 1–12, 2021.

32 Elise D Miller-Hooks and Hani S Mahmassani. Least expected time paths in stochastic, Time-Varying transportation networks. *Transportation Science*, 34(2):198–215, May 2000.

33 Elise Deborah Miller-Hooks. *Optimal routing in time-varying, stochastic networks: Algorithms and implementations*. PhD thesis, The University of Texas at Austin, 1997.

34 J P L Nasa. NASADEM merged DEM global 1 arc second V001 [dataset], 2020. Accessed: 2021-6-9. `doi:10.5067/MEaSUREs/NASADEM/NASADEM_HGT.001`.

35 Yu (marco) Nie and Xing Wu. Shortest path problem considering on-time arrival probability. *Trans. Res. Part B: Methodol.*, 43(6):597–613, July 2009.

36 Patrick Niklaus. A unified framework for electric vehicle routing. Master's thesis, Karlsruhe Institute of Technology, 2017.

37 Mehrdad Niknami and Samitha Samaranayake. Tractable pathfinding for the stochastic On-Time arrival problem. In *Experimental Algorithms*, pages 231–245. Springer International Publishing, 2016.

38 Evdokia Nikolova. Approximation algorithms for reliable stochastic combinatorial optimization. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 338–351. Springer Berlin Heidelberg, 2010.

39 Evdokia Nikolova, Matthew Brand, and David R Karger. Optimal route planning under uncertainty. *ICAPS*, 2006.

40 Evdokia Nikolova, Jonathan A Kelner, Matthew Brand, and Michael Mitzenmacher. Stochastic shortest paths via quasi-convex maximization. In *Algorithms – ESA 2006*, pages 552–563. Springer Berlin Heidelberg, 2006.

41 Axel Parmentier and Frédéric Meunier. Stochastic shortest paths and risk measures. *arXiv*, August 2014. `arXiv:1408.0272`.

42 Simon Aagaard Pedersen, Bin Yang, and Christian S Jensen. Fast stochastic routing under time-varying uncertainty. *VLDB J.*, October 2019.

43 Simon Aagaard Pedersen, Bin Yang, and Christian S Jensen. Anytime stochastic routing with hybrid learning. *Proceedings VLDB Endowment*, 13(9):1555–1567, May 2020.

44 Simon Aagaard Pedersen, Bin Yang, and Christian S Jensen. A hybrid learning approach to stochastic routing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1910–1913, April 2020.

45 Payas Rajan, Moritz Baum, Michael Wegner, Tobias Zündorf, Christian J West, Dennis Schieferdecker, and Daniel Delling. Robustness generalizations of the shortest feasible path problem for electric vehicles. In Matthias And Federico, Müller-Hannemann, editor, *Proceedings of 21st Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2021)*, Open Access Series in Informatics (OASIcs), pages 11:1–11:18, Dagstuhl, Germany, September 2021. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

46 Payas Rajan and Chinya V Ravishankar. The phase abstraction for estimating energy consumption and travel times for electric vehicle route planning. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '19, pages 556–559, New York, NY, USA, 2019. ACM.

47 Payas Rajan and Chinya V Ravishankar. Tiering in contraction and edge hierarchies for stochastic route planning. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '21, pages 616–625, New York, NY, USA, November 2021. Association for Computing Machinery.

48 Guillaume Sabran, Samitha Samaranayake, and Alexandre Bayen. Precomputation techniques for the stochastic on-time arrival problem. In *2014 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, Proceedings, pages 138–146. Society for Industrial and Applied Mathematics, December 2013.

49 M Sachenbacher, M Leucker, A Artmeier, and J Haselmayr. Efficient Energy-Optimal routing for electric vehicles. *AAAI*, 2011.

50 Moshe Shaked and J George Shanthikumar. *Stochastic Orders*. Springer New York, October 2006.

51 Liang Shen, Hu Shao, Ting Wu, William H K Lam, and Emily C Zhu. An energy-efficient reliable path finding algorithm for stochastic road networks with electric vehicles. *Transp. Res. Part C: Emerg. Technol.*, 102:450–473, May 2019.

52 Kenneth A Small, Clifford Winston, and Jia Yan. Uncovering the distribution of motorists' preferences for travel time and reliability. *Econometrica*, 73(4):1367–1382, July 2005.

53 Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys (CSUR)*, 46(4):45, April 2014.

54 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, New York, NY, USA, 2012. ACM.

55 Sabine Storandt. Route planning for bicycles – exact constrained shortest paths made practical via contraction hierarchy. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

56 Sabine Storandt and Stefan Funke. Cruising with a Battery-Powered vehicle and not getting stranded. In *AAAI*, volume 3, page 46, 2012.

57 Martin Strehler, Sören Merting, and Christian Schwan. Energy-efficient shortest routes for electric and hybrid vehicles. *Trans. Res. Part B: Methodol.*, 103(Supplement C):111–135, September 2017.

58 Christoph Witzgall and Alan J Goldman. Most profitable routing before maintenance. In *OPERATIONS RESEARCH*, page B82. INST OPERATIONS RESEARCH MANAGEMENT SCIENCES 901 ELKRIDGE LANDING RD, STE 400, LINTHICUM HTS, MD, USA, 1965.

59 Bin Yang, Chenjuan Guo, Christian S Jensen, Manohar Kaul, and Shuo Shang. Stochastic skyline route planning under time-varying uncertainty. In *2014 IEEE 30th International Conference on Data Engineering*, pages 136–147, March 2014.

60 Shanjiang Zhu and David Levinson. Do people use the shortest path? an empirical test of wardrop's first principle. *PLoS One*, 10(8):e0134322, August 2015.

61 Weiwei Zhuang, Yadong Li, and Guoxin Qiu. Statistical inference for a relaxation index of stochastic dominance under density ratio model. *J. Appl. Stat.*, pages 1–19, August 2021.

RLBWT Tricks

Nathaniel K. Brown
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Travis Gagie
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Massimiliano Rossi
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA

Abstract

Until recently, most experts would probably have agreed we cannot backwards-step in constant time with a run-length compressed Burrows-Wheeler Transform (RLBWT), since doing so relies on rank queries on sparse bitvectors and those inherit lower bounds from predecessor queries. At ICALP '21, however, Nishimoto and Tabei described a new, simple and constant-time implementation. For a permutation π, it stores an $O(r)$-space table – where r is the number of positions i where either $i = 0$ or $\pi(i+1) \neq \pi(i) + 1$ – that enables the computation of successive values of $\pi(i)$ by table look-ups and linear scans. Nishimoto and Tabei showed how to increase the number of rows in the table to bound the length of the linear scans such that the query time for computing $\pi(i)$ is constant while maintaining $O(r)$-space.

In this paper we refine Nishimoto and Tabei's approach, including a time-space tradeoff, and experimentally evaluate different implementations demonstrating the practicality of part of their result. We show that even without adding rows to the table, in practice we almost always scan only a few entries during queries. We propose a decomposition scheme of the permutation π corresponding to the LF-mapping that allows an improved compression of the data structure, while limiting the query time. We tested our implementation on real-world genomic datasets and found that without compression of the table, backward-stepping is drastically faster than with sparse bitvector implementations but, unfortunately, also uses drastically more space. After compression, backward-stepping is competitive both in time and space with the best existing implementations.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Compressed String Indexes, Repetitive Text Collections, Burrows-Wheeler Transform

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.16

Related Version *Full Version*: https://arxiv.org/abs/2112.04271

Supplementary Material *Software (Source Code)*: https://github.com/drnatebrown/r-index-f
archived at swh:1:dir:62d807654bc1f9a8781427668e68212f8d99a5b6

Funding This work was funded by NIH R01AI141810 and R01HG011392, NSERC Discovery Grant RGPIN-07185-2020, and NSF IIBR 2029552 and IIS 1618814.

Acknowledgements Many thanks to Omar Ahmed, Christina Boucher and Ben Langmead for discussions and assistance during our research, and to the anonymous reviewers for their insightful feedback.

1 Introduction

The FM-index [5] is the basis for key tools in computational genomics, such as the popular short-read aligners BWA [12] and Bowtie [11], and is probably now the most important application of the Burrows-Wheeler Transform (BWT) [4]. As genomic databases have grown

and researchers and clinicians have realized that using only one or a few reference genomes biases their results and diagnoses, interest in computational pan-genomics has surged and versions of the FM-index based on the run-length compressed BWT (RLBWT) [13] have been developed that can index thousands of genomes in reasonable space [6, 10, 16]. Those versions have all relied heavily on compressed sparse bitvectors, however, which are inherently slower than the bitvectors used in regular FM-indexes (see [14] for details). Experts would probably have guessed that sparse bitvectors were an essential component for RLBWT-based pan-genomic indexes – until Nishimoto and Tabei [15] recently showed how to replace them with theoretically more efficient alternatives.

In particular, Nishimoto and Tabei's result gives an approach which achieves constant time LF-mapping in $O(r)$-space [15]. Speeding up LF can reduce the time for basic queries over the RLBWT and other applications. For example, Ahmed et al.'s SPUMONI [1] tool allows rapid targeted nanopore sequencing over compressed pan-genome indexes using approximate matching statistics; "nontarget" DNA molecules are ejected from the sequencer with an emphasis on speed. Their method depends on LF-mapping to extend matches, and otherwise "jumping" forwards or backwards in the BWT based on threshold computation. Thresholds over the BWT is a rather new approach, introduced by Bannai et al. in 2020 [2], suggesting further improvements may be developed; however, avoiding the lower bounds inherited from predecessor queries from rank on sparse bitvectors[1] is a more surprising result. For tools that heavily depend on LF, experiments showing practical results provide an opportunity for speed improvements that otherwise would not have been expected to be attainable.

In this paper we focus on the first part of Nishimoto and Tabei's result: we demonstrate experimentally that we can reduce the time for basic queries on an RLBWT by replacing queries on sparse bitvectors by table lookups, sequential scans, and queries on relatively short uncompressed bitvectors. We implement LF-mapping over the RLBWT using table lookup; preliminary results showed this could be made practical even without theoretical worst case time guarantees. Although their result also applies to the ϕ function over the RLBWT [15], we focus on LF since it allows backward-stepping (performed before locating, which requires ϕ) and its seems more compressible for LF; we leverage the unique structure of LF to partition columns of the table into non-decreasing subsequences.

With this motivation, we present various techniques and optimizations towards a practical implementation. To demonstrate its practicality, we use real-world genomic datasets to perform count queries using haplotypes of chromosome 19 and SARS-CoV2 genomes. We find that our implementations are competitive in time/space with the best existing methods: in the average case without row insertions, and exploring a run splitting approach to loosely bound sequential scanning in the worst case. Further analysis shows in practice, sequential scans are quite rare, but can become more common as n/r grows, motivating our run splitting and further approaches.

The rest of this paper is laid out as follows: in Section 2 we present the two parts of Nishimoto and Tabei's result and explain how they relate to RLBWT-based pan-genomic indexes; Section 3 describes methods used to make the result practical for implementation; in Section 4 we present our experimental results; and in Section 5 we analyse its practicality and summarize findings.

[1] Conventionally, LF-mapping in runs bounded space relies on rank queries over sparse bitvectors.

2 Nishimoto and Tabei's Result

Suppose we want to compactly store a permutation π on $\{0, \ldots, n-1\}$ such that we can evaluate $\pi(i)$ quickly when given i. If π is chosen arbitrarily then $\Theta(n)$ space is necessary to store it in the worst case, and sufficient to allow constant-time evaluation. If the sequence $\pi(0), \pi(1), \pi(2), \ldots, \pi(n-1)$ consists of a relatively small number b of unbroken incrementing subsequences, however – meaning $\pi(i+1) = \pi(i) + 1$ whenever $\pi(i)$ and $\pi(i+1)$ are in the same subsequence – then we can store π in $O(b)$ space and evaluate it in $O(\log \log n)$ time. To do this, we simply store in an $O(b)$-space predecessor data structure with $O(\log \log n)$ query time – such as a compressed sparse bitvector – each value i such that $\pi(i)$ is the head of one of those subsequences, with $\pi(i)$ as satellite data; we evaluate any $\pi(i)$ in $O(\log \log n)$ time as

$$\pi(i) = \pi(\text{pred}(i)) + i - \text{pred}(i) \,.$$

Nishimoto and Tabei first proposed a simple alternative $O(b)$-space representation:[2] we store a sorted table in which, for each subsequence head p, there is quadruple: p; the length of the subsequence starting with p; $\pi(p)$; and the index of the subsequence containing $\pi(p)$.

If we know the index of the subsequence containing i then we can look up the quadruple for that subsequence and find both its head p and $\pi(p)$, then compute $\pi(i) = \pi(p) + i - p$ in constant time. If we want to compute $\pi^2(i)$ the same way, however, we should compute the index of the subsequence containing $\pi(i)$, since $\pi(i)$ may be in a later subsequence than $\pi(p)$. To do this, we look up the quadruple for the subsequence containing $\pi(p)$ (which takes constant time since we have its index) and find its head and length, from which we can tell if $\pi(i)$ is in the subsequence. If it is not, we continue reading and checking the quadruples for the following subsequences (which takes constant time for each one, since they are next in the table) until we find the one that does contain $\pi(i)$.

Sequentially scanning the table to find the quadruple for the subsequence containing $\pi(i)$ could take $\Omega(b)$ time in the worst case, so Nishimoto and Tabei then proved the following result, which implies we can artificially divide some of the subsequences before building the table, such that all the sequential scans are short. We still find their proof surprising, so we have included a summary of it below which introduces our parameter d. This refinement of the original theorem allows for a time/space tradeoff.

▶ **Theorem 1** (Nishimoto and Tabei [15]). *Let π be a permutation on $\{0, \ldots, n-1\}$,*

$$P = \{0\} \cup \{i \ : \ 0 < i \leq n-1, \pi(i) \neq \pi(i-1) + 1\} \,,$$

and $Q = \{\pi(i) \ : \ i \in P\}$. For any integer $d \geq 2$, we can construct P' with $P \subseteq P' \subseteq \{0, \ldots, n-1\}$ and $Q' = \{\pi(i) \ : \ i \in P'\}$ such that

- *if $q, q' \in Q'$ and q is the predecessor of q' in Q', then $|[q, q') \cap P'| < 2d$,*
- *$|P'| \leq \frac{d|P|}{d-1}$.*

Proof. We start by setting $P_0 = P$ and $Q_0 = Q$. Suppose at some point we have P_i and $Q_i = \{\pi(i) \ : \ i \in P_i\}$. If there do not exist $q, q' \in Q_i$ such that q is the predecessor of q' in Q_i and $|[q, q') \cap P_i| \geq 2d$, then we stop and return $P' = P_i$ and $Q' = Q_i$; otherwise, we choose some such q and q'.

[2] We may have taken some artistic license with their format.

We choose the $(d+1)$st largest element p in $[q, q') \cap P_i$ and set $P_{i+1} = P_i \cup \{\pi^{-1}(p)\}$ and $Q_{i+1} = Q_i \cup \{p\} = \{\pi(i) \ : \ i \in P_{i+1}\}$. Since $q < p < q'$ we have $p \notin Q_i$ and so $\pi^{-1}(p) \notin P_i$. Therefore, $|P_{i+1}| = |P_i| + 1$ and so, by induction, $|P_{i+1}| = |P| + i + 1$.

Let E_i be the set of intervals $[u, u')$ such that $u, u' \in Q_i$ and u is the predecessor of u' in Q_i and $|[u, u') \cap P_i| \geq d$, and let E_{i+1} be the set of intervals $[u, u')$ such that $u, u' \in Q_{i+1}$ and u is the predecessor of u' in Q_{i+1} and $|[u, u') \cap P_{i+1}| \geq d$. Since $E_{i+1} = (E_i \backslash \{[q, q')\}) \cup \{[q, p), [p, q')\}$, we have $|E_{i+1}| = |E_i| + 1$ and, by induction, $|E_{i+1}| \geq i + 1$.

Since the intervals in E_{i+1} are disjoint and each contain at least d elements of P_{i+1}, we have $|P_{i+1}| \geq d|E_{i+1}| \geq d(i+1)$. Since $|P_{i+1}| = |P| + i + 1$ and $|P_{i+1}| \geq d(i+1)$, we have $|P| + i + 1 \geq d(i+1)$ and thus $i + 1 \leq \frac{|P|}{d-1}$ and $|P_{i+1}| = |P| + i + 1 \leq \frac{d|P|}{d-1}$. It follows that we find P' and Q' after at most $\frac{|P|}{d-1}$ steps. ◀

To discuss how Theorem 1 relates to RLBWTs, we first recall the definitions of the suffix array (SA), the BWT, the LF mapping and ϕ for a text $T[0..n-1]$:

- $\mathrm{SA}[i]$ is the starting position of the lexicographically ith suffix of T;
- $\mathrm{BWT}[i]$ is the character immediately preceding that suffix;
- $\mathrm{LF}(i)$ is the position of $\mathrm{SA}[i] - 1$ in SA;
- $\phi(i)$ is the value that precedes i in SA.

Let T be defined over an alphabet Σ of size σ. For convenience we assume T ends with a special symbol $T[n-1] = \$$ that occurs nowhere else, we consider strings and arrays as cyclic and we work modulo n.

It is not difficult to see that LF and ϕ (and thus also ϕ^{-1}) are permutations that can be divided into at most r of unbroken incrementing subsequences, where r is the number of runs in the BWT.[3] First, if $\mathrm{BWT}[i] = \mathrm{BWT}[i+1]$ then $\mathrm{LF}(i+1) = \mathrm{LF}(i) + 1$, so there are at most r values for which $\mathrm{LF}(i+1) \neq \mathrm{LF}(i) + 1$. Second, if $\mathrm{BWT}[i] = \mathrm{BWT}[i+1]$ so $\mathrm{LF}(i+1) = \mathrm{LF}(i) + 1$ then

$$\mathrm{SA}[\mathrm{LF}(i)] = \phi(\mathrm{SA}[\mathrm{LF}(i+1)])$$

and, as illustrated in Figure 1,

$$\phi(\mathrm{SA}[i+1]) = \mathrm{SA}[i] = \mathrm{SA}[\mathrm{LF}(i)] + 1 = \phi(\mathrm{SA}[\mathrm{LF}(i+1)]) + 1 = \phi(\mathrm{SA}[i+1] - 1) + 1$$

or, choosing $i' = \mathrm{SA}[i+1] - 1$, we have $\phi(i'+1) = \phi(i') + 1$. It follows that there are at most r values for which $\phi(i'+1) \neq \phi(i') + 1$. Nishimoto and Tabei's result therefore gives us $O(r)$-space data structures supporting LF, ϕ and ϕ^{-1} in constant time.

As a practical aside we note that, although applying Theorem 1 means we store quadruples for sub-runs in the BWT, we can store with them the indexes of the maximal runs containing them and thus, for example, store SA samples in an r-index only at the boundaries of maximal runs and not sub-runs.

The queries needed for most RLBWT-based pan-genomic indexes[4] can be implemented using LF, ϕ, ϕ^{-1} and access, rank and select queries on the string $R[0..r-1]$ in which $R[i]$ is the distinct character appearing in the ith run in BWT, which can be supported with a

[3] Realizing this about ϕ, however, led directly to Gagie, Navarro and Prezza's r-index [6].

[4] For example, for the recent pan-genomic index MONI [16], we need LF, ϕ, ϕ^{-1} and access to so-called thresholds. A threshold for a consecutive pair of runs of the same character in BWT is a position of a minimum LCP value in the interval between those runs. If we know the index of the run containing a particular character $\mathrm{BWT}[i]$ and its offset in that run, and we want to know whether it is before or after the threshold for the pair of runs of another character c bracketing $\mathrm{BWT}[i]$, then we can find in $O(\log \sigma)$ time the index of the preceding run of cs; if we have the index of the run containing the threshold and its offset in that run stored with that preceding run of cs, then we can tell immediately if $\mathrm{BWT}[i]$ is before or after the threshold.

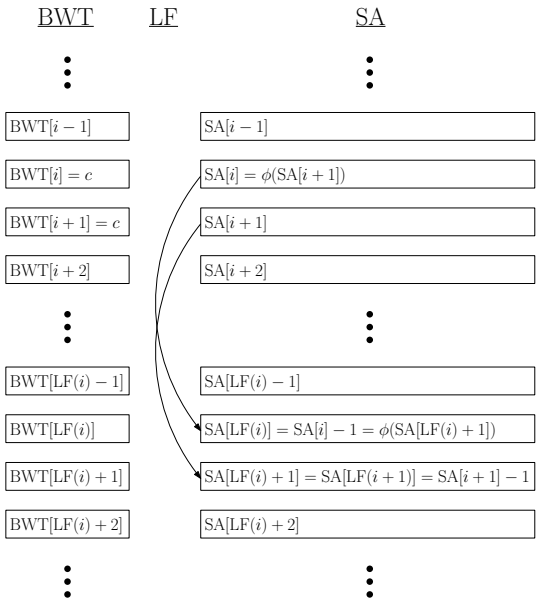


Figure 1 An illustration of why $\mathrm{BWT}[i] = \mathrm{BWT}[i+1]$ implies $\phi(\mathrm{SA}[i+1]) = \phi(\mathrm{SA}[i+1]-1)+1$.

wavelet tree on R. Of course that wavelet tree uses bitvectors, but even with uncompressed bitvectors it takes only $r \lg \sigma + o(r \log \sigma)$ bits, where σ is the size of the alphabet (usually 4 for genomics and pan-genomics), and supports those queries in $O(\log \sigma)$ time (or constant time when $\sigma = \log^{O(1)} n$).

3 Practical Approach

To provide a practical implementation of Nishimoto and Tabei's first result, we slightly modify the structure of the table. Consider the permutation to be LF(i) over the BWT, with runs being unary substrings of the BWT. In Section 2 we presented the quadruples using absolute indexes over the permutation, but we can instead perform access using the run index itself: let positions of run heads in the BWT be the array $I[0..r-1]$ storing the sorted values i such that $i = 0$ or $\mathrm{LF}(i-1) \neq \mathrm{LF}(i) - 1$. For all $k \in \{0, 1, \ldots, |I| - 1\}$ we store a triple containing: the length of the run, i.e. $I[k+1] - I[k]$, where $I[r] = n$; the index of the run containing $\mathrm{LF}(I[k])$, i.e., $\max\{j \mid I[j] \leq \mathrm{LF}(I[k])\}$; and the offset d of $\mathrm{LF}(I[k])$ in that run. Let j be a position in the k-th run, the offset of $\mathrm{LF}(j)$ and $\mathrm{LF}(I[k])$ is $I[k] - j$, hence we can find the correct run containing $\mathrm{LF}(j)$ and its offset in that run using a sequential scan as described in Section 2. With this approach, we can represent positions in the BWT as run/offset pairs and implement LF accordingly, i.e. $(k', d') \leftarrow \mathrm{LF}(k, d)$. This change removes the need for the p column of the table, with successive LF steps performed using the returned run/offset pair; access row k' with offset d' and perform $\mathrm{LF}(k', d')$.

3.1 Block Compression

For each row on the previous representation of the BWT, we store the character of the run corresponding to the row to enable support of count and inversion queries. Figure 2 shows an example of this uncompressed table. Preliminary results showed that left uncompressed, LF-mapping could be made drastically faster than a sparse bitvector implementation (seen in

Section 4 as `rle-string`) for inversion or LF queries. However, the result is also drastically larger; this formulation is not practical because it requires storing three integers and one character for each run, and to perform count operations, it requires scanning the run heads to find the preceding and following run of the character we are seeking. One first improvement is to store the array $R[0..r-1]$ in a wavelet-tree as described in Section 2, which supports rank and select queries to efficiently find the preceding and following run of a given character.

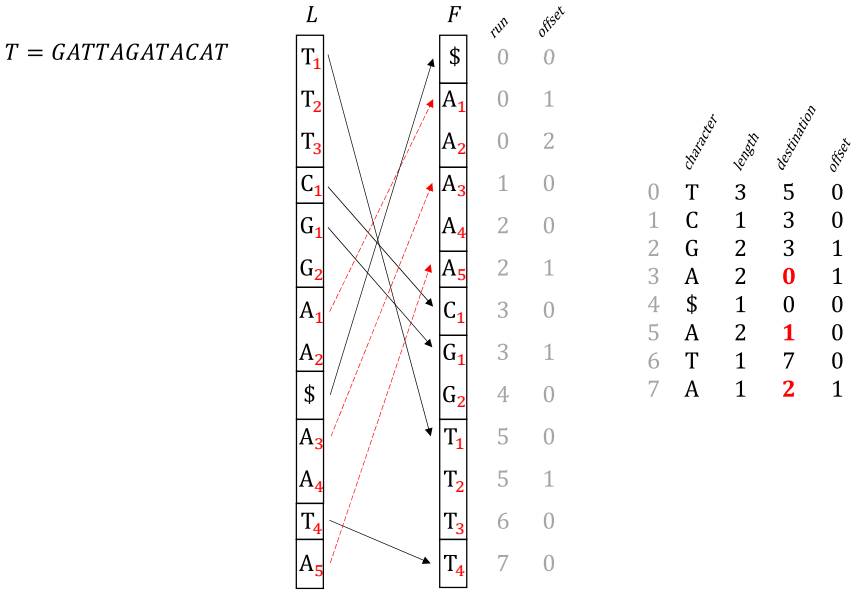


Figure 2 For an example text $T = GATTAGATACAT$, the LF mapping and subsequent uncompressed table is built (with appended terminal character \$). The run/offset columns show positions with respect to the L column used to find a mappings predecessor. Notice that highlighted stored mappings (destinations) for any run of As form a non-decreasing subsequence.

The tabular approach exploits space locality of the entries that facilitate the linear scans required by the algorithm when accessing rows sequentially; however, there is no apparent relationship which makes row-wise compression easy. To mitigate locality concerns, we partition the table into blocks of size B which are loaded in a cache friendly manner. Using a fixed B, we can easily perform modular arithmetic to map positions within the blocks. For each block we store the corresponding character of each run in a wavelet tree that allows fast rank and select queries inside the block (using uncompressed bitvectors). For each character c of the alphabet, the position of the first run of c's preceding the beginning of the block and following the end of the block is stored, allowing efficient retrieval of these characters' correct rows when they are not stored in the wavelet tree and occur in another block. For example, we may need to look to another block if some character has no occurrences in the current block, or has no occurrences before/after some position.

To improve compression inside the block, we compress the lists of lengths and offsets using directly-addressable codes (DACs, see [14]); we divide the list of run indices into σ sub-lists, each containing the indices from rows corresponding to runs of a distinct character $c \in \Sigma$. Compressing the lengths and offsets in DACs is naive compression[5] leveraging the

[5] DACs are a simple method to allow both random access alongside compression; however, more specific techniques would be preferred if these columns have exploitable properties that we could not uncover.

length of a value's bit representation while also supporting random access. For mapping destinations, it follows from LF that the mapping indices across a common character c form a non-decreasing sub-sequence [4] as highlighted in Figure 2. If we store in a block, for each of the σ sub-lists, the mapping index of the first occurrence in the list, then the rest of the list can be truncated as a difference from the base mapping. We can also choose to represent the sub-lists by partial differences; for m occurrences of a character c let $M[0..m-1]$ be such a sub-list where we explicitly store the first mapping $M[0]$, and represent the list as $D[0] = 0, D[i] = M[i] - M[i-1]$. Storing only partial differences allows us to recover the mapping using prefix sum, which we expand upon in Section 3.2 alongside an approach over absolute offsets from the base. To manoeuvre around our positional change to run indices, we also store a sparse bitvector marking sampled run head positions in the BWT, which is used after backwards-stepping to recover the absolute index from a run/offset pair.[6]

3.2 Optimizations

Compressing the mapping column as "difference lists" gives various representations of exploiting the σ non-decreasing sub-sequences:

DAC Sampling. By storing the partial differences space efficiently and sampling the absolute difference from the base, the number of random accesses needed to recover the correct value is bounded when computing the prefix sum. Implementing the approach using DACs to store the partial differences, we have a first method to retrieve mappings in compressed space while avoiding a costly traversal of the entire list. Although basic, this method is a simple choice to illustrate how we can leverage these sequences being non-decreasing.

Linear Interpolation. We perform linear interpolation between sampled offsets (as opposed to partial differences); with a sample rate s, prior sample x, next sample z, and unsampled difference y at position i. For each y, we then store its difference $\Delta = y - \epsilon$ from a weighted average defined by

$$\epsilon = x + (z - x) \cdot (i - s \cdot (\lfloor i/s \rfloor)/s)$$

into a DAC. [7] Given i and s, we lookup x and z to compute ϵ, after which we compute $(y - \epsilon) + \epsilon = y$ from our stored value $\Delta = y - \epsilon$ to recover the mapping. At worst the stored value can only be the difference between the sampled values themselves, and we expect each value to tend towards the interpolated average obtained by assuming a linear increase between samples.

Bitvector. Construct an uncompressed bitvector in which the number of 0s before the kth 1 is the offset from the first pointer (which is stored explicitly) to the kth. For example, given a sequence $M = [11, 16, 19, 21]$ and corresponding partial differences $D = [0, 5, 3, 2]$, we store the first pointer $M[0] = 11$ alongside the bitvector

10000010001001

[6] Although we introduce a sparse bitvector into our data structure, it is not used during sequential LF stepping, but rather as an "exit" or "entrance" from the table's run/offset pairs.

[7] We store a bitvector denoting the sign of the stored component, allowing us to compress unsigned integers using the DAC.

constructed as described above. Performing select$(k) - k$ over this bitvector returns the number of 0s prior to the kth 1 and recovers the difference; in essence, a prefix sum over the partial differences where we remove the k number of 1s from our calculation. Adding the stored $M[0]$ to this difference restores the original value. Given our example and $k = 3$, we have

$$M[0] + \text{select}(3) - 3 = 11 + 11 - 3 = 19 = M[2]$$

and we recover the correct value at $M[2]$ ($i = 2$ corresponds to the $k = 3$ bit due to 0-based array indexing).

To further optimize for practical input, consider an alternative to the wavelet tree suitable for small alphabets or when query support is needed for only a subset of characters. Where the wavelet tree performs rank and select over multiple tree levels, we could instead store full length uncompressed bitvectors in our blocks, one for each chosen character c marking positions i where $\text{BWT}[i] = c$. For large alphabets, this approach is much larger than a wavelet tree representation; however, for genomic datasets which in practice support queries on few characters such as the nucleobases $\{A, C, G, T\}$, this alternative may be preferred. As this is the case in our experiments, we use this restricted alphabet trick to trade off space for increased speed in performing rank and select operations. A summary of the structure of our proposed practical approach is shown in Figure 3; an overview of the hierarchy of the proposed optimizations with respect to components of the data structure and the varying options which we have implemented.

3.3 Scanning Complexity

We have not yet implemented the second part of Nishimoto and Tabei's result because we correctly expected their idea of table lookup (perhaps modified slightly) to be interesting and practical by itself. Over real world datasets (as discussed in Section 5), our typical sequential scan is very small; however, theoretically we use $\Omega(r)$-time in the worst case for such a scan for LF . In fact, there are strings for which the average time for a scan is $\Omega(r)$. Suppose a string has $\text{BWT}[0..n-1] = (bc)^{n/10} \cdot (a)^{4n/5}$ with $r = n/5 + 1$ runs. By LF properties we have $\frac{3n}{5}$ LF steps which require scanning $r - 1$ rows, as described in Figure 4. Similarly, we encounter $\Omega(n \cdot r)$-time for inversion, as we perform exactly n possible LF steps during a full retrieval of the original string.

In practice, a very similar string can be produced which preserves a similar worst case. Consider a randomly generated binary string, for our purposes over the alphabet $\Sigma = \{b, c\}$. We then interleave the sequence with four consecutive a characters between each of the original characters (resulting in $\frac{4n}{5}$ a characters). The number of runs we expect in its BWT cannot be much less than the number of runs in the original sequence, since the introduced a characters are easily run-length compressed and such a technique would improve compression of any random sequence otherwise. The expected number of runs in a random binary string is half its length ($\frac{n}{10}$), also observed in practical experiments, and thus mapping to a characters results in almost the same case as Figure 4. To perform some practical bounding against scans without theoretical guarantees, we allow splitting of large runs by specifying the maximum acceptable run length to provide an alternative construction.

3.4 Count Queries

Standard FM-indexes are particularly good at counting queries, both in theory and in practice, and counting was also the first query supported quickly and in small space by RLBWT-based versions of the FM-index [13] (time- and space-efficient reporting was developed much

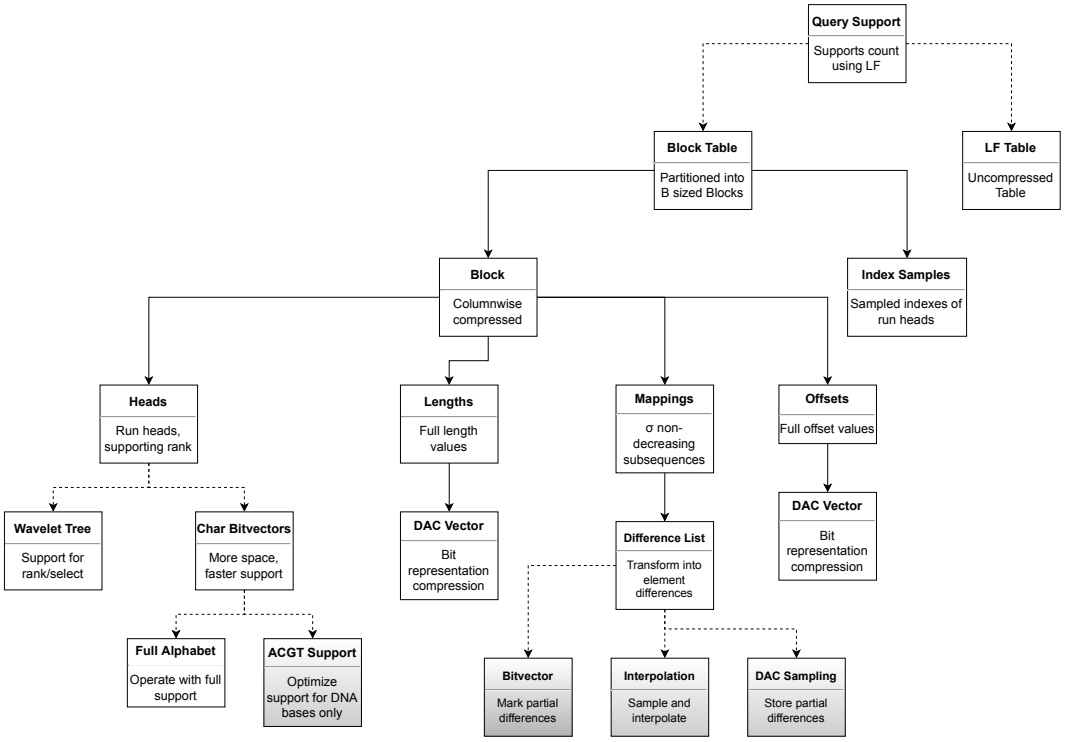


Figure 3 Shows hierarchy of implementation, outlining different approaches and optimizations. Solid lines show required components of parents given our work, where dotted lines denote multiple options being available. For example, the various methods to recover the mapping of a run head are shown as children of difference list. Shaded nodes show paths that are implemented for experiments in Section 4.

later [6]). It seems appropriate, therefore, to test with counting queries our implementation of the first part of Nishimoto and Tabei's result. A counting query for pattern $S[0..m-1]$ in text $T[0..n-1]$ returns the number of occurrences of S in T, by backward searching for S and returning the length of the BWT interval containing the characters preceding occurrences of S in T. We can implement a backward step using access to the string R described in Section 2, up to 2 rank queries and 2 select queries on R, and 2 LF queries.

Suppose the interval BWT$[s..e]$ contains the characters preceding occurrences of $S[i+1..m-1]$ in T and we know both the indices j_s and j_e of the runs containing BWT$[s]$ and BWT$[e]$, and the offsets of those characters in those runs. We need not assume we know s and e themselves. If $R[j_s] = S[i]$ then the BWT interval containing the characters preceding occurrences of $S[i..m-1]$ in T, starts at BWT[LF(s)]. Otherwise, it starts at BWT[LF(s')], where s' is the first character in run

$$j_{s'} = R.\text{select}_{S[i]}(R.\text{rank}_{S[i]}(j_s) + 1)\,.$$

Symmetrically, if $R[j_e] = S[i]$ then the interval ends at BWT[LF(e)]; otherwise, it ends at BWT[LF(e')], where e' is the last character in run

$$j_{e'} = R.\text{select}_{S[i]}(R.\text{rank}_{S[i]}(j_e))\,.$$

(If $j_{s'} > j_{e'}$ then $S[i..m-1]$ does not occur in T.) These operations are all supported across our block compressed table, and the final interval positions in the BWT can be computed using sampled run head positions to return the final count.

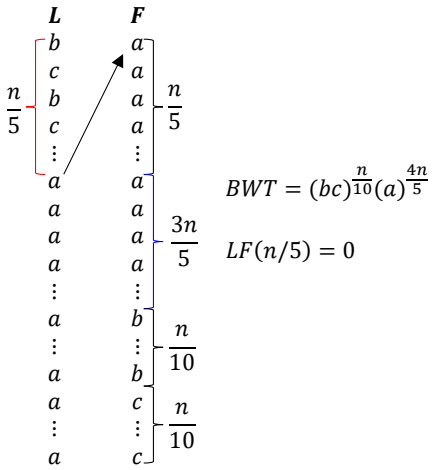


Figure 4 Visual representation of amortized analysis in Section 3.3. Notice that given a BWT of this form, any character a corresponding to run k with $I[k] = n/5$ stores $\text{LF}(n/5) = 0$ as its mapping. If the offset d is greater than $n/5$, then the sequential scan must cross the boundaries of each run of b or c, of which there are $n/5$ in total; since there is only one run of a, we scan $r - 1$ entries, and perform this operation for $\frac{3n}{5}$ possible steps. Amortized over all possible LF steps, we cannot avoid $\Omega(r)$ scans in the worst case.

4 Experiments

Our code was written in C++ and compiled with flags `-O3 -DNDEBUG -funroll-loops -msse4.2` using data structures from `sdsl-lite` [7]. We performed our experiments on a server with an Intel® Xeon® Silver 4214 CPU running at 2.20GHz with 32 cores and 100 GB of memory. Our code is available at `https://github.com/drnatebrown/r-index-f.git`. Count query times were measured using Google Benchmark, and construction with the Unix /usr/bin/time command.

4.1 Data Structures

For our table lookup implementations, we partition into blocks of size $B = 2^{20}$ and sample every 16th run position in the BWT. We compared the following data structures:

lookup-bv table lookup with bitvector marking differences with 0s, recovered with select described in Section 3.2.

lookup-int table lookup with linear interpolation between sampled values described in Section 3.2 with sample rate 16.

lookup-dac table lookup with DAC sampling of differences described in Section 3.2 with sample rate 5.

lookup-split2 table lookup with naive run splitting using *lookup-bv* data structure described in Sections 3.2, 3.3. Runs larger than twice the average length n/r are split.

lookup-split5 table lookup identical to *lookup-split2*, except runs larger than five times the average length n/r are split.

wt-fbb fixed-block boosting wavelet tree of [8] using default parameters; implementation at `https://github.com/dominikkempa/faster-minuter`.

rle-string run-length encoded string of the r-index [6]; implementation based off `https://github.com/nicolaprezza/r-index`.

RLCSA the BWT component[8] of the run-length encoded compressed suffix array of [13] using default parameters; implementation at `https://github.com/adamnovak/rlcsa`.

4.2 Datasets

We tested our data structures for construction and query on 4 collections of 128, 256, 512 and 1000 haplotypes of chromosome 19 from the 1000 Genomes Project [17] (`chr19`) and 4 collections of 100k, 200k, 300k, 400k SARS-CoV2 genomes from the EBI's COVID-19 data portal [9][9] (`Sars-CoV2`). Each set is a superset of the previous one. Table 1 describes the lengths n and ratio n/r of the datasets.

Table 1 Table of the different datasets. In column 1 and 2 we report the name and description of the datasets, in column 3 we report the number of sequences in the collection, in column 4 we report the length of the file, and in column 5 the ratio of the length to the number of runs in the BWT.

Name	Description	N	$n/10^6$	n/r
`chr19`	Human chromosome 19	128	7568.01	222.24
`chr19`	Human chromosome 19	256	15136.04	424.93
`chr19`	Human chromosome 19	512	30272.08	771.54
`chr19`	Human chromosome 19	1,000	59125.12	1287.38
`Sars-CoV2`	Sars-CoV2 genomes database	100,000	2979.01	881.16
`Sars-CoV2`	Sars-CoV2 genomes database	200,000	5958.35	977.19
`Sars-CoV2`	Sars-CoV2 genomes database	300,000	8944.37	1178.00
`Sars-CoV2`	Sars-CoV2 genomes database	400,000	11931.17	1328.92

4.3 Construction

In Figure 5 we report the time and memory for construction of the data structures for the `chr19` and `Sars-CoV2` datasets. `RLCSA` is omitted, since it is the only data structure not built using prefix free parsing (PFP) [3], and its construction time far exceeded the other methods.

4.4 Query

To query the data structures we performed counting queries for 10000 randomly chosen substrings each of length 10, 100, 1000 and 10000. In Figure 6 and 7 we report the time and memory for querying of the data structures for the `chr19` and `Sars-CoV2` datasets respectively.

5 Discussion

With respect to our table lookup implementations, `lookup-bv` and its variants (`lookup-split2`, `lookup-split5`) perform better than the alternatives (`lookup-int`, `lookup-dac`) a majority of the time across all queries, while being smaller in space. For query lengths greater than 10 on `chr19`, these approaches are faster than `rle-string` but slightly larger, while

[8] We build the data structure without suffix-array sampling.
[9] The complete list of accession numbers is reported in the repository.

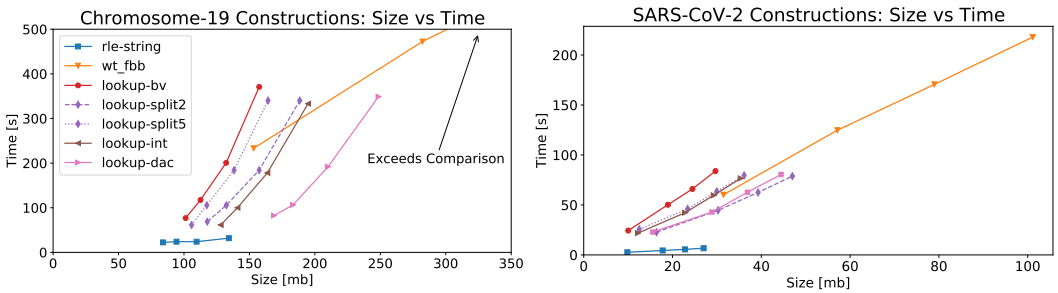


Figure 5 Construction for `chr19` of 128, 256, 512 and 1000 copies (left) and `Sars-CoV2` of 100k, 200k, 300k and 400k copies (right). Copies increase for an instance plotted left to right. For `chr19` we partially omit `wt-fbb` for being magnitudes larger than other values (approximately 4 times slower and larger than `lookup-bv` for 512 copies and similarly 5 times slower and 7 times larger for 1000).

slower than `RLCSA` but smaller in size; we occupy a time/space trade-off position between these values. This is while also being much smaller than `wt-fbb` whose space makes it an outlier despite best speeds for various queries.

On `Sars-CoV2`, our implementations perform well on queries of length 10, with `lookup-split2` the fastest implementation and other approaches competitive in both time/space. For query lengths greater than 10, the non-splitting approaches (`lookup-bv`, `lookup-int`, `lookup-dac`) perform the worst across data structures with respect to speed. With splitting approaches, we are comparable to `rle-string` in time but worse in space. Although again an outlier in space, `wt-fbb` performs fastest, with `RLCSA` occuping the least space with comparable speed to `wt-fbb`.

In terms of size/construction, we perform worse than `r`le-string across all data, but are highly competitive for `lookup-bv`'s space despite slower construction. For our implementations, `lookup-bv` is the definitive choice across results in regard to both space and construction time. When compared to `RLCSA`, despite being more space-efficient on `chr19` across `lookup-bv` approaches, we cannot compete on `Sars-CoV2` where it is a clear winner across all data structures. This motivates applying table lookup to also speed up `RLCSA`; however, we note adding support for ϕ and ϕ^{-1} (thus, supporting locate) to `RLCSA` is still an open problem.

With regard to our splitting approaches, they are superior to `lookup-bv` for long query lengths and as n/r rises. To examine the cause in terms of n/r and growing text collections, we examine the number of sequential scans required across LF steps during count queries of length 100 for `chr19` in Figure 8. Although the distribution is similar across all copies near zero, with a majority requiring no sequential scan and most of the rest scanning very few, worst cases become both more prevalent and longer as the number of copies and n/r grows. This gives further insight into the success of the splitting approaches in these instances, as bounding the maximum runs also bounds worst case sequential scans. We find this result intriguing with respect to Theorem 1 when n/r or the worst case number of scans is high. Concentrating on Nishimoto and Tabei's first result, `lookup-bv` performs competitively in space/time for low n/r with naive run splitting as a practical alternative otherwise in our observed experiments.

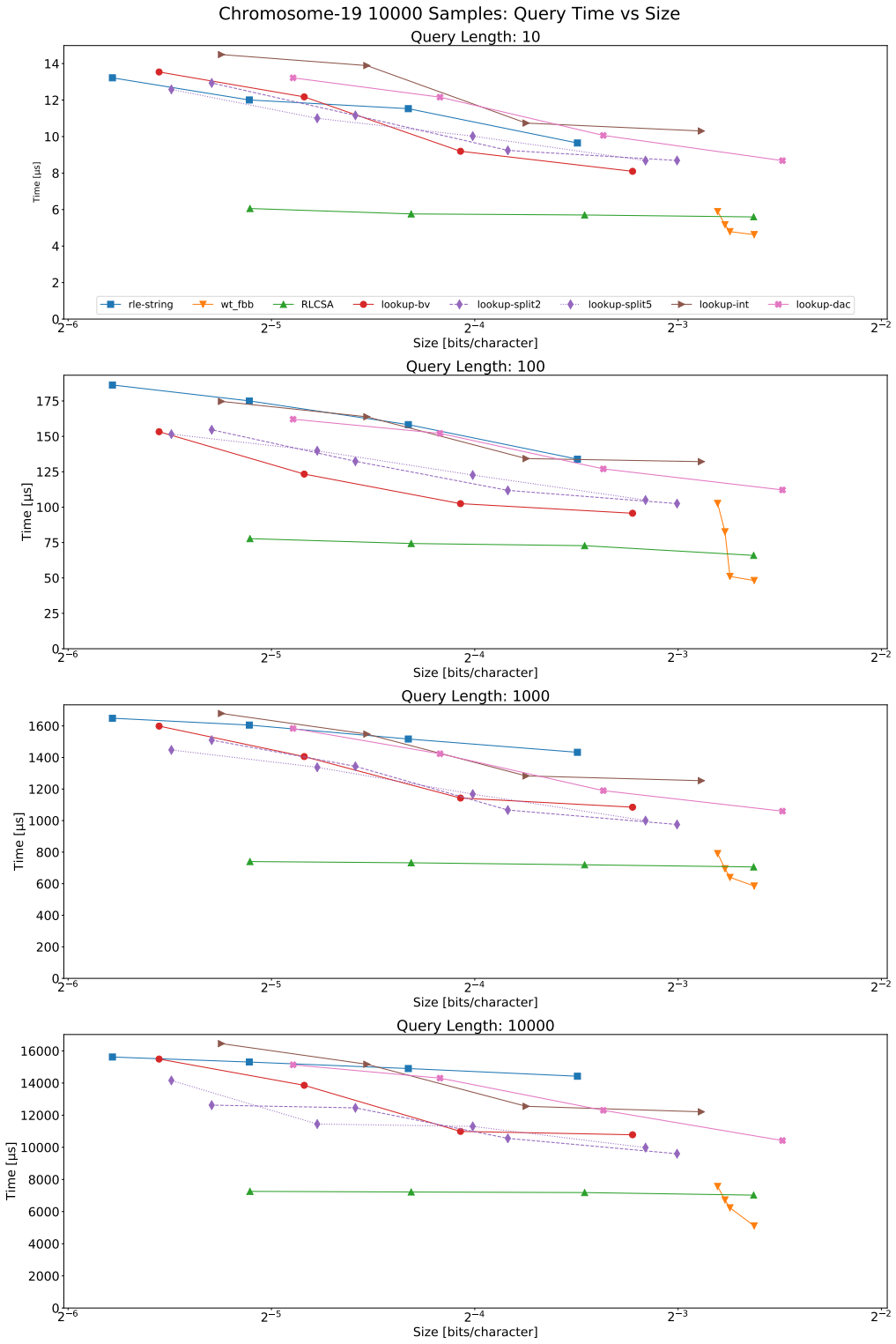


Figure 6 The time per query to count the occurrences of 128, 256, 512 and 1000 copies of `chr19` for 10000 randomly-chosen substrings of length 10, 100, 1000 and 10000 each. Copies for a single line are read from largest number of copies to smallest, left to right. The x axis is logarithmically scaled, motivated by doubling the number of copies across examples.

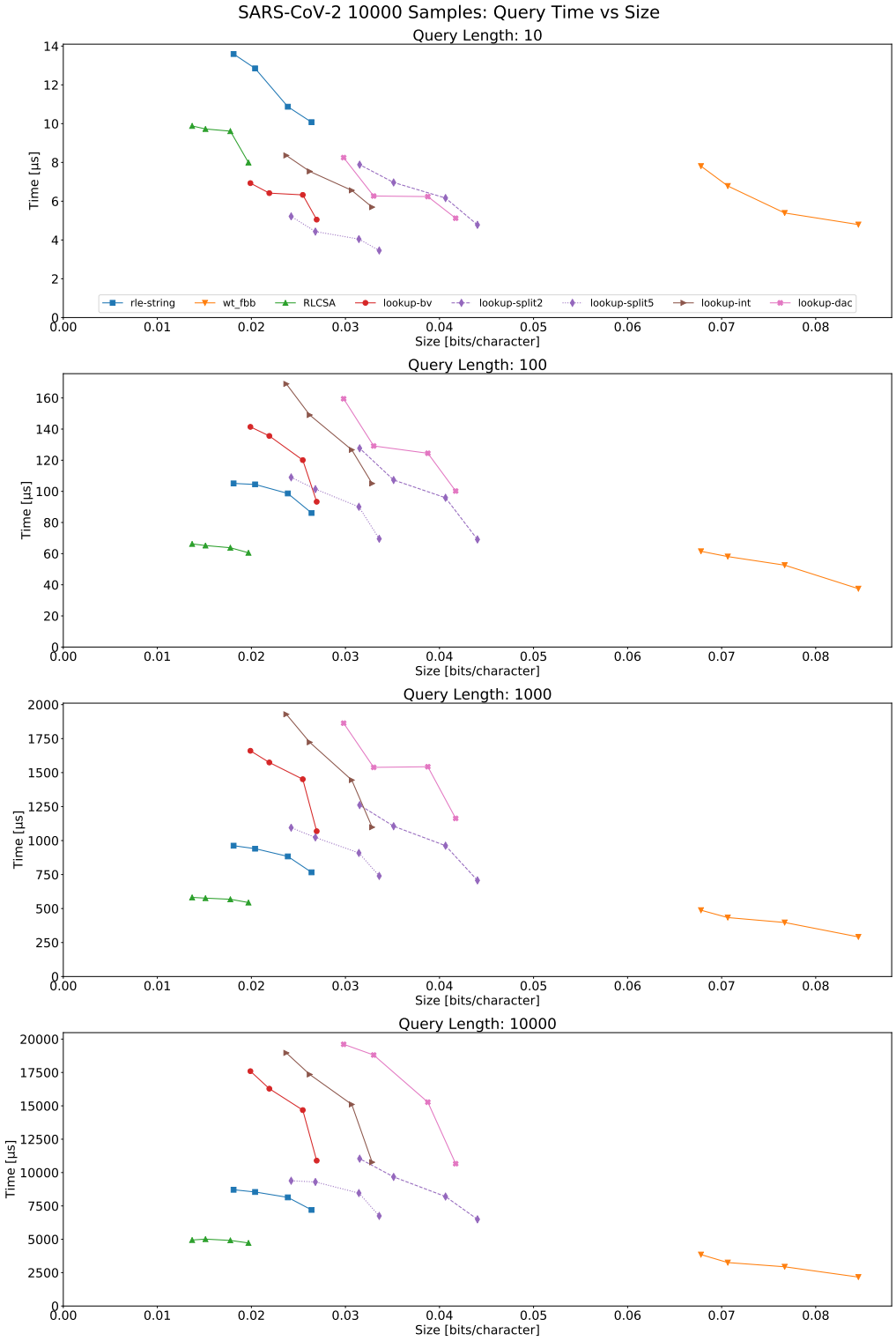


Figure 7 The time per query to count the occurrences of 100k, 200k, 300k and 400k `Sars-CoV2` copies for 10000 randomly-chosen substrings. Results are given for queries of length 10, 100, 1000 and 10000. Copies for a single line are read from largest number of copies to smallest, left to right.

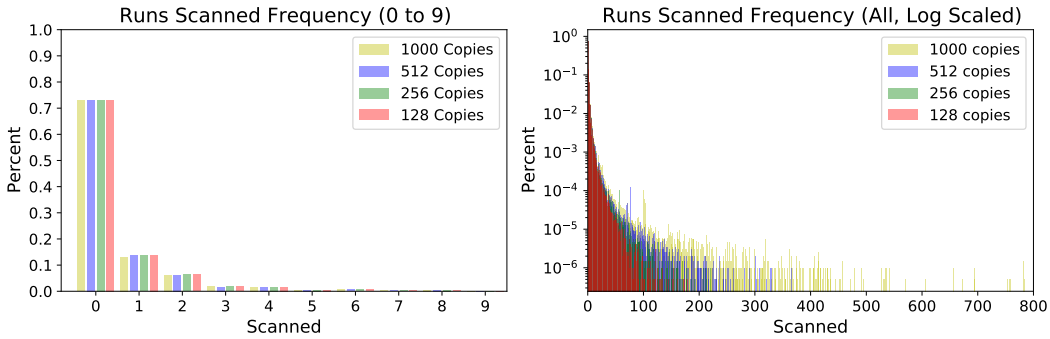


Figure 8 Frequencies in percentage of runs scanned for any LF step across 10000 count queries of length 100 for 100, 200, 512 and 1000 copies of `chr19`. Plot on left is restricted only to steps scanning 0 to 9 runs; plot on right shows all scans, log scaled since the frequency of scans decreases quickly for large values.

References

1. Omar Ahmed, Massimiliano Rossi, Sam Kovaka, Michael C. Schatz, Travis Gagie, Christina Boucher, and Ben Langmead. Pan-genomic matching statistics for targeted nanopore sequencing. *iScience*, 24(6):102696, 2021.
2. Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theor. Comput. Sci.*, 812:96–108, 2020.
3. Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Mol. Biol.*, 14(1):13:1–13:15, 2019.
4. Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, DEC, 1994.
5. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
6. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.
7. Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.
8. Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Fixed block compression boosting in FM-indexes: Theory and practice. *Algorithmica*, 81(4):1370–1391, 2019.
9. Peter W Harrison et al. The COVID-19 Data Portal: accelerating SARS-CoV-2 and COVID-19 research through rapid open access data sharing. *Nucleic Acids Research*, 49(W1):W619–W623, 2021.
10. Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan-genomics read alignment. *J. Comput. Biol.*, 27(4):500–513, 2020.
11. Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3):1–10, 2009.
12. Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinform.*, 25(14):1754–1760, 2009.
13. Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010.

14 Gonzalo Navarro. *Compact Data Structures – A Practical Approach*. Cambridge University Press, 2016.

15 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on bwt-runs compressed indexes. In *48th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 101:1–101:15, 2021.

16 Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. Moni: A pangenomic index for finding maximal exact matches. *J. Comput. Biol.*, 29(2):169–187, 2022.

17 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

Heuristic Computation of Exact Treewidth

Hisao Tamaki
Department of Computer Science, Meiji University, Tokyo, Japan

Abstract

We are interested in computing the treewidth $\mathrm{tw}(G)$ of a given graph G. Our approach is to design heuristic algorithms for computing a sequence of improving upper bounds and a sequence of improving lower bounds, which would hopefully converge to $\mathrm{tw}(G)$ from both sides. The upper bound algorithm extends and simplifies the present author's unpublished work on a heuristic use of the dynamic programming algorithm for deciding treewidth due to Bouchitté and Todinca. The lower bound algorithm is based on the well-known fact that, for every minor H of G, we have $\mathrm{tw}(H) \leq \mathrm{tw}(G)$. Starting from a greedily computed minor H_0 of G, the algorithm tries to construct a sequence of minors $H_0, H_1, \ldots H_k$ with $\mathrm{tw}(H_i) < \mathrm{tw}(H_{i+1})$ for $0 \leq i < k$ and hopefully $\mathrm{tw}(H_k) = \mathrm{tw}(G)$.

We have implemented a treewidth solver based on this approach and have evaluated it on the bonus instances from the exact treewidth track of PACE 2017 algorithm implementation challenge. The results show that our approach is extremely effective in tackling instances that are hard for conventional solvers. Our solver has an additional advantage over conventional ones in that it attaches a compact certificate to the lower bound it computes.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases graph algorithm, treewidth, heuristics, BT dynamic programming, contraction, obstruction, minimal forbidden minor, certifying algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.17

Acknowledgements I thank Holger Dell for posing the challenging bonus instances, which have kept defying my "great ideas", showing how they fail, and pointing to yet greater ideas.

1 Introduction

Treewidth is a graph parameter which plays an essential role in the graph minor theory [15, 16, 17] and is an indispensable tool in designing graph algorithms (see, for example, a survey [6]). See Section 2 for the definition of treewidth and tree-decompositions. Let $\mathrm{tw}(G)$ denote the treewidth of graph G. Deciding if $\mathrm{tw}(G) \leq k$ for given G and k is NP-complete [2], but admits a fixed-parameter linear time algorithm [5].

Practical algorithms for treewidth have also been actively studied [8, 9, 3, 20, 18, 1], with recent progresses stimulated by PACE 2016 and 2017 [13] algorithm implementation challenges. The modern treewidth solvers use efficient implementations of the dynamic programming algorithm due to Boudhitté and Todinca (BT) [10]. After a first leap in that direction [20], some improvements have been reported [18, 1], but those improvements are incremental.

In this paper, we pursue a completely different approach. We develop a heuristic algorithm for the upper bound as well as one for the lower bound. These algorithms iteratively improve the bounds in hope that they converge to the exact treewidth from both sides.

Our upper bound algorithm is based on the following idea. For $\mathcal{B} \subseteq 2^{V(G)}$, we say that $\mathcal{B}$ admits a tree-decomposition of G if every bag of this tree-decomposition belongs to $\mathcal{B}$. The treewidth of G with respect to $\mathcal{B}$, denoted by $\mathrm{tw}_{\mathcal{B}}(G)$, is the smallest k such that $\mathcal{B}$ admits a tree-decomposition of G of width k. If $\mathcal{B}$ admits no tree-decomposition of G, then $\mathrm{tw}_{\mathcal{B}}(G)$ is undefined. A vertex set $X \subseteq V(G)$ is a *potential maximal clique* of G if it is a maximal clique of some minimal triangulation of G. We denote by $\Pi(G)$ the set of all potential maximal

cliques of G. Bouchitté and Todinca [10] observe that $\Pi(G)$ admits a tree-decomposition of G of width $\mathrm{tw}(G)$ and present a dynamic programming algorithm (BT dynamic programming) to compute $\mathrm{tw}(G)$ based on this fact. Indeed, BT dynamic programming can be applied to an arbitrary set Π of potential maximal cliques to compute $\mathrm{tw}_\Pi(G)$. This allows us to work in a solution space where each solution is a set of potential maximal cliques rather than an individual tree-decomposition. A solution Π encodes a potentially exponential number of tree-decompositions it admits and offers rich opportunities of improvements in terms of $\mathrm{tw}_\Pi(G)$. This approach has been proposed by the present author in his unpublished work [19], where he presents several *ad hoc* operations to enrich Π in hope of reducing $\mathrm{tw}_\Pi(G)$. We extend and simplify this approach by replacing those operations with a single merging operation: given two sets Π_1 and Π_2 of potential maximal cliques, we construct a new set Π that includes $\Pi_1 \cup \Pi_2$ together with some additional potential maximal cliques potentially useful for making $\mathrm{tw}_\Pi(G)$ smaller than both $\mathrm{tw}_{\Pi_1}(G)$ and $\mathrm{tw}_{\Pi_2}(G)$. See Section 3 for more details.

The lower bound algorithm is based on the well-known fact that, for every minor H of G, we have $\mathrm{tw}(H) \leq \mathrm{tw}(G)$. Starting from a greedily computed minor H_0 of G, the algorithm tries to construct a sequence of minors $H_0, H_1, \ldots H_k$ with $\mathrm{tw}(H_i) < \mathrm{tw}(H_{i+1})$ for $0 \leq i < k$ and hopefully $\mathrm{tw}(H_k) = \mathrm{tw}(G)$. Although minors have been used to compute lower bounds on the treewidth [9], the goal has been to quickly obtain a lower bound of reasonable quality to be used, say, in branch-and-bound procedures. There seems to be no attempt in the literature to develop an algorithm which, given a minor H of G with $\mathrm{tw}(H) < \mathrm{tw}(G)$, construct a minor H' of G with an improved lower bound $\mathrm{tw}(H') > \mathrm{tw}(H)$. In view of this lack of attempts, our finding that this task can be performed with reasonable efficiency in practice might be somewhat surprising. See Section 4 for details.

We have implemented a treewidth solver based on this approach and evaluated it on the bonus instance set from the PACE 2017 algorithm implementation challenge for treewidth. This set is designed to remain challenging for solvers to be developed after the challenge. It consists of 100 instances and, according to the summary provided with the set, the time spent to compute the exact treewidth by the winning solvers of PACE 2017 is longer than an hour for 57 instances and longer than 12 hours for 23 instances, including 9 instances which fail to be solved at all. The results of applying our solver on the 91 solved instances are summarized as follows. With a timeout of 30 minutes using two threads (one for the upper bound and the other for the lower bound), 62 instances are exactly solved; for 20 of the other instances, the upper bound equals the exact treewidth and the lower bound is off by one. Moreover, with a timeout of 6 hours, our solver exactly solves 2 of the 9 unsolved instances. These results suggest that our approach is extremely effective in coping with instances that are hard for conventional solvers. See Section 6 for details.

The source code of the solver used in our experiments is available at [22].

2 Preliminaries

Graph notation

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. As G is simple, each edge of G is a subset of $V(G)$ with exactly two members that are adjacent to each other in G. The *complete graph* on V, denoted by $K(V)$, is a graph with vertex set V in which every vertex is adjacent to all other vertices. The subgraph of G induced by $U \subseteq V(G)$ is denoted by $G[U]$. We sometimes use an abbreviation $G \setminus U$ to stand for $G[V(G) \setminus U]$.

A vertex set $C \subseteq V(G)$ is a *clique* of G if $G[C]$ is a complete graph. For each $v \in V(G)$, $N_G(v)$ denotes 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 neighborhood 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 neighborhood of U in G*, denoted by $N_G[U]$, is defined by $N_G[U] = U \cup N_G(U)$.

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* or simply a *component* of G if it is connected and is inclusion-wise maximal subject to this condition. A vertex set $S \subseteq V(G)$ is a *separator* of G if $G \setminus S$ has more than one component. A graph is a *cycle* if it is connected and every vertex is adjacent to exactly two vertices. A graph is a *forest* if it does not have a cycle as a subgraph. A forest is a *tree* if it is connected.

Tree-decompositions

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, indexed by the nodes of T, such that the following three conditions are satisfied. We call 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 $I_v = \{i \in V(T) \mid v \in X_i\} \subseteq V(T)$ is connected in 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 smallest k such that there is a tree-decomposition of G of width k.

It is well-known that, for each pair (i, j) of adjacent nodes of a tree-decomposition $\mathcal{T} = (T, \mathcal{X})$, the intersection $X_i \cap X_j$ is a separator of G. We say that $\mathcal{T}$ *induces* separator S if there is an adjacent pair (i, j) such that $S = X_i \cap X_j$.

Minimal separators and potential maximal cliques

Let G be a graph and S a separator of G. For distinct vertices $a, b \in V(G)$, S is an *a-b separator* if there is no path between a and b in $G \setminus S$; it is a *minimal a-b separator* if it is an a-b separator and no proper subset of S is an a-b separator. A separator is a *minimal separator* if it is a minimal a-b separator for some $a, b \in V(G)$.

Graph H is *chordal* if every induced cycle of H has exactly three vertices. H is a *triangulation of graph G* if it is chordal, $V(G) = V(H)$, and $E(G) \subseteq E(H)$. A triangulation H of G is *minimal* if it there is no triangulation H' of G such that $E(H')$ is a proper subset of $E(H)$. A vertex set $X \subseteq V(G)$ is a *potential maximal clique* of G, if X is a maximal clique in some minimal triangulation of G. We denote by $\Pi(G)$ the set of all potential maximal cliques of G and by $\Pi_k(G)$ the set of all potential maximal cliques of G of cardinality at most k.

Bouchitté-Todinca dynamic programming

The treewidth algorithm of Bouchitté and Todinca [10] is based on the fact that every graph G has a minimal triangulation H such that $\mathrm{tw}(H) = \mathrm{tw}(G)$ (see [14] for a clear exposition). This fact straightforwardly implies that $\Pi(G)$ admits an optimal tree-decomposition of G. Their algorithm consists of an algorithm for constructing $\Pi(G)$ and a dynamic programming algorithm (BT dynamic programming) to compute $\mathrm{tw}_{\Pi(G)}(G)$. As noted in the introduction, BT dynamic programming can be applied to compute $\mathrm{tw}_\Pi(G)$ for an arbitrary $\Pi \subseteq \Pi(G)$.

The most time-consuming part of their treewidth algorithm is the construction of $\Pi(G)$. Empirically observing that $\Pi_{k+1}(G)$ is substantially smaller than $\Pi(G)$ for $k \leq \mathrm{tw}(G)$, authors of modern implementations of the BT algorithm [20, 18, 1] use BT dynamic programming

with $\Pi = \Pi_{k+1}(G)$ to decide if $\mathrm{tw}(G) \leq k$. Moreover, they try to avoid the full generation of $\Pi_{k+1}(G)$, by being lazy and generating a potential maximal clique only when it becomes absolutely necessary in evaluating the recurrence.

Both our upper and lower bound algorithms use, as a subprocedure, such an implementation of the BT algorithm for treewidth, in particular an implementation of the version proposed in [18]. In addition, our upper bound algorithm uses BT dynamic programming in its fully general form, to evaluate each solution Π in our solution space as described in the introduction. The efficiency of BT dynamic programming, which runs in time linear in $|\Pi|$ with a factor polynomial in $|V(G)|$, is crucial in our upper bound algorithm.

Contractions and minors

Let $F \subseteq E(G)$ be a forest on $V(G)$. The *contraction* of G by F, denoted by G/F is a graph whose vertices are the connected components of F and two components C_1 and C_2 are adjacent to each other if and only if there is $v_1 \in C_1$ and $v_2 \in C_2$ such that v_1 and v_2 are adjacent to each other in G. A graph H is a *minor* of G if it is a subgraph of some contraction of G. It is well-known and is easy to verify that $\mathrm{tw}(H) \leq \mathrm{tw}(G)$ if H is a minor of G.

Minimal triangulation algorithms

There are many algorithms for minimal triangulation of a graph (see [14] for a survey). For purposes in the current work, we are interested in algorithms that produce a minimal triangulation of small treewidth. Although a minimal triangulation H of G such that $\mathrm{tw}(H) = \mathrm{tw}(G)$, hence of the smallest treewidth, can be computed by the BT algorithm for treewidth, we need a faster heuristic algorithm. The MMD (Minimal Minimum Degree) algorithm [4] is known to perform well, in terms of the treewidth of the resulting triangulation. We use a variant MMAF (Minimal Minimum Average Fill) [21] of MMD which performs slightly better than the original MMD on benchmark instances.

Safe separators and almost-clique separators

Bodlaender and Koster [7] introduced the notion of safe separators for treewidth. Let S be a separator of a graph G. We say that S is *safe for width* k, if S is induced by some tree-decomposition of G of width k. It is simply safe if it is safe for width $\mathrm{tw}(G)$. The motivation of looking at safe separators is the fact that there are easily verifiable sufficient conditions for S being safe and a safe separator detected by those sufficient conditions can be used to reduce the problem of deciding if $\mathrm{tw}(G) \leq k$ to smaller subproblems. A trivial sufficient condition is that S is a clique. Bodlaender and Koster observed that this condition can be relaxed to S being an *almost-clique*, where S is an almost-clique if $S \setminus \{v\}$ is a clique for some $v \in S$. More precisely, a minimal separator that is an almost-clique is safe. They showed that this observation leads to a powerful preprocessing method of treewidth computation. An *almost-clique separator decomposition* of graph G is a tree-decomposition $\mathcal{A}$ of G such that every separator induced by $\mathcal{A}$ is an almost-clique minimal separator. For each bag A_i of $\mathcal{A}$, let G_i be a graph on A_i obtained from $G[A_i]$ by adding edges of $K(N(C))$ for every component C of $G \setminus A_i$. The following proposition holds [7].

▶ **Proposition 1.**

1. $\mathrm{tw}(G)$ *is the maximum of* $\mathrm{tw}(G_i)$*, where* i *ranges over the nodes of the decomposition, and a tree-decomposition of* G *of width* $\mathrm{tw}(G)$ *is obtained by combining tree-decompositions of* G_i *as prescribed by* $\mathcal{A}$*.*
2. G_i *is a minor of* G *for each* i*.*

Unpublished work of the present author [21] shows that this preprocessing approach is effective for instances that are much larger than those tested in [7], using a heuristic method for constructing almost-clique separator decompositions. We use his implementation in the current work.

We also make an unconventional use of safe separators in our lower bound algorithm. When we have a lower bound of k on $\mathrm{tw}(G)$, we wish to evaluate a minor H of G for the possibility of leading to a stronger lower bound. We use the set of all minimal separators of H that are safe for width k in this evaluation. Note that the computation of such a set is possible because H is small.

3 The upper bound algorithm

Recall that $\Pi(G)$ denotes the set of all potential maximal cliques of G. In our upper bound algorithm, a *solution* for G is a subset Π of $\Pi(G)$ that admits at least one tree-decomposition of G and the *value* of solution Π is $\mathrm{tw}_\Pi(G)$.

Our algorithm starts from a greedy solution and iteratively improves the solution. To improve a solution Π, we *merge* it with another solution Ω into another solution Π' in hope of having $\mathrm{tw}_{\Pi'}(G) < \min\{\mathrm{tw}_\Pi(G), \mathrm{tw}_\Omega(G)\}$. This merged solution Π' contains $\Pi \cup \Omega$ together with some other potential maximal cliques so that Π' would admit a tree-decomposition that contains some bags in Π, some bags in Ω, and some bags belonging to this additional set of potential maximal cliques. We describe below how these additional potential maximal cliques are computed.

Let $X \in \Pi$ and $Y \in \Omega$ be distinct and not crossing each other. Then, there is a unique component C of $G \setminus X$ such that $Y \subseteq N[C]$ and a unique component D of $G \setminus Y$ such that $X \subseteq N[D]$. Let $U = N[C] \cap N[D]$ and let H be a graph on U obtained from $G[U]$ by adding edges of $K(N(B))$ for each component B of $G \setminus U$. Let $\hat{H}$ be a minimal triangulation of H with small treewidth: if $|U|$ does not exceed a fixed threshold BASE_SIZE (=60), then we use the BT algorithm for treewidth to compute $\hat{H}$; otherwise we use MMAF to compute $\hat{H}$. Here, BASE_SIZE is a parameter of the algorithm represented as a constant in the implementation. The parenthesized number is the value of this parameter used in our experiment. In the following, we use the same convention for citing algorithm parameters. Note that each maximal clique of $\hat{H}$ is either a potential maximal clique of G or a minimal separator of G. If $\mathrm{tw}(\hat{H}) \leq \mathrm{tw}_\Pi(G)$, then we add all potential maximal cliques of G that are maximal cliques of $\hat{H}$ to Π'. When this happens, then Π' admits a tree-decomposition of width at most $\max\{\mathrm{tw}_\Pi(G), \mathrm{tw}_\Omega(G)\}$ consisting of some bags in Π, some bags in Ω, and some bags that are maximal cliques of $\hat{H}$. In this way, Π' would admit tree-decompositions that are not admitted by the simple union $\Pi \cup \Omega$ and, with some luck, some of the newly admitted tree-decomposition may have width smaller than $\mathrm{tw}_\Pi(G)$.

The procedure MERGE(Π, Ω) merges Π with Ω, applying the above operation to some pairs $X \in \Pi$ and $Y \in \Omega$ chosen as follows. We first pick $X \in \Pi$ uniformly at random. Then, let C be the largest component of $G \setminus X$. We choose $Y \in \Omega$ such that $Y \subseteq N[C]$ and $|Y| \leq \mathrm{tw}_\Pi(G)$. The first condition ensures that the method in the previous paragraph can be applied to this pair of X and Y. The second condition is meant to increase the chance of newly admitted tree-decompositions to have width smaller than $\mathrm{tw}_\Pi(G)$. We sort the candidates of such Y in the nondecreasing order of $|N[C] \cap N[D]|$, where D is the component of Y such that $X \subseteq N[D]$, and use the first N_TRY (=50) elements of this sorted list. We prefer Y such that $U = N[C] \cap N[D]$ is small, because that would increase the chance of the minimal triangulation $\hat{H}$, described in the previous paragraph, to have small treewidth. All the resulting potential maximal cliques from these trials are added to Π'.

In addition to procedure Merge, our algorithm uses two more procedures InitialSolution() and Improve(Π) described below. The input graph G is fixed in these procedures.

InitialSolution() generates an initial solution Π. We use a randomize version of MMAF to generate N_INITIAL_GREEDY(=10) minimal triangulations of G and take H with the smallest treewidth. The solution Π returned by the call InitialSolution() is the set of maximal cliques of H.

Improve(Π) returns a solution Π′ with $\Pi \subseteq \Pi'$, where efforts are made to make $\mathrm{tw}_{\Pi'}(G)$ strictly smaller than $\mathrm{tw}_{\Pi}(G)$. We proceed in the following steps.

1. Let Ω = InitialSolution().
2. While $\mathrm{tw}_{\Omega}(G)) > \mathrm{tw}_{\Pi}(G)$, replace Ω by Improve(Ω).
3. Return Merge(Π, Ω).

Note the solution Ω to be merged with Π is generated independently of Π and improved so that $\mathrm{tw}_{\Omega}(G) \leq \mathrm{tw}_{\Pi}(G)$ before being merged with Π.

Given these procedures, the main iteration of our algorithm proceeds as follows. It is supposed that the algorithm has an access to lower bounds provided by the lower bound algorithm.

1. Let Π = InitialSolution(). Report $\mathrm{tw}_{\Pi}(G)$ as the initial upper bound on $\mathrm{tw}(G)$, together with a tree-decomposition of G of width $\mathrm{tw}_{\Pi}(G)$ admitted by Π.
2. While $\mathrm{tw}_{\Pi}(G)$ is greater than the current lower bound, replace Π by Improve(Π). When this replacement reduces $\mathrm{tw}_{\Pi}(G)$, report this new upper bound on $\mathrm{tw}(G)$, together with a tree-decomposition of G of width $\mathrm{tw}_{\Pi}(G)$ admitted by Π. We also shrink Π, removing all members of cardinality greater than $k+2$, whenever $\mathrm{tw}_{\Pi}(G)$ is improved to k.

4 The lower bound algorithm

In our lower bound algorithm, we use a procedure we call Lift, which, given a graph G and a forest F on $V(G)$, finds another forest F' such that $\mathrm{tw}(G/F') > \mathrm{tw}(G/F)$; it inevitably fails if $\mathrm{tw}(G/F) = \mathrm{tw}(G)$. Given this procedure, the overall lower bound algorithm proceeds in the following steps. Let G be given. It is supposed that the algorithm has an access to the upper bound being computed by the upper bound algorithm.

1. Construct a contraction G/F of G, using a greedy heuristic for contraction-based lower bounds on treewidth.
2. While $\mathrm{tw}(G/F)$ is smaller than the current upper bound on $\mathrm{tw}(G)$, replace F by Lift(G, F) unless this call fails.

When the current upper bound is larger than $\mathrm{tw}(G)$, it is possible that the call Lift(G, F) is made for F such that $\mathrm{tw}(G/F) = \mathrm{tw}(G)$. In such an event, the call would eventually fail but the time it takes would be at least as the time taken by conventional solvers to compute $\mathrm{tw}(G)$. Our solver implements a mechanism to let such a call terminate as soon as the upper bound is updated to be equal to the current lower bound $\mathrm{tw}(G/F)$.

The design of procedure Lift is described in the following subsections.

4.1 Contraction lattice

First consider looking for the result of Lift(G, F) among the subsets of F. Assuming that $\mathrm{tw}(G/F) < \mathrm{tw}(G)$, a subset F' of F such that $\mathrm{tw}(G/F') > \mathrm{tw}(G/F)$ certainly exists.

For each $A \in 2^F$, let H_A denote the contraction $G/(F \setminus A)$. Then, $\Lambda(G,F) = \{H_A \mid A \in 2^F\}$ is a lattice isomorphic to the power set lattice 2^F, with top G and bottom G/F. Brute force searches for H with $\mathrm{tw}(H) > \mathrm{tw}(G/F)$ in this lattice are hopeless as $|F|$ can be large: we typically have $|F| > 100$ for graph instances we target.

We need to understand the terrain of this lattice to design an effective search method. In the remainder of this subsection and subsequent subsections, let k denote $\mathrm{tw}(G/F)$. Call $H \in \Lambda(G,F)$ *lifted* if $\mathrm{tw}(H) > k$; otherwise call it *unlifted*. Let $\mathrm{ML}(G,F)$ denote the set of minimal lifted elements of $\Lambda(G,F)$. Call an unlifted element H_A *covered* if there is some $H_B \in \mathrm{ML}(G,F)$ with $A \subset B$. Let $\mathrm{COV}(G,F)$ denote the set of all covered unlifted elements of $\Lambda(G,F)$. Ideally, we wish to confine our search in $\mathrm{COV}(G,F) \cup \mathrm{ML}(G,F)$. This would be possible if there is a way of knowing, for each covered element H_A and $e \in F \setminus A$, if $H_{A \cup \{e\}}$ is still covered. Then, we would start with the clearly covered element $H_\emptyset$ and greedily ascend the lattice staying in $\mathrm{COV}(G,F)$ until we hit an element in $\mathrm{ML}(G,F)$. Although such an ideal scenario is unlikely to be possible, we still aim at something close to it in the following sense. For each $A \in 2^F$ such that H_A is unlifted, call $S \subseteq A$ an *excess* in A if $H_{A \setminus S}$ is covered. We wish to confine our search among elements with a small excess. We employ a strategy that works only for pairs (G,F) with a special property, which is described in the following subsections.

4.2 Critical fills

Call a pair $\{u,v\}$ of distinct vertices of G a *fill* of G if u and v are not adjacent to each other in G. For a fill e of G, let $G+e$ denote the graph on $V(G)$ with edge set $E(G) \cup \{e\}$. We say that a fill e of G is *critical for* F if $\mathrm{tw}((G+e)/F) > \mathrm{tw}(G/F)$.

Suppose G has a critical fill for F. Observe the following.

1. Since adding a single edge to G/F increases its treewidth, a small number of uncontractions applied to G/F may suffice to increase its treewidth. Thus, we may expect that there is a lifted element H_A in the lattice $\Lambda(G,F)$ such that $|A|$ is smaller than the value that is expected in a general case (without a critical fill). This would make our search for a lifted element easier.
2. The fact that e is a critical fill could be used to guide our search for a lifted element.

The next section describes how we exploit the existence of a critical fill to guide our search.

4.3 Breaking a critical fill

Assuming that G has a fill $e = \{u,v\}$ critical for F, we look for a lifted element H_A in the lattice $\Lambda(G,F)$. We call the procedure for this operation BreakFill(F, e). The reason of this naming is that, informally speaking, we remove the fill e from $G+e$ by uncontracting some edges in F maintaining the treewidth. We need some preparations before we describe this procedure.

▶ **Proposition 2.** *If H_A is unlifted then e is critical for $F \setminus A$.*

Proof. Since H_A is unlifted, we have $\mathrm{tw}(G/(F \setminus A)) = k$. So, it suffices to show that $\mathrm{tw}((G+e)/(F \setminus A)) > k$. We have $\mathrm{tw}((G+e)/F) > k$, since e is critical for F. We also have $\mathrm{tw}((G+e)/(F \setminus A)) \geq \mathrm{tw}((G+e)/F)$, since $(G+e)/F$ is a contraction of $(G+e)/(F \setminus A)$. Therefore we have $\mathrm{tw}((G+e)/(F \setminus A)) > k$. ◀

Let u_A (v_A) denote the vertex of H_A into which u (v, respectively) is contracted. We say that a separator S of H_A *crosses* e if u_A and v_A belong to two distinct components of $H_A \setminus S$. Define $\mathrm{ncs}_k(A, e)$ to be the number of minimal separators of H_A that are safe for width k and moreover cross e. Observe the following.

1. We have $\mathrm{ncs}_k(A, e) > 0$ if H_A is unlifted. To see this, suppose otherwise that H_A is unlifted but $\mathrm{ncs}_k(A) = 0$. Then, H_A has a tree-decomposition $\mathcal{T}$ of width k such that none of the minimal separators induced by $\mathcal{T}$ crosses e. Then, $\mathcal{T}$ is a tree-decomposition of $(G + e)/(F \setminus A)$ as well, contradicting the assumption that e is critical for F and hence for $F \setminus A$ by Proposition 2.
2. We have $\mathrm{ncs}_k(A, e) = 0$ if H_A is lifted. This is simply because H_A does not have any minimal separator that is safe for treewidth k if $\mathrm{tw}(H_A) > k$.

We empirically observe a tendency that $\mathrm{ncs}_k(A, e)$ decreases as H_A approaches a lifted element from below. Based on this observation, we use this function ncs_k to guide our search for lifted elements. We are ready to describe our procedure BreakFill(F, e). It involves two parameters UNC_CHUNK (=5) and N_TRY (=100) and proceeds as follows.

1. Let $A = \emptyset$.
2. While H_A is unlifted, repeat the following:
   a. Pick N_TRY random supersets A' of A with cardinality $|A|$ + UNC_CHUNK (or $|F|$ if this exceeds $|F|$) and let A_{best} be A' such that $\mathrm{ncs}_k(A', e)$ is the smallest.
   b. Replace A by A_{best}.
3. Return A.

This procedure is correct in a purely theoretical sense: since $H_F = G$ is always lifted, provided $\mathrm{tw}(G/F) < \mathrm{tw}(G)$, it eventually returns some A such that H_A is lifted. The time required for this to happen, however, can be prohibitively long since we are supposing that $|F|$ is fairly large. The success of this procedure hinges on the effectiveness of our heuristic relying on the critical fill.

4.4 Procedure Lift

Our procedure Lift(G, F) is recursive and works in the following steps.

1. Choose a fill e of G with the following heuristic criterion. For each $v \in V(G)$, let $d_F(v)$ denote the degree of v' in G/F, where v' is the vertex of G/F into which v contracts. Then we choose $e = \{u, v\}$ so as to maximize the pair $(d_F(u), d_F(v))$ in the lexicographic ordering, where the order of u and v is chosen so that $d_F(u) \leq d_F(v)$.
2. Let F_1 =Lift($(G + e)/F$). If $\mathrm{tw}(G/F_1) > \mathrm{tw}(G/F)$ then return F_1; otherwise, observing that e is critical for F_1, call BreakFill(e, F_1) to find $F_2 \subseteq F_1$ such that $\mathrm{tw}(G/F_2) > \mathrm{tw}(G/F_1)$.
3. Greedily compute a maximal forest F_3 on $V(G)$ such that $F_2 \subseteq F_3$ and $\mathrm{tw}(G/F_3) = \mathrm{tw}(G/F_2)$. Return F_3.

The criterion for choosing e in the first step is based on the following heuristic reasoning. Let u'' (v'') be the vertex of G/F_1 into which u (v, respectively) contracts. We expect that if the size of the minimum vertex cut between u'' and v'' in G/F_1 is large, then the minimum cardinality of A such that H_A is lifted in $\Lambda(G, F_1)$ would have a tendency to be small. Indeed, if the size of this cut is as large as k, then no separator of cardinality at most k crosses e and therefore we have $\mathrm{tw}(H_\emptyset) > 0$. Although we cannot predict the size of the minimum u''-v''

cut in G/F_1 when we are choosing e, having larger degrees of u' and v' in G/F could have some positive influence toward this goal. This criterion, however, has not been evaluated with respect to this goal: further experimental studies are needed here.

We emphasize that Step 3 above is crucial in allowing us to work on relatively small contractions throughout the entire computation. Note also that, due to this step, the result of $\textsc{Lift}(G, F)$ is not a subset of F in general.

5 The overall algorithm

The upper bound algorithm and the lower bound algorithm, together with the preprocessing algorithm, are combined in the following manner. Fix the graph G given.

1. We compute an almost-clique separator decomposition $\mathcal{A}$ of G using the method in [21].
2. For each bag A_i of $\mathcal{A}$, let G_i denote the graph on A_i obtained from $G[A_i]$ by adding edges of $K(N(C))$ for each component C of $G \setminus A_i$. By Proposition 1, the task of computing $\mathrm{tw}(G)$ reduces to the tasks of computing $\mathrm{tw}(G_i)$ for i, for all nodes i of $\mathcal{A}$. Moreover, G_i is a minor of G.
3. Let i^* be such that $|A_{i^*}| \geq |A_i|$ for every node i of $\mathcal{A}$. The lower bound algorithm works on G_{i^*}. When it finds a new lower bound $\mathrm{tw}(G_{i^*}/F)$ on $\mathrm{tw}(G_{i^*})$, this is also a lower bound on $\mathrm{tw}(G)$ since G_{i^*} is a minor of G; we record this new lower bound $\mathrm{tw}(G_{i^*}/F)$ together with the minor G_{i^*}/F of G certifying it.
4. The upper bound algorithm works on G_i for each i, keeping the current solution Π_i of G_i for each i. After initializing Π_i for each i, we start iterations. In each iteration, we choose i_0 to be i such that the current upper bound $\mathrm{tw}_{\Pi_i}(G_i)$ on $\mathrm{tw}(G_i)$ is the largest and replace Π_{i_0} by $\textsc{Improve}(\Pi_{i_0})$, where the implicit graph it works on is set to G_i. When the maximum of the upper bounds $\mathrm{tw}_{\Pi_i}(G_i)$ decreases, we record the new upper bound on G together with the tree-decomposition of G combining $\mathcal{A}$ with the currently best tree-decomposition of G_i for all nodes i of $\mathcal{A}$.
5. As described in the previous sections, the upper bound algorithm and the lower bound algorithm have access to the current bound computed by each other and terminate when they match.

6 Experiments

We have evaluated our solver by experiments. The computing environment for our experiments is as follows. CPU: Intel Core i7-8700K, 3.70GHz; RAM: 64GB; Operating system: Windows 10Pro, 64bit; Programming language: Java 1.8; JVM: jre1.8.0_271. The maximum heap size is set to 60GB. The solver uses two threads, one for the upper bound and the other for the lower bound, although more threads may be invoked for garbage collection by JVM.

As described in the previous sections, both of the upper and lower bound algorithms use BT dynamic programming for deciding the treewidth, and enumerating the safe separators, of small graphs. Our solver uses an implementation of the semi-PID version of this algorithm [18], which is available at the same github repository [22] in which the entire source code of our solver is posted.

The upper bound computation uses a single sequence of pseudo-random numbers and the lower bound computation uses another independent single sequence. The initial seed is set to a fixed value of 1 for both of these sequences, for the sake of reproducibility. With this setting, our solver can be considered deterministic.

We use the bonus instance set from the exact treewidth track of PACE 2017 algorithm implementation challenge [13]. This set of instances are available at [12]. We quote the note by Holgar Dell, the PACE 2017 organizer, explaining his intention to compile these instances.

> The instance set used in the exact treewidth challenge of PACE 2017 is now considered to be too easy. Therefore, this bonus instance set has been created to offer a fresh and difficult challenge. In particular, solving these instances in five minutes would require a 1000x speed improvement over the best exact treewidth solvers of PACE 2017.

The set consists of 100 instances and their summary, available at [12] in csv format, lists each instance with the time spent for solving it and its exact treewidth if the computation is successful. According to this summary, the exact treewidth is known for 91 of those instances; the remaining 9 instances are unsolved.

Tables 1 and 2 show the list of those 91 solved instances. We number them in the increasing order of the computation time provided in the summary. Columns "n", "m", and "tw" give the number of vertices, the number of edges, and the treewidth, respectively, of each instance.

We see that some of these instances are indeed challenging. Even though they have been solved, they require more than a day to solve. We also note that, to date, no new solvers has been published that have overcome the challenge posed by this instance set.

We have run our solver on these instances with the timeout of 30 minutes. Figure 1 (instance No.1 – No.45) and Figure 2 (instance No.46 – No.91) show the results. In each column representing an instance G, the box with a blue number plots the time for obtaining the best upper bound computed before timeout, where the non-negative number d in the box indicates that this bound is $\mathrm{tw}(G) + d$. Similarly, the box with a red non-positive number d plots the time for obtaining the best lower bound, which is $\mathrm{tw}(G) + d$.

We see from these figures that the bounds computed by our solver are quite tight for most instances. Let us say that the result for instance G is of type (d_1, d_2) if the lower bound (upper bound) obtained by our solver is $\mathrm{tw}(G) + d_1$ ($\mathrm{tw}(G) + d_2$, respectively). Then, the results are of type $(0, 0)$ for 62 instances, $(-1, 0)$ for 20 instances, $(-2, 0)$ for 3 instances, $(-3, 0)$ for one instance, $(0, 1)$ for 4 instances, and $(-2, 1)$ for one instance.

We also see from these figures that the performance of our solver on an instance is not so strongly correlated to the hardness of the instance as measured by the time taken by conventional solvers. For example, of the last 6 instances, each of which took more than a day to solve by the PACE 2017 solvers, 5 are exactly solved by our solver and the remaining one has a result of type $(-1, 0)$. Most of the instances with poorer results, with the gap of 2 or 3 between the upper and lower bounds, occur much earlier in the list.

We have also run our solver on the 9 unsolved instances of the bonus set, with 6 hour timeout. It solved 2 of them and, for other 7 instances, the gap between the upper/lower bounds is 2 for 2 instances, 3 for 4 instances, and 7 for one instance.

The certificates of the lower bounds computed by our algorithm are small and easily verified. For each G, let $\mathrm{lb}(G)$ denote the best lower bound computed by our algorithm before the timeout of 30 minutes and let $\mathrm{nc}(G)$ denote the number of vertices of the certificate for this lower bound. Figures 3 and 4 show $\mathrm{lb}(G)$, $\mathrm{nc}(G)$, and the time to verify the certificate using our implementation of the BT algorithm, for each instance G. The average, the minimum, and the maximum of the ratio $\mathrm{nc}(G)$ / $\mathrm{lb}(G)$ over the 91 solved instances are 2.32, 1.25, and 3.9 respectively. The maximum of $\mathrm{nc}(G)$ over these instances is 49 and the time for verifying each certificate is at most 400 milliseconds.

Table 1 bonus instances with known treewidth (first half).

no.	name	n	m	tw	time
1	Sz512_15127_1.smt2-stp212.gaifman_3	175	593	14	4.74 seconds
2	MD5-32-1.gaifman_4	225	705	12	7.45 seconds
3	Promedas_69_9	133	251	9	11.5 seconds
4	GTFS_VBB_EndeApr_Dez2016.zip_train+metro_12	103	212	11	18.5 seconds
5	Promedas_56_8	155	299	10	28.3 seconds
6	Promedas_48_5	134	278	11	46.1 seconds
7	minxor128.gaifman_2	231	606	4	1.24 minutes
8	Promedas_49_8	184	367	10	1.92 minutes
9	FLA_14	266	423	8	3.06 minutes
10	post-cbmc-aes-d-r2.gaifman_10	263	505	11	4.32 minutes
11	Pedigree_11_7	202	501	14	4.64 minutes
12	countbitsarray04_32.gaifman_10	331	843	13	4.96 minutes
13	mrpp_4x4#8_8.gaifman_3	106	589	24	5.00 minutes
14	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_9	143	303	13	5.24 minutes
15	Promedus_38_15	208	398	10	5.35 minutes
16	Promedas_50_7	175	362	12	7.40 minutes
17	Promedus_34_11	157	289	11	7.61 minutes
18	GTFS_VBB_EndeApr_Dez2016.zip_train+metro_15	124	250	13	7.73 minutes
19	GTFS_VBB_EndeApr_Dez2016.zip_train+metro_14	123	248	13	7.88 minutes
20	Promedas_43_13	197	354	10	8.09 minutes
21	Promedas_46_8	175	318	11	8.36 minutes
22	Promedus_14_9	173	357	12	9.27 minutes
23	jgiraldezlevy.2200.9086.08.40.41.gaifman_2	95	568	34	9.76 minutes
24	modgen-n200-m90860q08c40-14808.gaifman_2	112	686	35	10.0 minutes
25	Promedus_38_14	242	462	10	11.1 minutes
26	Pedigree_11_6	205	503	14	14.9 minutes
27	Promedas_27_8	165	323	12	15.4 minutes
28	Promedas_45_7	159	313	12	17.4 minutes
29	jgiraldezlevy.2200.9086.08.40.46.gaifman_2	105	658	33	20.4 minutes
30	jgiraldezlevy.2200.9086.08.40.22.gaifman_2	111	675	33	22.2 minutes
31	Promedas_25_8	204	378	11	22.5 minutes
32	Pedigree_12_8	217	531	14	22.8 minutes
33	Promedus_34_12	210	389	11	26.1 minutes
34	Promedas_22_6	200	415	12	28.0 minutes
35	aes_24_4_keyfind_5.gaifman_3	104	380	23	31.5 minutes
36	Promedus_18_8	195	411	13	35.2 minutes
37	6s151.gaifman_3	253	634	14	36.8 minutes
38	LKS_15	220	385	10	39.4 minutes
39	Promedas_23_6	253	500	12	45.3 minutes
40	Promedus_28_14	193	351	11	47.8 minutes
41	Promedas_21_9	253	486	11	50.9 minutes
42	Promedas_59_10	209	396	11	56.3 minutes
43	Promedas_60_11	216	387	11	58.2 minutes
44	Promedas_69_10	194	379	12	1.08 hours
45	newton.3.3.i.smt2-stp212.gaifman_2	119	459	19	1.18 hours

Table 2 bonus instances with known treewidth (second half).

no.	name	n	m	tw	time
46	jgiraldezlevy.2200.9086.08.40.167.gaifman_2	92	552	34	1.22 hours
47	Promedus_34_14	188	352	12	1.30 hours
48	modgen-n200-m90860q08c40-22556.gaifman_2	135	855	33	1.33 hours
49	jgiraldezlevy.2200.9086.08.40.93.gaifman_2	100	593	36	1.33 hours
50	Promedas_61_8	156	305	13	1.48 hours
51	Promedas_30_7	164	320	13	1.62 hours
52	FLA_13	280	456	9	1.66 hours
53	am_7_7.shuffled-as.sat03-363.gaifman_6	189	424	14	1.72 hours
54	LKS_13	293	484	9	1.78 hours
55	SAT_dat.k80-24_1_rule_1.gaifman_3	130	698	22	1.83 hours
56	Promedas_28_10	333	605	11	1.84 hours
57	smtlib-qfbv-aigs-lfsr_004_127_112-tseitin.gaifman_6	316	669	13	1.89 hours
58	Promedas_11_7	191	385	13	2.64 hours
59	Promedus_20_13	193	353	12	3.39 hours
60	Pedigree_13_12	264	646	15	3.50 hours
61	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_10	187	385	14	3.64 hours
62	Promedas_22_8	224	441	13	3.77 hours
63	FLA_15	325	522	9	4.68 hours
64	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_15	198	406	14	4.99 hours
65	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_13	190	390	14	5.35 hours
66	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_12	197	404	14	5.46 hours
67	Promedas_63_8	181	374	14	5.53 hours
68	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_11	192	395	14	5.64 hours
69	Pedigree_12_14	284	703	15	6.18 hours
70	Promedus_12_15	293	533	11	6.19 hours
71	Promedus_12_14	272	494	11	6.63 hours
72	Promedas_44_9	276	534	12	6.65 hours
73	Promedas_32_8	238	487	13	7.05 hours
74	NY_13	283	448	9	7.16 hours
75	Promedus_18_10	187	397	14	8.98 hours
76	Promedas_34_8	174	348	14	9.11 hours
77	Promedas_62_9	217	427	13	9.67 hours
78	Promedus_17_13	180	349	13	11.1 hours
79	Promedus_11_15	247	497	13	12.5 hours
80	Promedas_24_11	273	494	12	13.4 hours
81	Promedus_14_8	199	417	14	15.1 hours
82	am_9_9.gaifman_6	212	480	15	15.7 hours
83	NY_11	226	369	10	17.9 hours
84	mrpp_8x8#24_14.gaifman_3	140	856	28	18.5 hours
85	Pedigree_12_10	286	712	16	22.1 hours
86	Promedas_55_9	221	425	13	27.0 hours
87	Pedigree_12_12	277	683	16	27.9 hours
88	Promedas_46_15	227	416	13	28.4 hours
89	Pedigree_13_9	268	665	16	38.9 hours
90	Promedas_51_12	230	431	13	40.2 hours
91	Promedus_27_15	189	353	13	41.6 hours

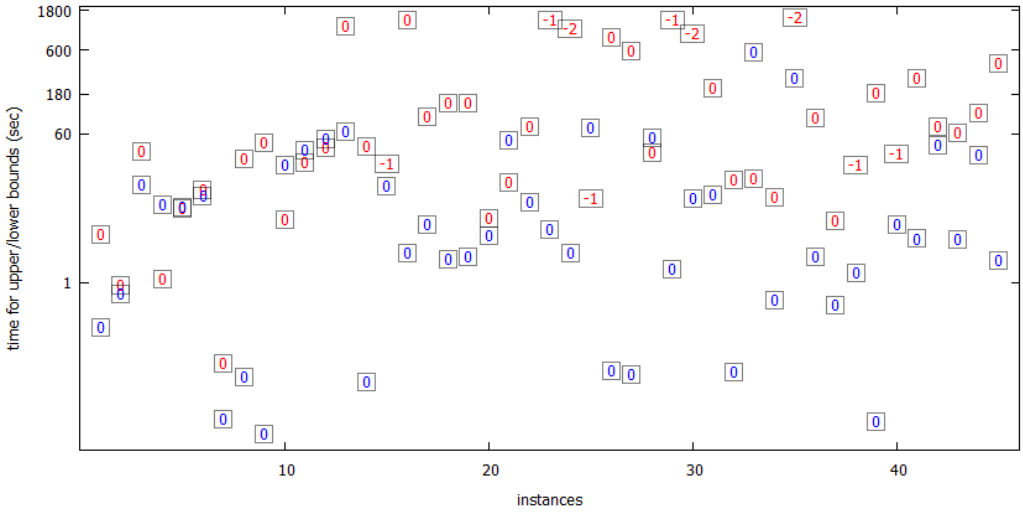


Figure 1 Time for computing upper/lower bounds for instances 1–45.

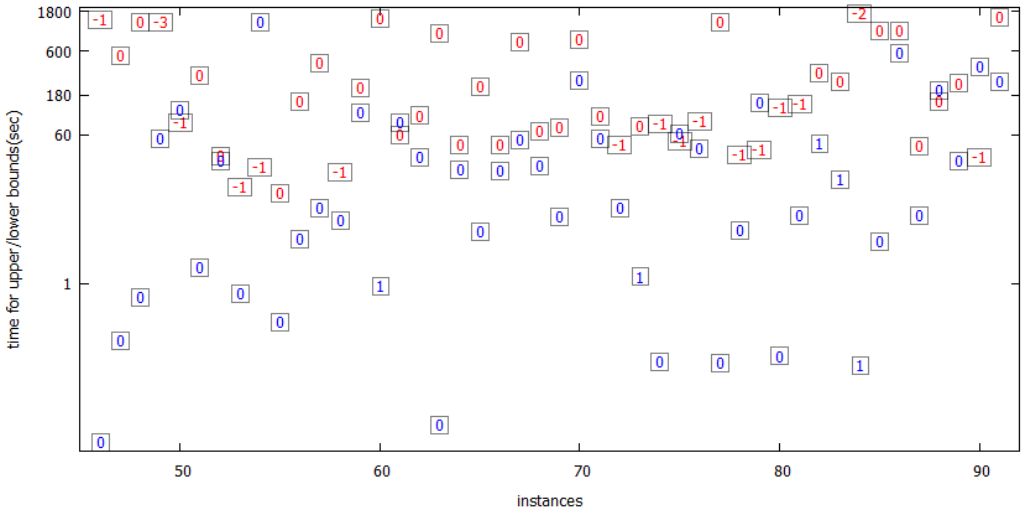


Figure 2 Time for computing upper/lower bounds for instances 46–91.

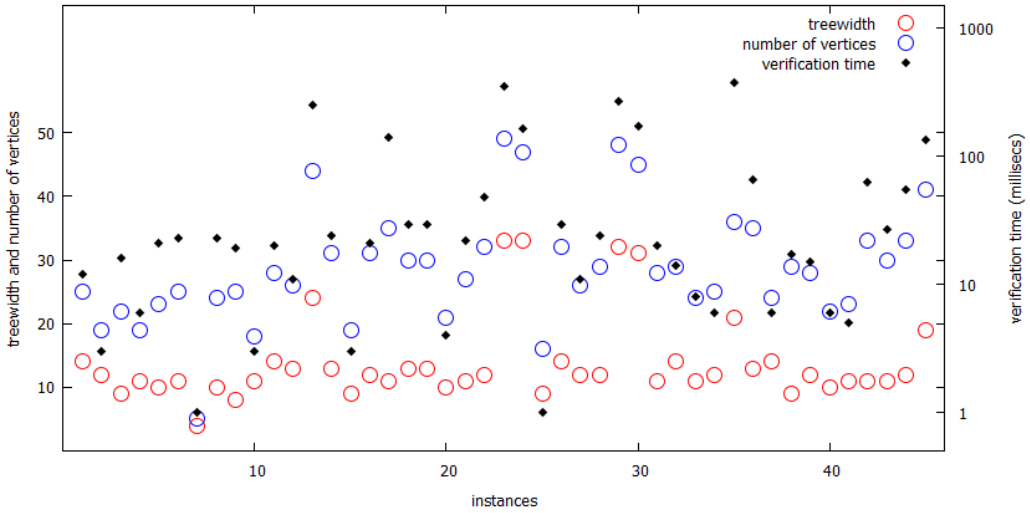


Figure 3 Lower bound certificates for instances 1–45.

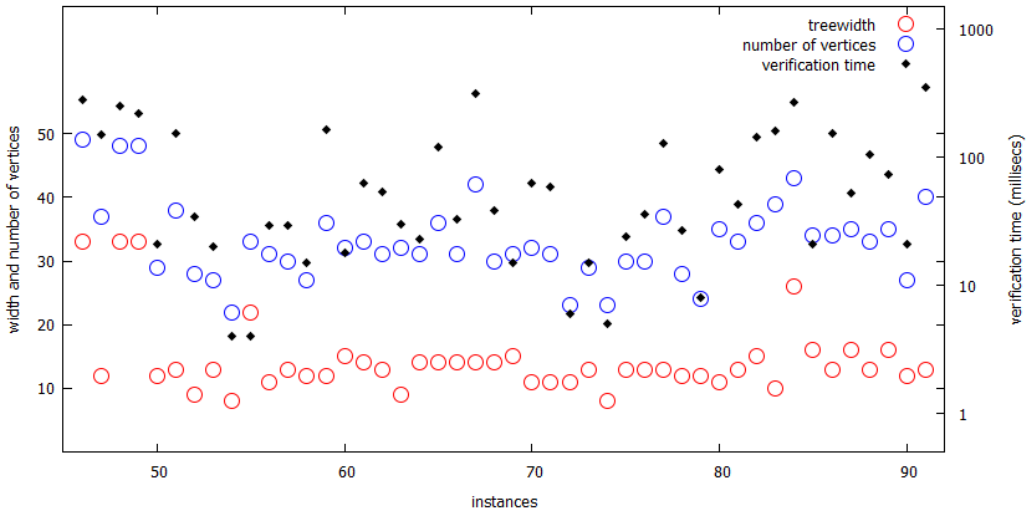


Figure 4 Lower bound certificates for instances 46–91.

7 Conclusions and future work

Our experiments using the bonus instance set from PACE 2017 algorithm implementation challenge have revealed that our approach to treewidth computation is extremely effective in tackling instances that are hard for conventional treewidth solvers. Even when it fails to give the exact treewidth, it produces a lower bound very close to the upper bound. In many applications, such a pair of tight bounds would be satisfactory, since it shows that further search for a better tree-decomposition could only result in a small improvement if at all.

To examine the strength and the weakness of our approach more closely, evaluation on more diverse sets of instances is necessary. For the upper bound part, there are several implementations of heuristic algorithms publicly available, such as the submissions to the heuristic treewidth track of PACE 2017 [11]. Although they are primarily intended for large instances for which exact treewidth appears practically impossible to compute, some of them are nonetheless potential alternatives to our upper bound algorithm. Comparative studies would be needed to determine which algorithm is most suitable for our purposes. On the other hand, it would also be interesting to evaluate our upper bound algorithm on large instances that are the principal targets of those algorithms.

Since our lower approach is new, there are several potential improvements that have not been tried out yet. More work could result in better performances.

We may also ask several theoretical questions regarding our lower bound approach. For example, it would be interesting to ask if the lower bound algorithm can be turned into a fixed parameter tractable algorithm for treewidth. It would also be interesting and useful to identify parameters or structures of graph instances that make them difficult for our lower bound algorithm.

References

1 Ernst Althaus, Daniela Schnurbusch, Julian Wüschner, and Sarah Ziegler. On tamaki's algorithm to compute treewidths. In *19th International Symposium on Experimental Algorithms (SEA 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

2 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.

3 Max Bannach, Sebastian Berndt, and Thorsten Ehlers. Jdrasil: A modular library for computing tree decompositions. In *16th International Symposium on Experimental Algorithms (SEA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

4 Anne Berry, Pinar Heggernes, and Genevieve Simonet. The minimum degree heuristic and the minimal triangulation process. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 58–70. Springer, 2003.

5 Hans L Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1996.

6 Hans L Bodlaender. Treewidth: characterizations, applications, and computations. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 1–14. Springer, 2006.

7 Hans L Bodlaender and Arie MCA Koster. Safe separators for treewidth. *Discrete Mathematics*, 306(3):337–350, 2006.

8 Hans L Bodlaender and Arie MCA Koster. Treewidth computations i. upper bounds. *Information and Computation*, 208(3):259–275, 2010.

9 Hans L Bodlaender and Arie MCA Koster. Treewidth computations ii. lower bounds. *Information and Computation*, 209(7):1103–1119, 2011.

10 Vincent Bouchitté and Ioan Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM Journal on Computing*, 31(1):212–232, 2001.

11 Holgar Dell. PACE-challenge/Treewidth. https://github.com/PACE-challenge/Treewidth, 2017. [github repository, accessed January 12, 2022].

12 Holgar Dell. Treewidth-PACE-2017-bonus-instances. https://github.com/PACE-challenge/Treewidth-PACE-2017-bonus-instances/, 2017. [github repository, accessed January 12, 2022].

13 Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration. In *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

14 Pinar Heggernes. Minimal triangulations of graphs: A survey. *Discrete Mathematics*, 306(3):297–317, 2006.

15 Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

16 Neil Robertson and Paul D Seymour. Graph minors. xiii. the disjoint paths problem. *Journal of combinatorial theory, Series B*, 63(1):65–110, 1995.

17 Neil Robertson and Paul D Seymour. Graph minors. xx. wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2):325–357, 2004.

18 Hisao Tamaki. Computing treewidth via exact and heuristic lists of minimal separators. In *International Symposium on Experimental Algorithms*, pages 219–236. Springer, 2019.

19 Hisao Tamaki. A heuristic use of dynamic programming to upperbound treewidth. *arXiv preprint*, 2019. arXiv:1909.07647.

20 Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. *Journal of Combinatorial Optimization*, 37(4):1283–1311, 2019.

21 Hisao Tamaki. A heuristic for listing almost-clique minimal separators of a graph. *arXiv preprint*, 2021. arXiv:2108.07551.

22 Hisao Tamaki. twalgor/tw. https://github.com/twalgor/, 2022. [github repository].

On the Satisfiability of Smooth Grid CSPs

Vasily Alferov
National Research University Higher School of Economics, Saint Petersburg, Russia

Mateus de Oliveira Oliveira
University of Bergen, Norway

Abstract

Many important NP-hard problems, arising in a wide variety of contexts, can be reduced straightforwardly to the satisfiability problem for CSPs whose underlying graph is a grid. In this work, we push forward the study of grid CSPs by analyzing, from an experimental perspective, a symbolic parameter called smoothness.

More specifically, we implement an algorithm that provably works in polynomial time on grids of polynomial smoothness. Subsequently, we compare our algorithm with standard combinatorial optimization techniques, such as SAT-solving and integer linear programming (ILP). For this comparison, we use a class of grid-CSPs encoding the pigeonhole principle. We demonstrate, empirically, that these CSPs have polynomial smoothness and that our algorithm terminates in polynomial time.

On the other hand, as strong evidence that the grid-like encoding is not destroying the essence of the pigeonhole principle, we show that the standard propositional translation of pigeonhole CSPs remains hard for state-of-the-art SAT solvers, such as minisat and glucose, and even to state-of-the-art integer linear-programming solvers, such as Coin-OR CBC.

2012 ACM Subject Classification Theory of computation → Discrete optimization

Keywords and phrases Grid CSPs, Smoothness, SAT Solving, Linear Programming

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.18

Supplementary Material *Software*: `https://github.com/AutoProving/GridCSPs`
archived at `swh:1:dir:04dab414f88c68d273bc136840c6286e26e0d0b7`

Funding *Mateus de Oliveira Oliveira*: Research Council of Norway, Grant Numbers 288761 and 326537.

1 Introduction

In this work, we consider constraint satisfaction problems (CSPs) where variables are arranged on an $m \times n$-grid, and where the domain of each variable is the set $\{1, \ldots, k\}$. Constraints are local, in the sense that they can only relate pairs of variables that correspond edges of the grid. In our work, these CSPs are called (m, n, k)-grid CSPs, or simply as grid CSPs when the parameters m, n, k are not relevant.

Grid CSPs have a wide variety of applications, ranging from board games to the simulation of Turing machines running for a given number of steps. From a complexity-theoretic perspective, the problem of determining whether a given grid CSP is satisfiable can be solved in polynomial time for $k \leq 2$ by a straightforward reduction to 2-SAT. On the other hand, for $k \geq 3$ the problem becomes NP-complete. Indeed, several natural NP-complete problems reduce straightforwardly to the satisfiability problem for grid CSPs with constant-size domains, including problems arising in the context of pattern recognition [20, 22], image processing [5, 20], tiling systems [15, 21], formal language theory [9, 12, 16, 20], and many others. Since the satisfiability problem for grid-like CSPs is NP-complete, and can be used to model many classes of interesting problems, the identification of subclasses of grid CSPs that can be solved efficiently is a fundamental quest.

We approach this quest by leveraging on the notion of *smoothness*, a complexity measure defined in [6] in the context of the picture satisfiability problem. In particular, we consider a similar notion of smoothness for grid-CSPs and prove analytically that the satisfiability problem for polynomially-smooth instances is solvable in polynomial time. From an experimental perspective, we implement our algorithm and evaluate the performance of our solver on a class of grid-CSPs encoding the pigeonhole-principle, which essentially states that there is no bijection between a set of $n+1$ and a set of n elements. In particular, we confirm experimentally that grid-CSPs encoding the pigeonhole principle has polynomial smoothness. In particular, our algorithm was able to solve both positive instances (that require mapping n pigeons to n holes), and negative instances (that require mapping $n+1$ pigeons to n holes) in time $O(n^3)$.

In an influential work, Haken showed that a family of propositional formulas H_m encoding the pigeonhole principle requires resolution refutations of exponential size [3, 10]. Super-polynomial lower bounds for this principle have been shown for a variety of proof systems, including constant-depth Frege proof systems [13, 14, 19]. It turns out that the formulas arising from Haken's encoding of the pigeonhole principle are also hard in practice to state-of-the-art SAT-solvers based on the technique of conflict-driven clause-learning (CDCL) [7, 17]. Indeed, it has been shown that certain variants of CDCL-based SAT solvers, such as those introduced in [7, 17], are equivalent in power to resolution-based proof systems [2, 4, 11, 18].

We give strong evidence that our encoding of the pigeonhole principle as grid CSPs preserves its inherent difficulty by showing empirically that the most direct propositional translation of our grid CSPs into SAT instances remain hard for two of the most well known SAT solvers based on the CDCL paradigm (minisat [7] and glucose [1]). Going further, we show experimentally that the most straightforward integer linear-programming formulation of our grid CSPs is hard to the state-of-the-art integer linear programming solver Coin-OR CBC solver [8].

Our theoretical and experimental results give strong evidence that the smoothness of grid-CSPs is a parameter that may be valuable in the study of combinatorial problems involving constraints based on the pigeonhole principle.

2 Preliminaries

We denote by $\mathbb{N} \doteq \{0, 1, \ldots\}$ the set of natural numbers (including zero), and by $\mathbb{N}_+ \doteq \mathbb{N} \setminus \{0\}$ the set of positive natural numbers. For each $c \in \mathbb{N}_+$, we let $[c] \doteq \{1, 2, \ldots, c\}$ and $[\![c]\!] \doteq \{0, 1, \ldots, c-1\}$.

Let Σ be an alphabet and $w \in \mathbb{N}_+$. A (Σ, w)-*layer* is a tuple $B \doteq (\ell, r, T, I, F, \iota, \phi)$, where $\ell \subseteq [\![w]\!]$ is a set of *left states*, $r \subseteq [\![w]\!]$ is a set of *right states*, $T \subseteq \ell \times \Sigma \times r$ is a set of *transitions*, $I \subseteq \ell$ is a set of *initial states*, $F \subseteq r$ is a set of *final states* and $\iota, \phi \in \{0, 1\}$ are Boolean flags satisfying the two following conditions:

1. if $\iota = 0$ then $I = \emptyset$;
2. if $\phi = 0$ then $F = \emptyset$.

In what follows, we may write $\ell(B)$, $r(B)$, $T(B)$, $I(B)$, $F(B)$, $\iota(B)$ and $\phi(B)$ to refer to the sets ℓ, r, T, I and F and to the Boolean flags ι and ϕ, respectively.

We let $\mathcal{B}(\Sigma, w)$ denote the set of all (Σ, w)-layers. Note that, $\mathcal{B}(\Sigma, w)$ is non-empty and has at most $2^{\mathcal{O}(|\Sigma| \cdot w^2)}$ elements. Let $n \in \mathbb{N}_+$. A (Σ, w)-*ordered decision diagram* (or simply, (Σ, w)-*ODD*) of *length* n is a string $D \doteq B_1 \cdots B_n \in \mathcal{B}(\Sigma, w)^n$ of length n over the alphabet $\mathcal{B}(\Sigma, w)$ satisfying the following conditions:

1. for each $i \in [n-1]$, $\ell(B_{i+1}) = r(B_i)$;
2. $\iota(B_1) = 1$ and, for each $i \in \{2, \ldots, n\}$, $\iota(B_i) = 0$;
3. $\phi(B_n) = 1$ and, for each $i \in [n-1]$, $\phi(B_i) = 0$.

We note that Condition 2 guarantees that only the first layer of an ODD is allowed to have initial states. Analogously, Condition 3 guarantees that only the last layer of an ODD is allowed to have final states.

The *size* of an ODD $D = B_1 \ldots B_n$ is defined as $\mathsf{size}(D) = |\ell(B_1)| + \sum_{j=1}^{n} |r(B_j)|$. For each $n \in \mathbb{N}_+$, we denote by $\mathcal{B}(\Sigma, w)^{\circ n}$ the set of all (Σ, w)-ODDs of length n. The *width* of an ODD $D = B_1 \cdots B_n \in \mathcal{B}(\Sigma, w)^{\circ n}$ is defined as $\mathsf{w}(D) \doteq \max\{|\ell(B_1)|, \ldots, |\ell(B_n)|, |r(B_n)|\}$. We remark that $\mathsf{w}(D) \leq w$.

Let $D = B_1 \cdots B_n \in \mathcal{B}(\Sigma, w)^{\circ n}$ and $s = \sigma_1 \cdots \sigma_n \in \Sigma^n$. A *valid sequence* for s in D is a sequence of transitions $\langle(\mathfrak{p}_1, \sigma_1, \mathfrak{q}_1), \ldots, (\mathfrak{p}_n, \sigma_n, \mathfrak{q}_n)\rangle$ such that $\mathfrak{p}_{i+1} = \mathfrak{q}_i$ for each $i \in [n-1]$, and $(\mathfrak{p}_i, \sigma_i, \mathfrak{q}_i) \in T(B_i)$ for each $i \in [n]$. Such a valid sequence is called *accepting* for s if, additionally, $\mathfrak{p}_1 \in I(B_1)$ and $\mathfrak{q}_n \in F(B_n)$. We say that D *accepts* s if there exists an accepting sequence for s in D. The language of D, denoted by $\mathcal{L}(D)$, is defined as the set of all strings accepted by D, i.e. $\mathcal{L}(D) \doteq \{s \in \Sigma^n : s \text{ is accepted by } D\}$.

A (Σ, w)-layer B is called *deterministic* if the following conditions are satisfied:

1. for each $\mathfrak{p} \in \ell(B)$ and each $\sigma \in \Sigma$, there exists at most one right state $\mathfrak{q} \in r(B)$ such that $(\mathfrak{p}, \sigma, \mathfrak{q}) \in T(B)$;
2. if $\iota(B) = 1$, then $I(B) = \ell(B)$ and $|\ell(B)| = 1$.

On the other hand, a (Σ, w)-layer B is called *complete* if, for each $\mathfrak{p} \in \ell(B)$ and each $\sigma \in \Sigma$, there exists at least one right state $\mathfrak{q} \in r(B)$ such that $(\mathfrak{p}, \sigma, \mathfrak{q}) \in T(B)$. We let $\widehat{\mathcal{B}}(\Sigma, w)$ be the subset of $\mathcal{B}(\Sigma, w)$ comprising all deterministic, complete (Σ, w)-layers.

An ODD $D = B_1 \cdots B_n \in \mathcal{B}(\Sigma, w)^{\circ n}$ is called *deterministic* (*complete*, resp.) if, for each $i \in [n]$, B_i is a deterministic (complete, resp.) layer. We remark that, if D is deterministic, then there exists at most one valid sequence in D for each string in Σ^n. On the other hand, if D is complete, then there exists at least one valid sequence in D for each string in Σ^n. For each $n \in \mathbb{N}_+$, we let $\widehat{\mathcal{B}}(\Sigma, w)^{\circ n}$ be the subset of $\mathcal{B}(\Sigma, w)^{\circ n}$ comprising all deterministic, complete (Σ, w)-ODDs of length n.

We say that an ODD $D = B_1 \ldots B_n$ in $\mathcal{B}([k], w)^{\circ n}$ has non-determisitic degree d if all, but at most one, layers of D are deterministic. Additionally, if there is a j such that B_j is not deterministic, then for each state $q \in \ell(B_j)$, and each symbol $\sigma \in [k]$, there is at most d states $q' \in r(B_j)$ such that $(q, \sigma, q') \in T(B_j)$.

The following lemma can be proved by applying the standard power set construction followed by a standard minimization algorithm for ODDs, and by observing that only subsets of size at most d belonging to each frontier are relevant.

▶ **Lemma 1.** *Let D be an ODD in $\mathcal{B}([k], w)^{\circ n}$ be an ODD of nondeterministic degree d. Then one can construct in time $n \cdot w^{O(d)}$ a deterministic ODD D' with minimum number of states with the property that $\mathcal{L}(D') = \mathcal{L}(D)$.*

We will also need the following lemma stating that ODD representatives for synchronized products of languages accepted by two given ODDs can be computed efficiently.

▶ **Lemma 2** (Synchronized Product of Automata)**.** *Let D and D' be ODDs in $\mathcal{B}([k], k)^{\circ n}$. Let $V = \{V_1, \ldots, V_n\}$ be a set where for each $i \in [n]$, $V_i \subseteq [k] \times [k]$. Then one can construct in time $O(k^2 \cdot n)$ an ODD $D \otimes_V D'$ accepting the following language over $[k] \times [k]$.*

$$\mathcal{L}(D \otimes_V D') = \{(\sigma_1, \sigma_1') \ldots (\sigma_n, \sigma_n') \mid \sigma_1 \ldots \sigma_n \in \mathcal{L}(D),\ \sigma_1' \ldots \sigma_n' \in \mathcal{L}(D'), (\sigma_i, \sigma_i') \in V_i\}.$$

3 Smooth Grid Constraint Satisfaction Problems

Let m, n and k be positive integers. An (m, n, k)-*grid CSP* is specified by a pair $(\mathcal{V}, \mathcal{H})$ where $\mathcal{V} = \{V_{i,j}\}_{i\in[m-1],j\in[n]}$ is a collection of sets $V_{i,j} \subseteq [k] \times [k]$ called *local vertical constraints*, and $\mathcal{H} = \{H_{i,j}\}_{i\in[m],j\in[n-1]}$ is a collection of sets $H_{i,j} \subseteq [k] \times [k]$ called *local horizontal constraints*. A *solution* for $(\mathcal{V}, \mathcal{H})$ is an $m \times n$ matrix $M \in [k]^{m\times n}$ which satisfies all local vertical and horizontal constraints. More precisely, such a solution M satisfies the following conditions.

1. For each $(i, j) \in [m-1] \times [n]$, the pair $(M_{i,j}, M_{i+1,j})$ belongs to $V_{i,j}$.
2. For each $(i, j) \in [m] \times [n-1]$, the pair $(M_{i,j}, M_{i,j+1})$ belongs to $H_{i,j}$.

Let m and n be positive integers. We endow the set $[m] \times [n] = \{(i, j) \mid i \in [m], j \in [n]\}$ with a *lexicographic order* $<$ that sets $(i, j) < (i', j')$ if either $i < i'$, or $i = i'$ and $j < j'$. We write $(i, j) \leq (i', j')$ to denote that $(i, j) = (i', j')$ or $(i, j) < (i', j')$. For each $(i, j) \in [m] \times [n]$, we let

$$S(m, n, i, j) = \{(i', j') \in [m] \times [n] \ : \ (i', j') \leq (i, j)\}$$

be the set of all positions in $[m] \times [n]$ that are (lexicographically) smaller than or equal to (i, j). Given (m, n, k)-grid CSP $(\mathcal{V}, \mathcal{H})$, we say that a function $M : S(m, n, i, j) \to [k]$ is an (i, j)-partial $(\mathcal{V}, \mathcal{H})$-solution if the following conditions are satisfied.

1. $(M_{i',j'}, M_{i'+1,j'}) \in V_{i',j'}$ for each $(i', j'),\ (i' + 1, j')$ in $S(m, n, i, j)$.
2. $(M_{i',j'}, M_{i',j'+1}) \in H_{i',j'}$ for each $(i', j'),\ (i', j' + 1)$ in $S(m, n, i, j)$.

Note that for simplicity, we write $M_{i,j}$ in place of $M(i, j)$ to designate an entry of M. Intuitively, an (i, j)-partial $(\mathcal{V}, \mathcal{H})$-solution for N is a function that colors the positions of N up to the entry (i, j) with elements from Σ in such a way that the vertical and horizontal constraints imposed by $\mathcal{V}$ and $\mathcal{H}$ respectively are respected. If (i, j_1) and (i, j_2) are positions in $S(m, n, i, j)$ with $j_1 < j_2$, then we let $M_{i,[j_1,j_2]} = M_{i,j_1}...M_{i,j_2}$ be the string over $[k]$ formed by all entries at the i-th row of M between positions j_1 and j_2. Now let $(i, j) \in S(m, n, i, j)$ with $(i, j) \geq (1, n)$. The (i, j)-boundary of M is defined as follows.

$$\partial_{i,j}(M) = \begin{cases} M_{i,[1,n]} & \text{if } j = n. \\ \\ M_{i,[1,j]} \cdot M_{i-1,[j+1,n]} & \text{if } j < n. \end{cases} \tag{1}$$

In other words, if $j = n$, then $\partial(M)$ is the string consisting of all entries in the i-th row of M. On the other hand, if $j < n$, then $\partial(M)$ is obtained by concatenating the string corresponding to the first j entries of row i with the last $(n - j)$ entries of row $(i - 1)$. The notion of boundary of a partial solution is illustrated in Figure 1.

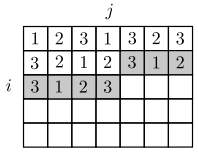


Figure 1 An (i, j)-partial solution M where $i = 3$ and $j = 4$. The grey entries form the boundary of M. Therefore $\partial_{i,j}(M) = 3123312$.

The (i, j)-feasibility boundary of $(\mathcal{V}, \mathcal{H})$, denoted by $\partial_{i,j}(\mathcal{V}, \mathcal{H})$, is defined as follows.

$$\partial_{i,j}(\mathcal{V}, \mathcal{H}) = \{\partial_{i,j}(M) \mid M \text{ is an } (i, j)\text{-partial } (\mathcal{V}, \mathcal{H})\text{-solution}\}. \tag{2}$$

Note that a $(\mathcal{V}, \mathcal{H})$-solution exists if and only if $\partial_{m,n}(\mathcal{V}, \mathcal{H})$ is non-empty.

▶ **Definition 3.** *We say that an (m, n, k)-grid CSP $(\mathcal{V}, \mathcal{H})$ is s-smooth if for each $i \in [m]$ and each $j \in [n]$, there is a deterministic ODD D of size at most s such that $\mathcal{L}(D) = \partial_{i,j}(\mathcal{V}, \mathcal{H})$.*

In [6] a similar notion of smoothness has been defined in the context of the picture satisfiability problem. The crucial difference is that our grid CSPs are a much more general combinatorial object, since the vertical constraints $V_{i,j}$ and the horizontal constraints $H_{i,j}$ may depend on the position (i, j), whereas in the context of pictures, all vertical (horizontal) constraints are required to be identical over the whole grid. The main result from [6] established that the satisfiability problem for pictures of polynomial smoothness is solvable in polynomial time. In this section, we generalize this result to the context of general grid CSPs.

The following lemma states that given an (m, n, k)-grid CSP $(\mathcal{V}, \mathcal{H})$, one can construct and initial ODD $D(\mathcal{H}, 1) \in \mathcal{B}([k], k)^{\circ n}$ accepting precisely those strings in $[k]^n$ that satisfy all local horizontal constraints in the first row of $\mathcal{H}$.

▶ **Lemma 4.** *Let $(\mathcal{V}, \mathcal{H})$ be an (m, n, k)-grid CSP. There is a deterministic ODD $D(\mathcal{H}, 1) \in \mathcal{B}([k], k)^{\circ n}$ such that*

$$\mathcal{L}(D(\mathcal{H}, 1)) = \{\sigma_1 \dots \sigma_n \in [k]^n \ : \ (\sigma_j, \sigma_{j+1}) \in H_{1,j} \text{ for each } j \in [n-1]\}$$

Proof. We let $D(\mathcal{H}, 1) = B_1 \dots B_n$ be the ODD in $\mathcal{B}([k], k)^{\circ n}$ where

1. $\ell(B_j) = r(B_j) = [k]$ for each $j \in [n]$,
2. $I(B_1) = \{1\}$ and $I(B_j) = \emptyset$ for each $j \in \{2, \dots, n\}$,
3. $F(B_n) = [k]$ and $F(B_j) = \emptyset$ for each $j \in \{1, \dots, n-1\}$,
4. $T(B_j) = \{(\sigma, \sigma', \sigma') \ : \ (\sigma, \sigma') \in H_{1,j}\}$.

Then it should be clear that a string $s = \sigma_1 \sigma_2 \dots \sigma_n$ belongs to $\mathcal{L}(D(\mathcal{H}, 1))$ if and only if

$$\langle (\sigma_1, \sigma_2, \sigma_2) \dots (\sigma_{n-1}, \sigma_n, \sigma_n) \rangle$$

is an accepting sequence for s in $D(\mathcal{H}, 1)$. By construction, this happens if and only if $(\sigma_j, \sigma_{j+1}) \in H_{1,j}$ for each $j \in [n-1]$. ◀

▶ **Lemma 5.** *Let k and w be a positive integers, $V \subseteq [k] \times [k]$ and D be a deterministic ODD in $\mathcal{B}([k], w)^{\circ n}$. Then one can construct in time $O(n \cdot w^k)$ an ODD $\mathsf{Up}(D, V, 1)$ accepting the following language*

$$\mathcal{L}(\mathsf{Up}(D, V, 1)) = \{aw \ : \ \exists b \in [k],\ bw \in \mathcal{L}(D),\ (b, a) \in V\}.$$

Proof. Let k and w be positive integers, B be a layer in $\mathcal{B}([k], w)$ and $V \subseteq [k] \times [k]$. We let $\mathfrak{l}(B, V)$ be the layer in $\mathcal{B}([k], k \cdot w)$ defined as follows.

1. $\ell(\mathfrak{l}(B, V)) = \ell(B)$ and $I(\mathfrak{l}(B, V)) = I(B)$.
2. $\iota(\mathfrak{l}(B, V)) = \iota(B)$ and $\phi(\mathfrak{l}(B, V)) = \phi(B)$.
3. $r(\mathfrak{l}(B, V)) = \{j \cdot w + j' \ : \ j \in r(B),\ j' \in [\![k]\!]\}$.
4. $F(\mathfrak{l}(B, V)) = \{j \cdot w + j' \ : \ j \in F(B), j' \in [\![k]\!]\}$.
5. $T(\mathfrak{l}(B, V)) = \{(i, b, j \cdot w + b) \ : \ \exists a \in [k], (i, a, j) \in T(B), (a, b) \in V\}$.

Additionally, we let $\mathfrak{r}(B, V)$ be the layer in $\mathcal{B}([k], k \cdot w)$ defined as follows.

1. $r(\mathfrak{r}(B, V)) = r(B)$ and $F(\mathfrak{r}(B, V)) = F(B)$.
2. $\iota(\mathfrak{r}(B, V)) = \iota(B)$ and $\phi(\mathfrak{r}(B, V)) = \phi(B)$.

3. $\ell(\mathfrak{r}(B,V)) = \{i \cdot w + i' \ : \ i \in \ell(B),\ i' \in [\![k]\!]\}$
4. $I(\mathfrak{r}(B,V)) = \emptyset$.
5. $T(\mathfrak{r}(B,V)) = \{(i \cdot w + i', a, j) \ : \ \exists a \in [k], (i,a,j) \in T(B)\}$.

Now, let $D = B_1 \dots B_n$ be a deterministic an ODD in $\mathcal{B}([k],w)^{\circ n}$ and let

$$D' = \mathfrak{l}(B_1,V)\mathfrak{r}(B_2,V)B_3 \dots B_n$$

be the ODD obtained from D by replacing B_1 with $\mathfrak{l}(B_1,V)$ and B_2 with $\mathfrak{r}(B_2,V)$. Then one can verify that

$$\mathcal{L}(D') = \{aw \ : \ \exists b \in [k],\ bw \in \mathcal{L}(D),\ (b,a) \in V\}.$$

Additionally, D' has non-deterministic degree at most k. Now we set the ODD $\mathsf{Up}(D,V,1)$ as the minimum deterministic ODD such that $\mathcal{L}(\mathsf{Up}(D,V,1)) = \mathcal{L}(D')$. Since D' has non-deterministic degree at most k, we have that $\mathsf{Up}(D,V,1)$ can be constructed in time $O(n \cdot w^k)$. ◀

▶ **Lemma 6.** *Let k and w be a positive integers, $V, H \subseteq [k] \times [k]$, D be a deterministic ODD in $\mathcal{B}([k],w)^{\circ n}$, and $j \in [n-1]$. Then one can construct in time $O(n \cdot w^k)$ an ODD $\mathsf{Up}(D,V,H,j)$ accepting the following language*

$$\mathcal{L}(\mathsf{Up}(D,V,H,j)) = \{ubav \ : \ \exists c \in [k], ubcv \in \mathcal{L}(D),\ |ub| = j-1,\ (b,a) \in H,\ \textit{and}\ (c,a) \in V\}.$$

Proof. Let k and w be positive integers, B be a layer in $\mathcal{B}([k],w)$ and $V, H \subseteq [k] \times [k]$. We let $\mathfrak{l}(B,V,H)$ be the layer in $\mathcal{B}([k], k \cdot w)$ defined as follows.

1. $\ell(\mathfrak{l}(B,V,H)) = \ell(B)$ and $I(\mathfrak{l}(B,V,H)) = I(B)$.
2. $\iota(\mathfrak{l}(B,V,H)) = \iota(B)$ and $\phi(\mathfrak{l}(B,V,H)) = \phi(B)$.
3. $r(\mathfrak{l}(B,V,H)) = \{j \cdot w + j' \ : \ j \in r(B),\ j' \in [\![k]\!]\}$.
4. $F(\mathfrak{l}(B,V,H)) = \{j \cdot w + j' \ : \ j \in F(B), j' \in [\![k]\!]\}$.
5. $T(\mathfrak{l}(B,V,H)) = \{(i,b,j \cdot w + b) \ : \ \exists a \in [k], (i,a,j) \in T(B), (a,b) \in V\}$.

Additionally, we let $\mathfrak{r}(B,V,H)$ be the layer in $\mathcal{B}([k], k \cdot w)$ defined as follows.

1. $r(\mathfrak{r}(B,V,H)) = r(B)$ and $F(\mathfrak{r}(B,V,H)) = F(B)$.
2. $\iota(\mathfrak{r}(B,V,H)) = \iota(B)$ and $\phi(\mathfrak{r}(B,V,H)) = \phi(B)$.
3. $\ell(\mathfrak{r}(B,V,H)) = \{i \cdot w + i' \ : \ i \in \ell(B),\ i' \in [\![k]\!]\}$
4. $I(\mathfrak{r}(B,V,H)) = \emptyset$.
5. $T(\mathfrak{r}(B,V,H)) = \{(i \cdot w + i', a, j) \ : \ \exists a \in [k], (i,a,j) \in T(B)\}$.

Now, let $D = B_1 \dots B_n$ be a deterministic an ODD in $\mathcal{B}([k],w)^{\circ n}$ and $j \in [n-1]$. Let

$$D' = B_1 \dots \mathfrak{l}(B_j,V,H)\mathfrak{r}(B_{j+1},V,H) \dots B_n$$

be the ODD obtained from D by replacing B_j with $\mathfrak{l}(B_j,V,H)$ and B_{j+1} with $\mathfrak{r}(B_{j+1},V,H)$. Then one can verify that

$$\mathcal{L}(D') = \{ubav \ : \ \exists c \in [k], ubcv \in \mathcal{L}(D),\ |ub| = j-1,\ (b,a) \in H,\ \text{and}\ (c,a) \in V\}.$$

Additionally, D' has non-deterministic degree at most k. Now we set the ODD $\mathsf{Up}(D,V,H,j)$ as the minimum deterministic ODD such that $\mathcal{L}(\mathsf{Up}(D,V,H,j)) = \mathcal{L}(D')$. Since D' has non-deterministic degree at most k, we have that $\mathsf{Up}(D,V,H,j)$ can be constructed in time $O(n \cdot w^k)$. ◀

The following corollary is a consequence of Lemmas 5 and 6.

▶ **Corollary 7.** *Let $(\mathcal{V},\mathcal{H})$ be an (m,n,k)-grid CSP, and let D be an ODD in $\mathcal{B}([k],w)^{\circ n}$.*

1. *If $\mathcal{L}(D)=\partial_{i,n}(\mathcal{V},\mathcal{H})$ for some $i\in[m-1]$, then*

$$\mathcal{L}(\mathsf{Up}(D,V_{i+1,1},1))=\partial_{i+1,1}(\mathcal{V},\mathcal{H}).$$

2. *If $\mathcal{L}(D)=\partial_{i,j}(\mathcal{V},\mathcal{H})$ for some $i\in[m]$ and some $j\in[n-1]$, then*

$$\mathcal{L}(\mathsf{Up}(D,V_{i,j+1},H_{i,j+1},j+1))=\partial_{i,j+1}(\mathcal{V},\mathcal{H}).$$

Algorithm 1 Decision algorithm for grid-like CSPs.

Data: An (m,n,k)-grid CSP $(\mathcal{V},\mathcal{H})$.
Result: Yes if $(\mathcal{V},\mathcal{H})$ is satisfiable. No otherwise.
$D_1 \leftarrow D(\mathcal{H},1)$;
for $i=2\dots m$ **do**
  $D_i \leftarrow \mathsf{Up}(D_{i-1},V_{i-1,1},1)$;
  for $j=2\dots n$ **do**
    $D_i \leftarrow \mathsf{Up}(D_i,V_{i-1,j},H_{i,j},j)$;
  end
end
Return Yes if $\mathcal{L}(D_m)\neq\emptyset$ and No otherwise.

The algorithm described in Algorithm 1 can be used to determine whether a given (m,n,k)-grid like CSP (V,H) is satisfiable. It turns out that the sequence of ODDs $D_1,...,D_m$ can be used to construct an actual solution in case it exists. The description of the construction is given below in Algorithm 2.

Algorithm 2 Construction of a solution of a satisfiable grid-like CSP.

Data: ODDs $D_1,\dots,D_m$ constructed in Algorithm 1 where $\mathcal{L}(D_m)\neq\emptyset$.
Result: A solution for the (m,n,k)-grid like CSP (V,H) input to Algorithm 1.
Let s be a string in $\mathcal{L}(D_m)$. ;
$M_m \leftarrow s$;
for $i=m-1\dots 1$ **do**
  Let $s\in\mathcal{L}(D_i)$ be such that $s\otimes_V M_{i+1}$ belongs to $\mathcal{L}(D_i\otimes_V D_{i+1})$.;
  $M_i \leftarrow s$;
end
Return the (m,n,k)-matrix M whose rows are $M_1,\dots,M_m$. ;

From Corollary 7 and Algorithms 1 and 2, we infer the following theorem.

▶ **Theorem 8.** *Let $(\mathcal{V},\mathcal{H})$ be an s-smooth (m,n,k)-grid CSP. Then one can determine whether $(\mathcal{V},\mathcal{H})$ has a solution using Algorithm 1 in time $s^{O(k)}\cdot m^{O(1)}\cdot n^{O(1)}$. In case a solution exists, it can be constructed within the same time bounds using Algorithm 2.*

4 Experiments

In this section, we evaluate the performance of our algorithm and compare it with two general-purpose SAT solvers (minisat and glucose) and the integer-programming solver Coin-OR CBC.

We start by defining the notion of Pigeonhole grid-CSPs, a CSP with uniform vertical and horizontal constraints over an alphabet of size 5 encoding the pigeonhole principle. A more contrieved version of this CSP was studied in [6], under the name of *pigeonhole pictures*. It was proved in [6] that these pigeonhole pictures have polynomial smoothness, and that the straightforward propositional translation of these CSPs is hard for the bounded-depth Frege proof system.

In this section, we defined a simpler notion of pigeonhole grid CSP than the one employed in [6]. The simplicity of our definition is due to the fact that in our setting local constraints may vary according to the position in the grid, while in [6] local constraints were required to be uniform. Both the fact that these CSPs have polynomial smoothness and the fact that they are hard to bounded Frege are inherited from the corresponding results in [6]. In this section, we confirm empirically both of these theoretical results. Additionally, we show empirically that the straightforward integer-programming formulation of the Pigeonhole grid-CSP is hard for state-of-the-art integer programming tools such as Cplex and Coin-OR CBC.

4.1 The Pigeonhole Grid CSP

For each two positive integers m and n we define an $(m, n, 5)$-grid CSP

$$\mathsf{PHP}(m, n) = (\mathcal{V}(m, n), \mathcal{H}(m, n))$$

encoding the principle that m pigeons are placed into n holes. Clearly such a CSP should be satisfiable if and only if $m \leq n$. To make the definition more intuitive, instead of defining the local vertical and horizontal constraints as subsets of $[5] \times [5]$, we let $\Sigma = \{bb, bg, gb, gg, rr\}$ and assume that these local constraints are defined as subsets of $\Sigma \times \Sigma$. First, for each $i \in [m-1]$ and any $j \in [n]$, we let the local vertical constraint $V_{i,j}$ be equal to the following relation.

$$V = \{\ (xb, yb),\ (xb, rr),\ (rr, xg),\ (xg, yg)\ \mid\ x, y \in \{b, g\}\ \}.$$

Now, there are three types of local horizontal constraints. For $i \in [m]$, we set $H_{i,1}$ equal to the relation

$$H_{left} = \{\ (bx, by),\ (bx, rr),\ (rr, gy)\ :\ x, y \in \{b, g\}\ \}. \tag{3}$$

For each $i \in [m]$ and each $j \in [n-2]$, we set $H_{i,j}$ equal to the relation.

$$H_{middle} = \{\ (bx, by),\ (bx, rr),\ (rr, gy),\ (gx, gy)\ \ :\ \ x, y \in \{b, g\}\ \} \tag{4}$$

Finally, for each $i \in [m]$, we set $H_{i,n-1}$ equal to the relation

$$H_{right} = \{\ (bx, rr),\ (rr, gy),\ (gx, gy)\ \ :\ \ x, y \in \{b, g\}\ \}. \tag{5}$$

Intuitively, if M is a solution for $\mathsf{PHP}(m, n)$ then M has one row for each pigeon and one column for each hole. The $M_{i,j} = rr$ indicates that the i-th pigeon is placed at the hole j. On the other hand, $M_{i,j} = bx$ for some $x \in \{g, b\}$ indicates that the i-th pigeon is placed in some hole greater than j, while $M_{i,j} = gx$ for some $x \in \{b, g\}$ indicates that the i-th pigeon is placed in some hole smaller than j. Analogously, if $M_{i,j} = xb$ for some $x \in \{b, g\}$, then the pigeon that is placed at the j-th hole is greater than i, while if $M_{i,j} = xg$, then the pigeon that is placed at the j-th hole is smaller than i.

Note that the way in which the horizontal constraints are defined guarantees that exactly one pigeon must occur in each row of a solution. This is because there is no allowed pair $(xy, x'y')$ where x is blue and x' is green. Therefore in a solution, each row must have at least one entry with value rr. Additionally, the vertical constraints guarantee that that at most one pigeon will occur in each column. Indeed, if some pigeon occurs in a position (i, j) then the second color in each entry below (i, j) must be green, while the second color of each entry above (i, j) must be blue. Therefore no two pigeons are allowed to appear on the same column of a satisfying assignment. In Figure 2 we depict a solution to the pigeonhole CSP $\mathsf{PHP}(4, 4)$, while one can readily check that the CSP $\mathsf{PHP}(4, 3)$ has no solution.

i)

bb	rr	gb	gb
bb	bg	bb	rr
bb	bg	rr	gg
rr	gg	gg	gg

ii)

rr	gb	gb
bg	bb	rr
bg	rr	gg
bg	bg	

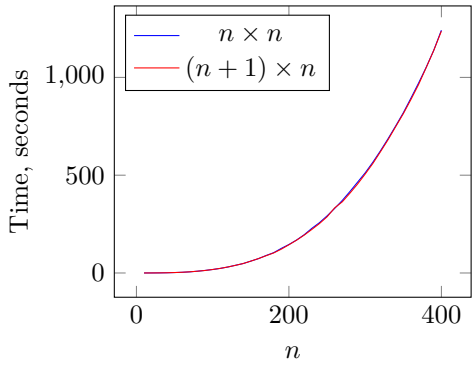
Figure 2 *i)* A solution to the pigeonhole CSP $\mathsf{PHP}(4, 4)$. *ii)* A maximal partial solution to $\mathsf{PHP}(4, 3)$. In this last case, it is not possible to assign a value to the entry $(4, 3)$ of the matrix in such a way that both the constraints $V_{3,3}$ and $H_{4,2}$ are satisfied.

▶ **Theorem 9** (Follow from results in [6]). *Let* $\mathsf{PHP}(m, n)$ *be the pigeonhole grid CSP defined above.*

1. $\mathsf{PHP}(m, n)$ *has a solution if and only if* $m \leq n$.
2. *The smoothness of* PHP *is bounded by* $m^{O(1)} \cdot n$.
3. *For each fixed* $d \in \mathbb{N}$, $\mathsf{PHP}(m + 1, m)$ *require depth-*d *Frege proofs of superpolynomial size.*

4.2 Solving Pigeonhole Pictures with the ODD solver

Since the $\mathsf{PHP}(m, n)$ has smoothness upper bounded by $m^{O(1)} \cdot n$, there is no visible difference between the performance of the solver on barely-satisfiable instances $\mathsf{PHP}(m, m)$ and the performance of the solver on barely-unsatisfiable instances $\mathsf{PHP}(m + 1, m)$. This is confirmed empirically in Figure 3.



n	$\mathsf{w}(D)$	$m \times m$, s	$(m+1) \times m$, s
50	51	2.172	2.220
100	101	17.834	17.780
150	151	60.715	60.552
200	201	145.587	144.179
250	251	291.692	286.952
300	301	511.754	505.581
350	351	818.724	813.929
400	401	1241.258	1236.734

Figure 3 (a) Performance of the ODD solver on $\mathsf{PHP}(m, m)$ and on $\mathsf{PHP}(m + 1, m)$. The running times grows as $O(m^3)$. The plots corresponding to both cases almost match. (b) Execution times (in seconds) and maximum width of minimized deterministic ODDs occurring during the execution of the ODD algorithm. Note that the maximum width is identical in both test cases.

4.3 Experiments with SAT Solvers

First we describe the straightforward translation from (m, n, k)-grid CSPs to CNFs. We note that the obtained CNFs have width at most k. Given such a CSP $(\mathcal{V}, \mathcal{H})$, the formula $\psi(\mathcal{V}, \mathcal{H})$ has a variable $x_{ij\sigma}$ for each $(i, j) \in [m] \times [n]$, and each $\sigma \in [k]$. Intuitively, the variable $x_{ij\sigma}$ is true if the position (i, j) of a solution is set to σ. The following set of clauses specifies that in a satisfying assignment, precisely value is associated with entry (i, j).

$$\text{OneSymbol}(M, i, j) \equiv \bigvee_{s \in [k]} x_{ijs} \quad \wedge \bigwedge_{s, s' \in [k], s \neq s'} (\overline{x}_{ijs} \vee \overline{x}_{ijs'}) \tag{6}$$

The next set of clauses expresses the fact that no pair $(\sigma, \sigma') \notin H_{i,j}$ occurs in consecutive horizontal positions at row i.

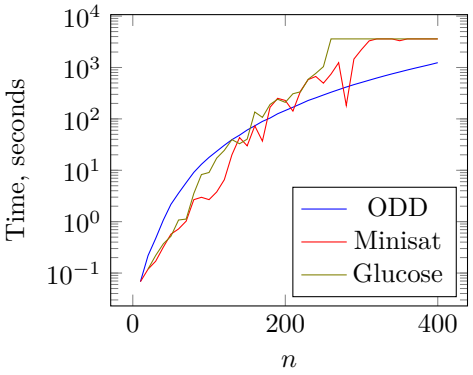
$$\text{Horizontal}(M, i) \equiv \bigwedge_{(\sigma,\sigma') \notin H_{i,j}, j \in [n-1]\}} (\overline{x}_{ij\sigma} \vee \overline{x}_{i(j+1)\sigma'}) \tag{7}$$

Similarly, the following set of of clauses expresses the fact that that no pair $(\sigma, \sigma') \notin V_{i,j}$ occurs in consecutive vertical positions at column j.

$$\text{Vertical}(M, j) \equiv \bigwedge_{(\sigma,\sigma') \notin V_{i,j}, i \in [m-1]\}} (\overline{x}_{ij\sigma} \vee \overline{x}_{(i+1)j\sigma'}) \tag{8}$$

Finally, we set the formula $\psi(M)$ as follows.

$$\psi(\mathcal{V}, \mathcal{H}) \equiv \bigwedge_{i=1}^{m} \text{Horizontal}(M, i) \ \wedge \bigwedge_{j=1}^{n} \text{Vertical}(M, j) \wedge \bigwedge_{ij} \text{OneSymbol}(M, i, j) \tag{9}$$



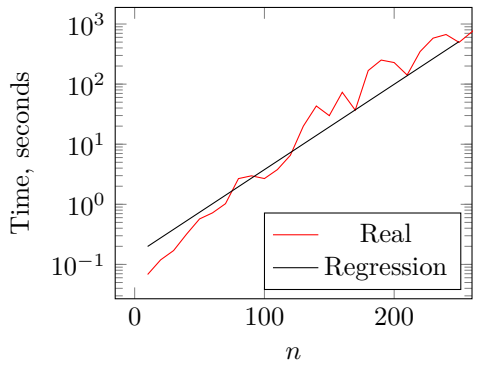
n	ODD, s	Minisat, s	Glucose, s
40	1.072	0.319	0.370
70	5.744	1.024	1.123
100	17.834	2.683	9.057
130	39.440	19.997	39.525
160	73.322	73.149	136.082
190	125.788	251.317	241.092
220	195.433	348.214	336.554
250	291.692	495.185	1038.347

Figure 4 Performances of ODD, Minisat and Glucose solvers on Pigeonhole picture of size $n \times n$.

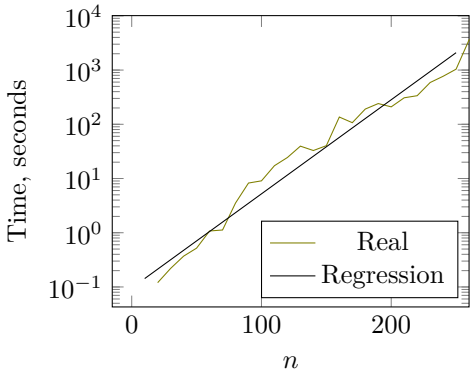
The test cases corresponding to the CNF translation of the grid CSPs $\mathsf{PHP}(m, m)$ and $\mathsf{PHP}(m + 1, m)$ were given as input to the SAT solvers Minisat 2.2[1] and Glucose 4.1[2].

The performance of these solvers on the barely satisfiable case is plotted on Figure 4. The timeout for each experiment was set at 3600 seconds. As it can be seen, Glucose solver times out for $n > 250$, and Minisat times out for $n > 350$. The performance of SAT solvers seems to be exponential, while ODD solver performs clearly in polynomial time.

[1] `http://www.minisat.se/Main.html`
[2] `https://www.labri.fr/perso/lsimon/glucose/`



(a) Minisat.

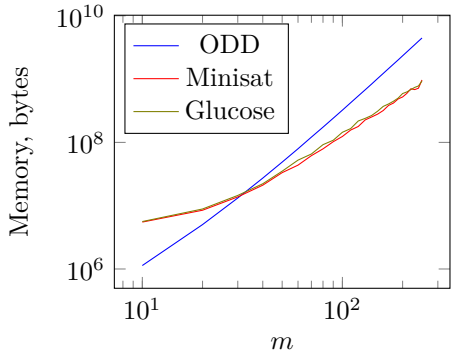


(b) Glucose.

Figure 5 Exponential regressions for SAT solver running times.

Based on the results of the tests, the empirical exponential approximations were estimated for running times of Minisat and Glucose solvers. For Minisat, it is $O^*(1.033^n)$, while for Glucose it is $O^*(1.041^n)$. The regressions are plotted on Figure 5.

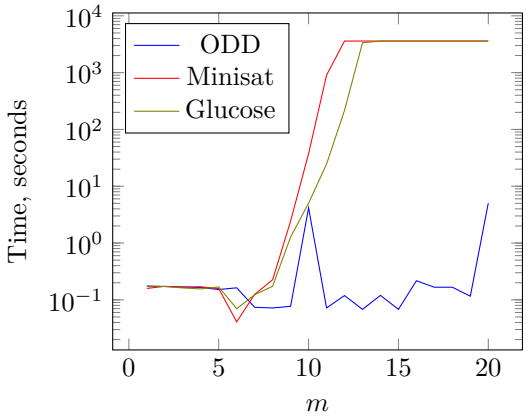
Interestingly, the amount of memory used by the ODD solver is significantly larger than the amount of memory used by SAT solvers. As it is clearly shown on Figure 6, both amounts are polynomial, but the degree of the polynomial for the ODD solver is larger. SAT solvers most probably use linear amount of memory in terms of the formula length (which is quadratic with respect to m). However, the ODD-based solver needs to store ODDs for all layers in order to be able to restore the solution, so the amount of memory used by this solver on this test is cubic.



n	ODD, Mb	Minisat, Mb	Glucose, Mb
40	27.273	21.082	22.202
70	121.126	61.706	65.559
100	323.002	123.988	144.478
130	677.117	226.307	239.545
160	1227.509	318.518	370.232
190	2007.272	485.630	500.627
220	3077.601	702.075	739.147
250	4458.830	969.132	934.675

Figure 6 Memory used by ODD, Minisat and Glucose solvers on Pigeonhole picture of size $m \times m$.

A more expressive difference is achieved for the barely unsatisfiable case, $\mathsf{PHP}(m+1, m)$. The performance is plotted in figure 7. Both solvers time out in this case even for for $n = 14$. Therefore, we can conclude that CNF encodings corresponding to $\mathsf{PHP}(m+1, m)$ are extremely hard for the tested SAT solvers.



m	ODD, s	Minisat, s	Glucose, s
5	0.152	0.158	0.169
6	0.163	0.041	0.070
7	0.074	0.125	0.123
8	0.072	0.226	0.174
9	0.077	2.436	1.277
10	4.169	37.477	5.043
11	0.072	903.836	24.553
12	0.119	3598.044	217.029

Figure 7 Performances of ODD, Minisat and Glucose solvers on Pigeonhole picture of size $(m+1) \times m$. Both SAT solvers timed out for $m \geq 14$, while the ODD solver run without problems in all test sizes (up to $m \leq 400$).

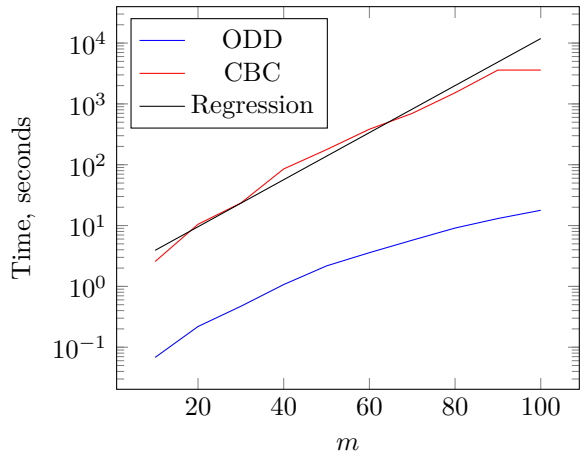
4.4 Integer Programming Translation

The ILP encoding of grid CSPs is done in a similar way to the CNF encoding. More precisely, a variable $x_{i,j,s} \in \{0,1\}$ is created for each $(i,j) \in [m] \times [n]$ and each $s \in [k]$. Then, the following constraints are added:

$$\text{OneSymbol}(M,i,j) \equiv \sum_{s \in [k]} x_{ijs} = 1 \tag{5'}$$

$$\text{Horizontal}(M,i,j,\sigma,\sigma') \equiv \; x_{ij\sigma} + x_{i(j+1)\sigma'} < 2, \quad \text{if } (\sigma,\sigma') \notin H_{i,j} \tag{6'}$$

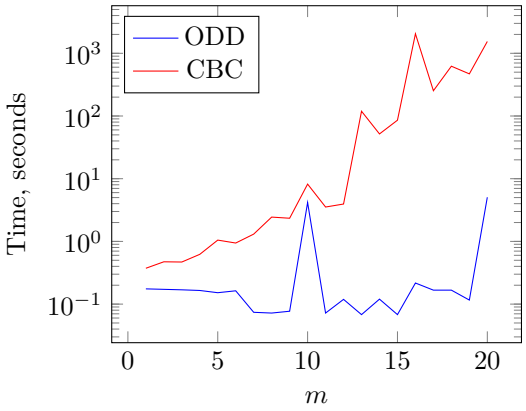
$$\text{Vertical}(M,i,j,\sigma,\sigma') \equiv \; x_{ij\sigma} + x_{(i+1)j\sigma'} < 2, \quad \text{if } (\sigma,\sigma') \notin V_{i,j} \tag{7'}$$



n	ODD, s	CBC, s
10	0.068	2.565
20	0.219	10.489
30	0.471	23.465
40	1.072	85.323
50	2.172	177.186
60	3.584	383.363
70	5.744	698.603
80	9.105	1522.991

Figure 8 Performances of ODD and CBC solvers on Pigeonhole picture if size $m \times m$.

The resulting ILP instances were given to the Coin-OR CBC[3] solver. The performance is plotted on Figure 8. For $m \times m$ instances the solver timed out for $m = 90$. The running time of the solvers grows clearly as an exponential function. On each of the test cases, the running time of the ILP sover is several orders of magitude above the running time of the ODD-based solver. For the ILP solver empirical exponential approximation of $O^*(1.093^n)$ was also built. The regression is shown on Figure 8. As in the case of SAT solvers, for the barely unsatisfiable case the ILP solver timed out much earlier (for $m = 21$). The results for $m \leq 20$ are plotted on Figure 9.



m	ODD, s	CBC, s
6	0.163	0.942
8	0.072	2.422
10	4.169	8.201
12	0.119	3.946
14	0.120	51.762
16	0.217	2046.332
18	0.167	622.360
20	5.060	1543.712

Figure 9 Performances of ODD and CBC solvers on Pigeonhole picture if size $(m + 1) \times m$.

5 Conclusion

In this work, we have lifted the notion of smoothness for pictures (grid-CSPs with uniform vertical and horizontal constraints) to the context of general grid CSPs, where the vertical and horizontal constraints at each position (i, j) may depend on (i, j). We have shown that the satisfiability problem for grid-CSPs of polynomial smoothness can be solved in polynomial time. Additionally, we have given evidence for the relevance of the concept of smoothness in practical situations by demonstrating empirically that the class for pigeonhole grids can be solved in cubic time.

This opens up the possibility of applying our algorithm to grid CSPs involving constraints where one particular object can appear at most once in any row/column, such as the problem of placing r towers in an $m \times n$ chess grid in such a way that no tower attacks each other. It can be shown in this case that the ODDs occurring in the execution of the algorithm have size polynomial in m, n and r.

References

1 Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018. doi:10.1142/S0218213018400018.

2 Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, pages 319–351, 2004.

[3] https://github.com/coin-or/Cbc

3 Paul Beame and Toniann Pitassi. Simplified and improved resolution lower bounds. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, pages 274–282. IEEE, 1996.

4 Samuel R Buss, Jan Hoffmann, and Jan Johannsen. Resolution trees with lemmas: Resolution refinements that characterize DLL algorithms with clause learning. *Logical Methods in Computer Science*, 4, 2008.

5 Alessandra Cherubini, Stefano Crespi Reghizzi, Matteo Pradella, and Pierluigi San Pietro. Picture languages: Tiling systems versus tile rewriting grammars. *Theoretical Computer Science*, 356(1):90–103, 2006.

6 Mateus de Oliveira Oliveira. Satisfiability via smooth pictures. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 13–28. Springer, 2016.

7 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

8 John Forrest and Robin Lougee-Heimer. Cbc user guide. In *Emerging theory, methods, and applications*, pages 257–277. INFORMS, 2005.

9 Dora Giammarresi and Antonio Restivo. Recognizable picture languages. *International Journal of Pattern Recognition and Artificial Intelligence*, 6(2&3):241–256, 1992.

10 Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

11 Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively P-simulate general propositional resolution. In *Proc. of the 23rd National Conference on Artificial Intelligence (AAAI 2008)*, pages 283–290, 2008.

12 Changwook Kim and Ivan Hal Sudborough. The membership and equivalence problems for picture languages. *Theoretical Computer Science*, 52(3):177–191, 1987.

13 Jan Krajíček. Lower bounds to the size of constant-depth propositional proofs. *The Journal of Symbolic Logic*, 59(01):73–86, 1994.

14 Jan Krajíček, Pavel Pudlák, and Alan Woods. An exponential lower bound to the size of bounded depth frege proofs of the pigeonhole principle. *Random Structures & Algorithms*, 7(1):15–39, 1995.

15 Michel Latteux and David Simplot. Recognizable picture languages and domino tiling. *Theoretical computer science*, 178(1):275–283, 1997.

16 Hermann A Maurer, Grzegorz Rozenberg, and Emo Welzl. Using string languages to describe picture languages. *Information and Control*, 54(3):155–185, 1982.

17 Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

18 Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, 2011.

19 Toniann Pitassi, Paul Beame, and Russell Impagliazzo. Exponential lower bounds for the pigeonhole principle. *Computational complexity*, 3(2):97–140, 1993.

20 Azriel Rosenfeld. *Picture languages: formal models for picture recognition*. Academic Press, 2014.

21 David Simplot. A characterization of recognizable picture languages by tilings by finite sets. *Theoretical Computer Science*, 218(2):297–323, 1999.

22 Gift Stromoney, Rani Siromoney, and Kamala Krithivasan. Abstract families of matrices and picture languages. *Computer Graphics and Image Processing*, 1(3):284–307, 1972.

An Experimental Evaluation of Semidefinite Programming and Spectral Algorithms for Max Cut

Renee Mirka
Cornell University, Ithaca, NY, USA

David P. Williamson
Cornell University, Ithaca, NY, USA

Abstract

We experimentally evaluate the performance of several Max Cut approximation algorithms. In particular, we compare the results of the Goemans and Williamson algorithm using semidefinite programming with Trevisan's algorithm using spectral partitioning. The former algorithm has a known .878 approximation guarantee whereas the latter has a .614 approximation guarantee. We investigate whether this gap in approximation guarantees is evident in practice or whether the spectral algorithm performs as well as the SDP. We also compare the performances to the standard greedy Max Cut algorithm which has a .5 approximation guarantee and two additional spectral algorithms. The algorithms are tested on Erdős-Renyi random graphs, complete graphs from TSPLIB, and real-world graphs from the Network Repository. We find, unsurprisingly, that the spectral algorithms provide a significant speed advantage over the SDP. In our experiments, the spectral algorithms return cuts with values which are competitive with those of the SDP.

2012 ACM Subject Classification Theory of computation → Approximation algorithms analysis

Keywords and phrases Max Cut, Approximation Algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.19

Supplementary Material *Software (Source Code)*: `https://github.com/rmirka/max-cut-experiments`; archived at `swh:1:dir:eb13652be65db33c0ea45e66314475a4327cae0d`

Funding This work was supported by the National Science Foundation [NSF CCF-2007009].

1 Introduction

Given as input a graph $G = (V, E)$ and weights $w_e \in \mathbb{R}^+$ for all $e \in E$, the Max Cut problem asks to partition V into two sets such that the sum of the weights of the edges crossing the partition is maximized. In particular, a cut is given by a pair of sets (S, T) such that $V = S \cup T$ and $S \cap T = \emptyset$. The value of this cut is

$$\sum_{(s,t)\in E: s\in S, t\in T} w_{(s,t)},$$

and Max Cut seeks to find a cut maximizing this quantity.

Max Cut is a problem of vast theoretical and practical significance. It is polynomial solvable for certain classes of graphs, e.g. planar graphs [9, 15], and is well-known to be NP-hard in general; it appears on Karp's original list of NP-complete problems [12]. Additionally, Max Cut has applications in fields such as data clustering [16], circuit design, and statistical physics [1]; see Poljak and Tuza for a comprehensive survey [17].

Many researchers have made improvements towards exact solvers for Max Cut. For general graphs of unbounded average degree, Williams presented a Max Cut algorithm using exponential space to exactly solve (and count the number of optimal solutions) in $O(m^3 2^{\omega n/3})$ time where $\omega < 2.376$ [24]. Croce, Kaminski, and Paschos introduced an algorithm to find a Max Cut in graphs with bounded maximum degree, Δ, running in $O^*(2^{(1-2/\Delta)n})$ time where $O^*()$ suppresses polynomial factors [4]. Golovnev improved this

to $O^*(2^{(1-3/(\Delta+1))n})$ [8]. Results from Hrga et al., Hrga and Povh, Krislock, Malick, and Roupin, and Rendl, Rinaldi, and Wiegele utilize branch and bound techniques to produce further exact solvers [10, 11, 14, 19]. However, due to the lack of an efficient (polynomial-time) algorithm, researchers have also considered finding good approximation algorithms. An α-approximation algorithm is a polynomial time algorithm which guarantees a solution with a value at least an α fraction of the optimal solution. As one of the most well-studied problems in theoretical computer science, there is a breadth of known approximation algorithms for Max Cut varying in runtime and approximation guarantee quality.

The simplest randomized approximation algorithm assigns a vertex $v \in V$ to either S or T with equal probability. In expectation, this is a .5-approximation algorithm. Another .5-approximation can be achieved through a simple greedy algorithm presented by Sahni and Gonzalez [21]. In this algorithm, start with $S, T = \emptyset$. While there are still unassigned vertices, any unassigned vertex v is chosen and the quantities $c_S(v) = \sum_{u \in S:(u,v) \in E} w_{(u,v)}$ and $c_T(v) = \sum_{u \in T:(u,v) \in E} w_{(u,v)}$ are computed. If $c_S(v) > c_T(v)$, v is assigned to T and otherwise to S.

The .5-approximation guarantee was the best known until Goemans and Williamson [7] presented a .878-approximation algorithm, which is the best possible guarantee assuming the Unique Games Conjecture [13]. Their algorithm relies on a semidefinite programming (SDP) relaxation of the Max Cut problem to find a high-value cut. While the approximation guarantee likely cannot be surpassed by another polynomial-time algorithm, solving the SDP can be quite costly in practice.

More recently, Trevisan [23] introduced a simple .531-approximation for Max Cut based on spectral partitioning. Soto [22] improved this guarantee to .614. Though the approximation guarantees are weaker than the SDP algorithm, the spectral techniques are much cheaper to implement. In theory, there is a trade off between the computational speed and solution quality of Goemans and Williamson's SDP algorithm versus Trevisan's spectral algorithm. This paper seeks to determine whether this trade off exists in practice or if Trevisan's algorithm returns solutions competitive with those of the SDP.

Several previous papers have experimentally compared Max Cut algorithms and heuristics. Bertoni, Campadelli, and Grossi compare cuts computed by their .39-approximation Lorena algorithm, inspired by Goemans-Williamns SDP, to the SDP and a neural .5-approximation algorithm [3]. They found, on average, Lorena provided larger cuts on random graphs in significantly less time than the SDP and comparable time to the neural algorithm. Dolezal, Hofmeister, and Lefmann compare cuts from six algorithms, including the SDP, on random graphs concluding that the computationally-cheap random .5-approximation algorithm provides the best tradeoff between runtime and cut quality [5]. Goemans and Williamson also included computational results in their initial paper demonstrating the SDP often outperforms its .878 approximation guarantee. Berry and Goldberg tested several graph partitioning heuristics against each other and against the SDP, finding the heuristics consistently produce larger cuts than the SDP [2]. Dunning, Gupta, and Silberholz performed a systematic review of Max Cut heuristics and computationally tested 19 of them [6]. As far as we are aware there are no previously published results comparing Trevisan's spectral algorithm.

In this paper, we evaluate the performances of the SDP, spectral, and greedy algorithms on a variety of graphs. Section 2 provides more complete descriptions of the five algorithms considered. Section 3 describes the experiments and presents the results of the algorithms on different classes of graphs. Finally, Section 4 concludes with a summary of the performances and introduces a few possible directions for future theoretical study.

2 Algorithms

This section describes the five algorithms that we implemented for Max Cut. Section 2.1 describes the benchmark greedy .5-approximation algorithm for Max Cut. Section 2.2 describes Trevisan's spectral algorithm for Max Cut, while Section 2.3 describes two simplifications of this algorithm. Finally Section 2.4 describes the SDP algorithm.

2.1 Greedy Algorithm

The first Max Cut algorithm we consider is the standard greedy algorithm. This fast and simple algorithm provides a benchmark for the speed and cut value of the others. For a graph $G = (V, E)$, we return a cut (L, R) by greedily assigning vertices to either L or R one at a time. We start with $L, R = \emptyset$. In each step of the algorithm, we choose a vertex $v \in V$ yet to be assigned to L or R. Then we add v to L or R by choosing the larger of the two cuts $(L \cup \{v\}, R)$ and $(L, R \cup \{v\})$ at this step.

2.2 Trevisan's Algorithm

The next algorithm for finding a large cut in a graph is Trevisan's spectral algorithm. Trevisan proved a .531 approximation ratio for the algorithm, while Soto improved the analysis to .614. Here, we describe Soto's presentation of the algorithm. Given a graph $G = (V, E)$ with $|V| = n$, the adjacency matrix $A = (a_{ij})$ is given by $a_{ij} = 1$ if $(i, j) \in E$ and 0 otherwise. Then the normalized adjacency matrix $\mathcal{A}$ is given by $\mathcal{A} = D^{-1/2}AD^{-1/2}$ where $D = diag(d)$ for $d(i)$ the degree of vertex i. In our implementation of the algorithm, we compute the eigenvector, x, corresponding to the minimum eigenvalue of $I + \mathcal{A}$. After normalizing x so that $\max_i |x_i| = 1$, a number t^2 is drawn uniformly at random from [0,1]. We let

$$L = \{v : x_v \leq -t\},$$
$$R = \{v : x_v \geq t\}, \text{ and}$$
$$V' = V \setminus (L \cup R).$$

Now (L, R) represents a partial cut of the vertices with V' being the vertices yet to be partitioned.

Given L, R, and V', we compute

C = total weight of the edges between L and R,

X = total weight of the edges between $L \cup R$ and V', and

M = total weight of all edges − total weight of the edges between vertices of V'.

If $C + X/2 - M/2 < 0$, we use the greedy algorithm to partition the vertices instead of t as the expected value of the cut is worse than that of greedy. If $C + X/2 - M/2 > 0$, we keep the partial cut and recurse to find a cut of the vertices in V' given by (L', R'). Finally, we return the larger of the cuts $(L \cup L', R \cup R')$ and $(L \cup R', R \cup L')$.

Since the results of Trevisan's algorithm are highly reliant on the t^2 value chosen, one could ask if there are ways to modify the algorithm to increase the likelihood of choosing a good t^2 value. We tested two methods. The first chooses more than one t^2 value at each stage of the algorithm. In particular, $T = \max(5, n/k)$ values were chosen where k was tested for $k = 1, 2, 5, 10, 15, 25, 50, 100$. The idea here was that choosing many random numbers should increase the probability of choosing a "good" random number. The major roadblock is deciding which partial cut corresponding to one of the t^2 values the algorithm should

recurse on. We tested the greedy choice. More specifically, C, X, and M were computed for each drawn t^2 value as above, and we kept the partial cut maximizing $C + X/2 - M/2$. We made this selection because it represents the partial cut that is currently performing better than the greedy algorithm by the largest margin. In particular, given a partial cut due to a partial assignment of vertices, we can consider three types of edges: edges with both endpoints assigned, edges with exactly one endpoint assigned, and edges with neither endpoint assigned. The third type are not affected by the partial cut and aren't considered in this iteration. However, C computes the value of the cut in Trevisan's algorithm due to the first type of edge. $X/2$ is the expected value added to this cut from the edges of the second type if the remaining vertices are greedily assigned, and $M/2$ is the expected value of the greedy cut due to edges of both of the first two types. Therefore, if $C + X/2 > M/2$, the partial cut being considered is performing better than the greedy algorithm would be in expectation. The greater the difference between $C + X/2$ and $M/2$, theoretically the better Trevisan's algorithm is performing compared to greedy. It is not obvious that this is the best heuristic, but it does allow the algorithm to test several random values quickly.

Alternatively, we also experimented with running Trevisan's algorithm for several iterations and maintaining the best cut that was found. The advantage here as opposed to the previous modification is we do not have to determine which t value to keep. However, the runtime is slower because the entire algorithm is run several times instead of adding additional quick random draws.

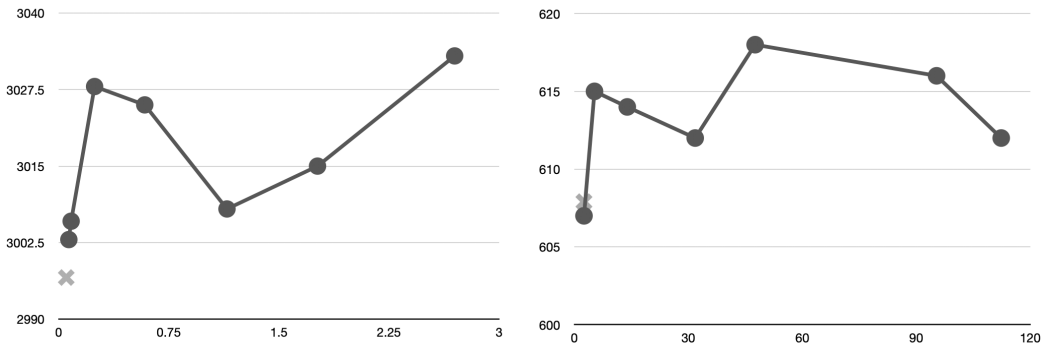
The results of these modifications for two of the tested graphs are provided in Figure 1. In each figure, the line represents the results from trials of running the algorithm multiple times (1, 2, 5, 10, 20, 35, and 50 times). Note that the line is not monotonically increasing. This is because each group of runs was unique and not a cumulative total. For example, when considering how well Trevisan's algorithm performs when running 10 iterations, we run 10 new iterations and do not build off of the 5 from the previous data point. The single dot represents the average runtime and cut value of the best result when implementing the first modification of multiple t^2 values.

The experiments were run on a variety of graphs and these are a representative sample. In general, it seems running the algorithm multiple times is more effective in increasing cut quality than choosing multiple t^2 values. However, the number of iterations needed is not obvious, though it appears at least 5 are beneficial. Due to this observation, we use this method of running Trevisan's algorithm 5 times and keeping the best cut for the experiments presented in this paper.

2.3 Simple Spectral and Sweep Cuts Algorithms

The simple spectral algorithm is a modification of Trevisan's algorithm described in the previous section. Instead of drawing a random number t in $[0, 1]$, we return the cut corresponding to $t = 0$. In particular, let x be the eigenvector corresponding to the smallest eigenvalue as before. Since scaling numbers by a positive factor does not change their sign, we may skip the normalizing step for x. We let $L = \{v : x_v < 0\}$ and $R = V \setminus L$ and return the cut (L, R). This modified simple spectral algorithm has no known approximation guarantee.

The sweep cuts algorithm works in a similar fashion. Here, we consider $n - 1$ different cuts and return the best. Given the smallest eigenvector x, we sort the entries so that $x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_n}$. Then we calculate the sweep cut value for $L_j = \{i_1, \ldots, i_j\}$ and $R_j = V \setminus L_j$ for $j = 1, \ldots, n - 1$. The sweep cuts algorithm returns the cut (L_j, R_j) of maximal value.



(a) johnson16-2-4.

(b) ca-netscience.

Figure 1 Plots depicting the effects on runtime and returned cut quality of running Trevisan's algorithm multiple times on the johnnson16-2-4 and ca-netscience graphs. The X and Y axes are the time in seconds and the cut value, respectively. The light gray 'x' is presented for comparison and is the result of running Trevisan's algorithm once, but testing many random values.

It is worth noting that the sweep cuts algorithm will always perform at least as well as the simple spectral algorithm in terms of cut value since one of the sweep cuts will be the same as the $t = 0$ cut. However, it is interesting to see how much better the sweep cuts algorithm performs since it is also guaranteed to have a slower runtime for the same reasons.

2.4 SDP Algorithm

Goemans and Williamson introduced a .878 approximation algorithm for Max Cut. Instead of directly solving

$$MaxCut(G) = \max_{x_i \in \{1,-1\}} \frac{1}{2} \sum_{i<j} w_{ij}(1 - x_i x_j)$$

where again w_{ij} is the weight of edge (i, j), they relax this program to one solvable by a semidefinite program. In particular, instead of requiring $x_i \in \{1, -1\}$, they require $v_i \in \mathbb{R}^n$ to be unit vectors and replace $x_i x_j$ with $\langle v_i, v_j \rangle$. Given a solution to this SDP relaxation, they draw a random vector $r \in \mathbb{R}^n$ uniformly from the unit sphere and partition the vertices according to

$$L = \{i : \langle r, v_i \rangle \leq 0\} \text{ and}$$
$$R = \{i : \langle r, v_i \rangle > 0\}.$$

This gives the .878 approximation in expectation. For our testing purposes, we draw 100 random vectors instead of 1. In terms of computation time, this is a cheap modification as the SDP does not have to be rerun. We return the maximum cut resulting from these 100 random vectors.

3 Experiments

All algorithms were implemented in Julia. They were run on a machine with a 2 GHz Intel Core i5 processor with 8 GB 1867 MHz LPDDR3 memory. The SDP algorithm was computed with the JuMP modeling language for Julia and the SCS package providing the splitting cone solver. The LinearAlgebra package was used for the eigenvector computations of the spectral algorithms.

We measured the algorithms' performance with three types of test data. We used 20 Erdős-Renyi random graphs with 50-500 vertices, 16 complete graphs from TSPLIB [18] with 29-280 vertices (average 124), and 17 sparser graphs from the Network Repository [20] with 39-1133 vertices (average 327).

3.1 Erdős-Renyi Random Graphs

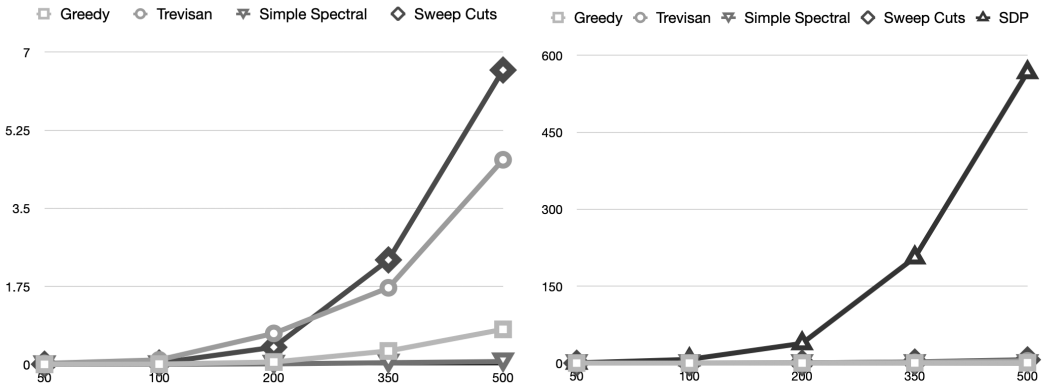
The first class of graphs tested was Erdős-Renyi random graphs. An Erdős-Renyi random graph $G(n, p)$ is a graph on n vertices where each possible edge is included independently with probability p. We tested random graphs with $n = 50, 100, 200, 350, 500$ and $p = .1, .25, .5, .75$. In our model, each included edge was given an edge weight of 1.

In terms of speed, the simple spectral algorithm significantly outperformed the other algorithms on all but three tested random graphs (where greedy was faster). On the other end of the spectrum, the SDP was far slower than the alternative algorithms. The time statistics are presented in Table 1. The plots in Figure 2a and Figure 2b illustrate how the computation times of each algorithm grow as the number of vertices increases. For these plots, we use the data from Table 1 with $p = .5$ fixed.

Table 1 The time in seconds each algorithm took to compute a cut of an Erdős-Renyi random graph.

Graph	Greedy	Trevisan	Simple Spectral	Sweep Cuts	SDP
G(50,0.1)	$\mathbf{5.560 \times 10^{-3}}$	2.503×10^{-1}	5.192×10^{-2}	3.485×10^{-2}	5.556×10^{-1}
G(50,0.25)	7.600×10^{-4}	1.533×10^{-2}	$\mathbf{6.600 \times 10^{-4}}$	2.410×10^{-3}	4.711×10^{-1}
G(50,0.5)	1.280×10^{-3}	2.354×10^{-2}	$\mathbf{7.800 \times 10^{-4}}$	4.760×10^{-3}	4.502×10^{-1}
G(50,0.75)	1.870×10^{-3}	1.741×10^{-2}	$\mathbf{8.000 \times 10^{-4}}$	8.690×10^{-3}	9.727×10^{-1}
G(100,0.1)	$\mathbf{2.000 \times 10^{-3}}$	3.597×10^{-2}	2.380×10^{-3}	1.106×10^{-2}	2.929
G(100,0.25)	3.860×10^{-3}	6.945×10^{-2}	$\mathbf{2.340 \times 10^{-3}}$	1.849×10^{-2}	3.440
G(100,0.5)	7.330×10^{-3}	1.021×10^{-1}	$\mathbf{2.370 \times 10^{-3}}$	3.206×10^{-2}	7.235
G(100,0.75)	1.064×10^{-2}	1.162×10^{-1}	$\mathbf{9.960 \times 10^{-3}}$	2.653×10^{-1}	5.823
G(200,0.1)	$\mathbf{1.222 \times 10^{-2}}$	2.464×10^{-1}	3.299×10^{-2}	6.941×10^{-2}	2.575×10^{1}
G(200,0.25)	2.963×10^{-2}	2.444×10^{-1}	$\mathbf{8.650 \times 10^{-3}}$	1.892×10^{-1}	2.942×10^{1}
G(200,0.5)	5.428×10^{-2}	6.949×10^{-1}	$\mathbf{1.266 \times 10^{-2}}$	3.853×10^{-1}	3.848×10^{1}
G(200,0.75)	7.809×10^{-2}	6.463×10^{-1}	$\mathbf{9.900 \times 10^{-3}}$	4.740×10^{-1}	4.945×10^{1}
G(350,0.1)	6.192×10^{-2}	1.022	$\mathbf{2.407 \times 10^{-2}}$	4.184×10^{-1}	1.216×10^{2}
G(350,0.25)	1.737×10^{-1}	1.201	$\mathbf{3.009 \times 10^{-2}}$	1.138	1.726×10^{2}
G(350,0.5)	3.013×10^{-1}	1.718	$\mathbf{3.848 \times 10^{-2}}$	2.342	2.058×10^{2}
G(350,0.75)	4.438×10^{-1}	2.015	$\mathbf{3.324 \times 10^{-2}}$	3.015	2.798×10^{2}
G(500,0.1)	1.668×10^{-1}	1.875	$\mathbf{7.049 \times 10^{-2}}$	1.622	3.355×10^{2}
G(500,0.25)	3.937×10^{-1}	2.239	$\mathbf{5.936 \times 10^{-2}}$	3.325	3.919×10^{2}
G(500,0.5)	7.859×10^{-1}	4.587	$\mathbf{6.472 \times 10^{-2}}$	6.598	5.669×10^{2}
G(500,0.75)	1.260	5.263	$\mathbf{7.195 \times 10^{-2}}$	9.837	8.116×10^{2}

The spectral algorithms also performed the best in terms of the returned cut quality for random graphs. The SDP returned the best result for three graphs but one of the cuts was matched by Trevisan's algorithm. Trevisan's algorithm provided the best cut for 5 graphs, and the sweep cuts algorithm was the second best option for all of these, in addition to being the best for 14 graphs. These results are provided in Table 2.



(a) All tested algorithms excluding the SDP. **(b)** All tested algorithms.

Figure 2 Plots depicting the effects on runtime of increasing the number of vertices of an Erdős-Renyi graph with $p = .5$. The X and Y axes are the number of vertices and the computation time in seconds, respectively.

Table 2 The value of the cut each algorithm returned for an Erdős-Renyi random graph.

Graph	Greedy	Trevisan	Simple Spectral	Sweep Cuts	SDP
G(50,0.1)	8.700×10^1	$\mathbf{9.600 \times 10^1}$	9.400×10^1	9.500×10^1	9.200×10^1
G(50,0.25)	1.970×10^2	2.060×10^2	2.060×10^2	2.080×10^2	$\mathbf{2.100 \times 10^2}$
G(50,0.5)	3.480×10^2	$\mathbf{3.600 \times 10^2}$	3.560×10^2	$\mathbf{3.600 \times 10^2}$	$\mathbf{3.600 \times 10^2}$
G(50,0.75)	5.140×10^2	5.140×10^2	4.990×10^2	5.190×10^2	$\mathbf{5.240 \times 10^2}$
G(100,0.1)	3.210×10^2	3.290×10^2	3.420×10^2	$\mathbf{3.430 \times 10^2}$	3.290×10^2
G(100,0.25)	7.640×10^2	7.830×10^2	7.850×10^2	$\mathbf{7.880 \times 10^2}$	7.860×10^2
G(100,0.5)	1.351×10^3	1.363×10^3	1.346×10^3	$\mathbf{1.375 \times 10^3}$	1.361×10^3
G(100,0.75)	2.019×10^3	2.024×10^3	2.020×10^3	$\mathbf{2.026 \times 10^3}$	2.016×10^3
G(200,0.1)	1.212×10^3	$\mathbf{1.250 \times 10^3}$	1.234×10^3	1.242×10^3	1.211×10^3
G(200,0.25)	2.795×10^3	2.859×10^3	2.847×10^3	$\mathbf{2.861 \times 10^3}$	2.778×10^3
G(200,0.5)	5.388×10^3	5.420×10^3	5.412×10^3	$\mathbf{5.423 \times 10^3}$	5.326×10^3
G(200,0.75)	7.784×10^3	7.855×10^3	7.831×10^3	$\mathbf{7.875 \times 10^3}$	7.815×10^3
G(350,0.1)	3.556×10^3	3.582×10^3	3.639×10^3	$\mathbf{3.651 \times 10^3}$	3.611×10^3
G(350,0.25)	8.378×10^3	8.544×10^3	8.583×10^3	$\mathbf{8.585 \times 10^3}$	8.236×10^3
G(350,0.5)	1.623×10^4	1.627×10^4	1.643×10^4	$\mathbf{1.649 \times 10^4}$	1.603×10^4
G(350,0.75)	2.356×10^4	$\mathbf{2.378 \times 10^4}$	2.374×10^4	2.374×10^4	2.353×10^4
G(500, .1)	7.155×10^3	7.155×10^3	7.303×10^3	$\mathbf{7.329 \times 10^3}$	7.097×10^3
G(500, .25)	1.673×10^4	1.697×10^4	1.712×10^4	$\mathbf{1.714 \times 10^4}$	1.652×10^4
G(500, .5)	3.272×10^4	3.275×10^4	3.313×10^4	$\mathbf{3.314 \times 10^4}$	3.311×10^4
G(500, .75)	4.820×10^4	$\mathbf{4.852 \times 10^4}$	4.847×10^4	4.849×10^4	4.813×10^4

Table 3 The time in seconds each algorithm took to compute a cut of a complete graph from TSPLIB.

Graph	Greedy	Trevisan	Simple Spectral	Sweep Cuts	SDP
bayg29	4.300×10^{-4}	5.040×10^{-3}	$\mathbf{3.100 \times 10^{-4}}$	9.700×10^{-4}	1.945×10^{-1}
bays29	7.500×10^{-4}	9.660×10^{-3}	$\mathbf{6.900 \times 10^{-4}}$	1.160×10^{-3}	3.002×10^{-1}
berlin52	$\mathbf{2.190 \times 10^{-3}}$	3.058×10^{-2}	2.540×10^{-3}	7.580×10^{-3}	9.291×10^{-1}
bier127	3.674×10^{-2}	3.520×10^{-1}	$\mathbf{1.370 \times 10^{-2}}$	1.125×10^{-1}	6.832
brazil58	$\mathbf{3.260 \times 10^{-3}}$	3.237×10^{-2}	4.370×10^{-3}	8.490×10^{-3}	1.229
brg180	$\mathbf{9.482 \times 10^{-2}}$	1.223	2.028×10^{-1}	3.072×10^{-1}	1.548×10^{1}
ch130	3.371×10^{-2}	3.355×10^{-1}	$\mathbf{1.221 \times 10^{-2}}$	1.352×10^{-1}	7.503
ch150	5.701×10^{-2}	9.769×10^{-1}	$\mathbf{1.811 \times 10^{-2}}$	1.745×10^{-1}	1.209×10^{1}
d198	1.094×10^{-1}	1.402	$\mathbf{3.997 \times 10^{-2}}$	4.869×10^{-1}	3.844×10^{1}
eil101	1.912×10^{-2}	5.535×10^{-1}	$\mathbf{8.870 \times 10^{-3}}$	7.724×10^{-2}	3.982
gr120	2.376×10^{-2}	4.412×10^{-1}	$\mathbf{1.050 \times 10^{-2}}$	1.153×10^{-1}	1.372×10^{1}
gr137	4.262×10^{-2}	6.078×10^{-1}	$\mathbf{1.493 \times 10^{-2}}$	1.665×10^{-1}	1.452×10^{1}
gr202	1.194×10^{-1}	2.471	$\mathbf{2.662 \times 10^{-2}}$	4.779×10^{-1}	3.067×10^{1}
gr96	1.632×10^{-2}	3.044×10^{-1}	$\mathbf{4.690 \times 10^{-3}}$	5.674×10^{-2}	4.753
kroA100	1.880×10^{-2}	2.037×10^{-1}	$\mathbf{7.720 \times 10^{-3}}$	5.644×10^{-2}	3.285
a280	2.988×10^{-1}	4.439	$\mathbf{6.127 \times 10^{-2}}$	1.534	1.555×10^{2}

3.2 Complete Graphs

The algorithms were also tested on 16 complete graphs from TSPLIB, an online library of sample instances for the Travelling Salesman Problem and related graph problems. The performance in regards to time largely mirrored that of the random graphs. The simple spectral algorithm was significantly faster than the rest of the algorithms on the vast majority of graphs, followed by the greedy, Trevisan's, and sweep cuts algorithms with relatively quick computation times, and the SDP with a massive slowdown. This data is presented in Table 3.

Again, the spectral algorithms most frequently returned the highest quality cut; these results are summarized in Table 4. For $\frac{15}{16}$ (93.75%) of these graphs, the best cut was found by either the simple spectral algorithm (5 times), Trevisan's algorithm (3 times) or the sweep cuts algorithm (12 times). Furthermore, for the graph d198 where the SDP computed the best cut, the loss in quality from the spectral solutions was quite small. These values are given in Table 5.

In Figure 3a, Figure 3b, Figure 4a, and Figure 4b, we provide a representative sample of the trade-off between runtime and returned cut value of the algorithms using the a280, ch150, and eil101 graphs.

3.3 Sparser Graphs

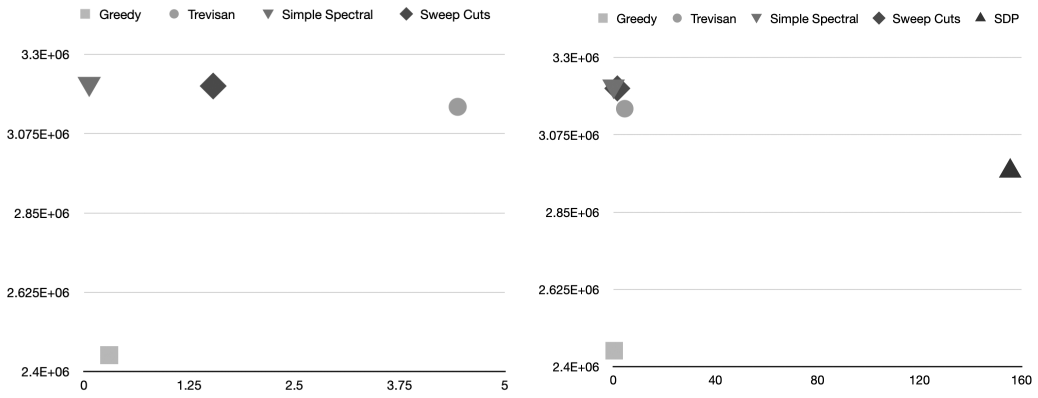
The third group of graphs is composed of a variety of graphs from the Network Repository, an online and interactive collection of network graph data coming from a variety of sources and applications. Though more structured than a random graph, these 17 graphs are sparser than the complete graphs tested in Section 3.2 and were chosen from a range of real-world scenarios. Unsurprisingly, the relationships between relative computation times remains unchanged. The simple spectral and greedy algorithms each accounted for about half of the fastest times while the SDP was consistently considerably slower (Table 6).

Table 4 The value of the cut each algorithm returned for a complete graph from TSPLIB.

Graph	Greedy	Trevisan	Simple Spectral	Sweep Cuts	SDP
bayg29	3.837×10^4	4.225×10^4	$\mathbf{4.269 \times 10^4}$	$\mathbf{4.269 \times 10^4}$	$\mathbf{4.269 \times 10^4}$
bays29	4.831×10^4	5.393×10^4	5.369×10^4	$\mathbf{5.399 \times 10^4}$	5.386×10^4
berlin52	4.532×10^5	4.616×10^5	4.465×10^5	$\mathbf{4.681 \times 10^5}$	4.522×10^5
bier127	2.162×10^7	2.300×10^7	2.322×10^7	$\mathbf{2.330 \times 10^7}$	2.320×10^7
brazil58	$\mathbf{2.319 \times 10^6}$	$\mathbf{2.319 \times 10^6}$	2.315×10^6	2.315×10^6	2.180×10^6
brg180	4.118×10^7	$\mathbf{4.616 \times 10^7}$	4.531×10^7	4.551×10^7	4.330×10^7
ch130	1.777×10^6	1.885×10^6	$\mathbf{1.888 \times 10^6}$	$\mathbf{1.888 \times 10^6}$	1.887×10^6
ch150	2.500×10^6	2.521×10^6	$\mathbf{2.526 \times 10^6}$	$\mathbf{2.526 \times 10^6}$	2.434×10^6
d198	9.635×10^6	1.286×10^7	1.292×10^7	1.293×10^7	$\mathbf{1.293 \times 10^7}$
eil101	1.052×10^5	$\mathbf{1.070 \times 10^5}$	1.063×10^5	1.064×10^5	1.058×10^5
gr120	2.123×10^6	2.147×10^6	2.156×10^6	$\mathbf{2.157 \times 10^6}$	2.154×10^6
gr137	2.241×10^7	3.044×10^7	3.066×10^7	$\mathbf{3.070 \times 10^7}$	3.070×10^7
gr202	1.372×10^7	1.533×10^7	1.559×10^7	$\mathbf{1.593 \times 10^7}$	1.581×10^7
gr96	8.967×10^6	1.156×10^7	$\mathbf{1.166 \times 10^7}$	$\mathbf{1.166 \times 10^7}$	1.157×10^7
kroA100	5.848×10^6	5.850×10^6	$\mathbf{5.897 \times 10^6}$	$\mathbf{5.897 \times 10^6}$	$\mathbf{5.897 \times 10^6}$
a280	2.447×10^6	3.151×10^6	3.21×10^6	$\mathbf{3.21 \times 10^6}$	2.970×10^6

Table 5 The percent decrease in cut value from the SDP to the spectral cuts.

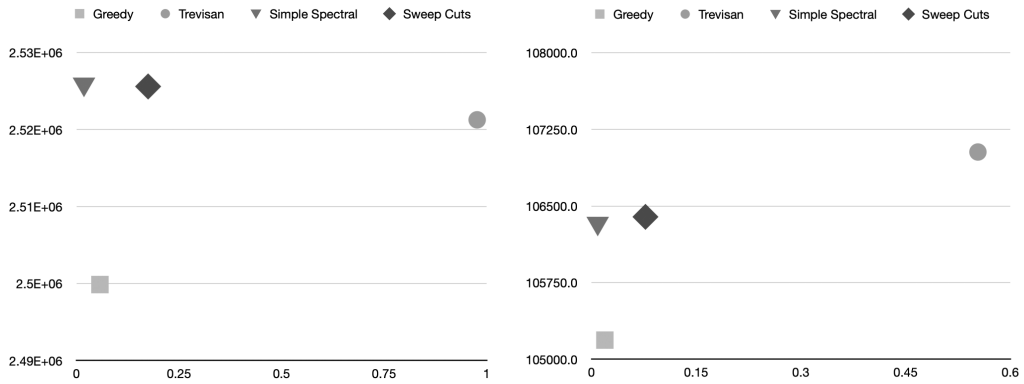
Graph	Trevisan	Simple Spectral	Sweep Cuts
d198	$\sim .6\%$	$\sim .1\%$	$\sim .06\%$



(a) All tested algorithms excluding the SDP.

(b) All tested algorithms.

Figure 3 Plots depicting the computation time and returned cut values of algorithms on the a280 graph. The X and Y axes are the runtime in seconds and the returned cut value, respectively.



(a) ch150.

(b) eil101.

Figure 4 Plots depicting the computation time and returned cut values of algorithms excluding the SDP on the ch150 and eil101 graphs. The X and Y axes are the runtime in seconds and the returned cut value, respectively.

For this group of graphs, the algorithms' relative cut quality is more varied than with the previous. Of the 17 graphs tested, the SDP returned the best cut for 7 instances whereas the spectral algorithms combined for 11 best (with one instance of a tie between the SDP and simple spectral) (Table 7).

In Figure 5a, Figure 5b, Figure 6a, and Figure 6b, we provide a representative sample of the trade-off between runtime and returned cut value of the algorithms using the graphs ia-infect-dublin, email-enron-only, and soc-dolphins.

4 Conclusion

The goal of this paper was to compare Max Cut algorithms with varying approximation guarantees in practice. In particular, we know the SDP has the provably best approximation guarantee; however, it is also the costliest in terms of computational space and time. This raises the question of whether or not the "cheaper" spectral Max Cut algorithms can perform competitively to the SDP in practice. Furthermore, if yes, can the approximation guarantees be improved? As demonstrated, the spectral and greedy algorithms provide a significant speed advantage over the SDP. Additionally, they often compute cuts better than or comparable to the cuts returned by the SDP, despite the disparity in approximation guarantees. The results of this experiment appear to illustrate spectral algorithms are competitive with the SDP algorithm in practice. This suggests that the investigation into approximation guarantees is a direction for further theoretical study.

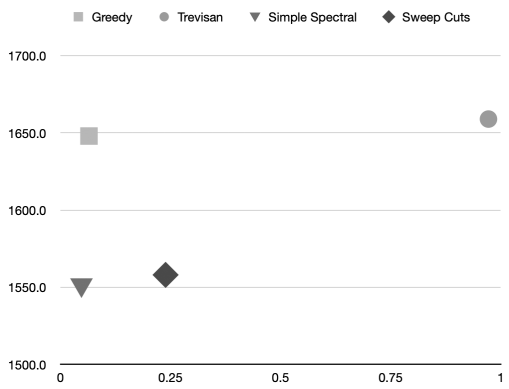
In terms of practical implementations, for the graphs that the SDP seems to perform better on, one could consider running Trevisan's algorithm for even more than 5 iterations and choosing the best cut returned. The magnitude of the speed advantage of Trevisan's algorithm would allow for many runs before being as costly as the SDP, especially since the initial eigenvector only needs to be computed once. Additionally, finding a viable heuristic to use when choosing multiple t^2 values would also provide implementation benefits. We attempted to improve Trevisan's algorithm through drawing additional random t^2 values and greedily choosing one. However, it is not obvious that this choice in heuristic is optimal.

Table 6 The time in seconds each algorithm took to compute a cut of a graph from the Network Repository arising in the real-world.

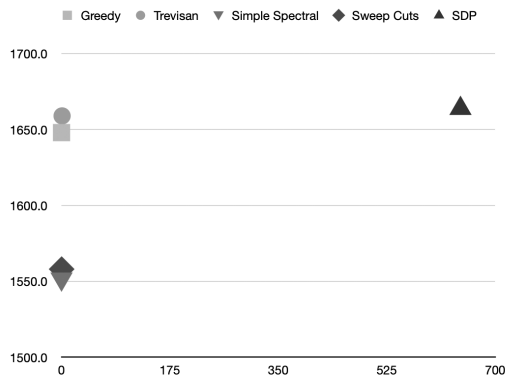
Graph	# vertices	# edges	Greedy	Trevisan	Simple Spectral	Sweep Cuts	SDP
ENZYMES8	88	133	$\mathbf{1.370 \times 10^{-3}}$	4.342×10^{-2}	1.776×10^{-1}	1.015×10^{-2}	2.356×10^{1}
eco-stmarks	54	356	$\mathbf{5.900 \times 10^{-4}}$	2.146×10^{-2}	1.993×10^{-1}	2.597×10^{-2}	6.939
johnson16-2-4	120	5460	2.519×10^{-2}	1.603×10^{-1}	$\mathbf{2.600 \times 10^{-3}}$	8.544×10^{-2}	8.178×10^{-1}
hamming6-2	64	1824	3.540×10^{-3}	5.081×10^{-2}	$\mathbf{1.190 \times 10^{-3}}$	1.558×10^{-2}	9.926×10^{-1}
ia-infect-hyper	113	2196	8.920×10^{-3}	1.248×10^{-1}	$\mathbf{3.280 \times 10^{-3}}$	4.649×10^{-2}	6.279
soc-dolphins	62	159	$\mathbf{4.600 \times 10^{-4}}$	1.845×10^{-2}	8.900×10^{-4}	1.690×10^{-3}	2.464
email-enron-only	143	623	5.960×10^{-3}	1.174×10^{-1}	$\mathbf{5.650 \times 10^{-3}}$	1.575×10^{-2}	5.681×10^{1}
dwt_209	209	976	1.349×10^{-2}	3.012×10^{-1}	$\mathbf{8.380 \times 10^{-3}}$	4.641×10^{-2}	7.073×10^{1}
inf-USAir97	332	2126	$\mathbf{5.780 \times 10^{-2}}$	2.944	7.350×10^{-2}	2.258×10^{-1}	3.361×10^{2}
ca-netscience	379	914	$\mathbf{2.590 \times 10^{-2}}$	5.124×10^{-1}	8.440×10^{-2}	1.146×10^{-1}	3.584×10^{2}
ia-infect-dublin	410	2765	6.480×10^{-2}	9.720×10^{-1}	$\mathbf{4.770 \times 10^{-2}}$	2.387×10^{-1}	6.438×10^{2}
road-chesapeake	39	170	$\mathbf{4.000 \times 10^{-4}}$	8.000×10^{-3}	5.000×10^{-4}	1.200×10^{-3}	2.759×10^{-1}
Erdos991	492	1417	$\mathbf{4.490 \times 10^{-2}}$	2.634	6.090×10^{-2}	1.933×10^{-1}	5.143×10^{2}
dwt_503	503	3265	7.240×10^{-2}	2.039	$\mathbf{6.640 \times 10^{-2}}$	3.471×10^{-1}	1.081×10^{3}
p-hat700-1	700	60999	1.264	1.009×10^{1}	$\mathbf{1.591 \times 10^{-1}}$	9.273	1.270×10^{3}
DD687	725	2600	$\mathbf{9.390 \times 10^{-2}}$	4.332	3.816×10^{-1}	6.521×10^{-1}	3.320×10^{3}
email-univ	1133	5451	$\mathbf{2.081 \times 10^{-1}}$	1.345×10^{1}	1.179	2.703	7.572×10^{3}

Table 7 The value of the cut each algorithm returned for a graph from the Network Repository.

Graph	# vertices	# edges	Greedy	Trevisan	Simple Spectral	Sweep Cuts	SDP
ENZYMES8	88	133	1.170×10^{2}	$\mathbf{1.260 \times 10^{2}}$	$\mathbf{1.260 \times 10^{2}}$	$\mathbf{1.260 \times 10^{2}}$	$\mathbf{1.260 \times 10^{2}}$
eco-stmarks	54	356	8.891×10^{2}	$\mathbf{1.190 \times 10^{3}}$	9.354×10^{2}	9.354×10^{2}	9.601×10^{2}
johnson16-2-4	120	5460	$\mathbf{3.036 \times 10^{3}}$	$\mathbf{3.036 \times 10^{3}}$	2.958×10^{3}	2.986×10^{3}	2.918×10^{3}
hamming6-2	64	1824	$\mathbf{9.920 \times 10^{2}}$	$\mathbf{9.920 \times 10^{2}}$	9.680×10^{2}	9.690×10^{2}	9.760×10^{2}
ia-infect-hyper	113	2196	1.213×10^{3}	$\mathbf{1.233 \times 10^{3}}$	1.227×10^{3}	1.227×10^{3}	1.211×10^{3}
soc-dolphins	62	159	1.120×10^{2}	1.120×10^{2}	1.190×10^{2}	$\mathbf{1.210 \times 10^{2}}$	1.150×10^{2}
email-enron-only	143	623	3.920×10^{2}	$\mathbf{4.130 \times 10^{2}}$	3.710×10^{2}	3.800×10^{2}	3.960×10^{2}
dwt_209	209	976	5.250×10^{2}	5.270×10^{2}	5.250×10^{2}	5.270×10^{2}	$\mathbf{5.400 \times 10^{2}}$
inf-USAir97	332	2126	9.661×10^{1}	9.820×10^{1}	8.184×10^{1}	9.337×10^{1}	$\mathbf{1.074 \times 10^{2}}$
ca-netscience	379	914	5.830×10^{2}	5.880×10^{2}	5.270×10^{2}	5.270×10^{2}	$\mathbf{6.110 \times 10^{2}}$
ia-infect-dublin	410	2765	1.648×10^{3}	1.659×10^{3}	1.550×10^{3}	1.558×10^{3}	$\mathbf{1.664 \times 10^{3}}$
road-chesapeake	39	170	1.230×10^{2}	1.230×10^{2}	1.210×10^{2}	1.230×10^{2}	$\mathbf{1.250 \times 10^{2}}$
Erdos991	492	1417	9.330×10^{2}	$\mathbf{9.340 \times 10^{2}}$	7.350×10^{2}	7.580×10^{2}	9.240×10^{2}
dwt_503	503	3265	1.822×10^{3}	1.822×10^{3}	$\mathbf{1.921 \times 10^{3}}$	$\mathbf{1.921 \times 10^{3}}$	1.909×10^{3}
p-hat700-1	700	60999	3.261×10^{4}	3.269×10^{4}	3.215×10^{4}	$\mathbf{3.305 \times 10^{4}}$	3.304×10^{4}
DD687	725	2600	1.669×10^{3}	1.671×10^{3}	1.616×10^{3}	1.617×10^{3}	$\mathbf{1.680 \times 10^{3}}$
email-univ	1133	5451	$\mathbf{3.546 \times 10^{3}}$	$\mathbf{3.546 \times 10^{3}}$	3.341×10^{3}	3.344×10^{3}	3.264×10^{3}

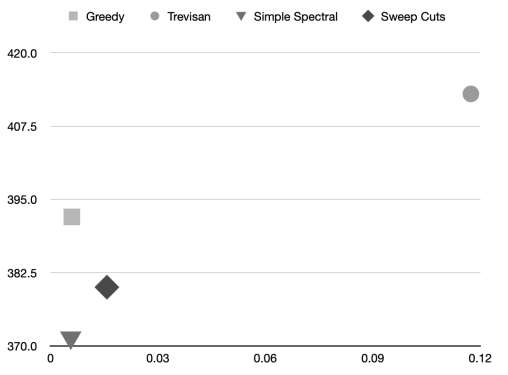


(a) All tested algorithms excluding the SDP.

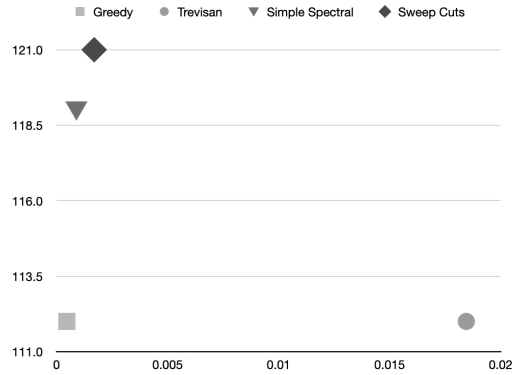


(b) All tested algorithms.

Figure 5 Plots depicting the computation time and returned cut values of algorithms on the ia-infect-dublin graph. The X and Y axes are the runtime in seconds and the returned cut value, respectively.



(a) email-enron-only.



(b) soc-dolphins.

Figure 6 Plots depicting the computation time and returned cut values of algorithms excluding the SDP on the email-enron-only and soc-dolphins graphs. The X and Y axes are the runtime in seconds and the returned cut value, respectively.

In particular, perhaps it is more useful to draw a fixed number of t^2 values but finish the algorithm's entire partitioning instead of estimating at that point in time. The magnitude by which the spectral algorithms are faster than the SDP allows this to be a reasonable option.

It is also worth noting the performances of the simple spectral and sweep cuts algorithms. Particularly for large graphs, these two algorithms along with the greedy algorithm are much faster than even Trevisan's algorithm, with the simple spectral almost always being several times faster than greedy (and sweep cuts being slightly slower than greedy). It is known that the greedy algorithm has a .5 approximation guarantee, but to the best of our knowledge, there is no known approximation guarantee for the simple spectral or sweep cuts algorithms. This raises the question of whether any approximation guarantee can be proven for either of these algorithms. A desired guarantee would be greater than greedy's .5; given the performance results presented here, it seems possible that this is achievable.

Relatedly, there is no indication that Soto's .614 approximation guarantee for Trevisan's algorithm is tight. It is clear that the algorithm often far surpasses this in practice. Can the analysis of this algorithm be improved?

References

1. Francisco Barahona, Martin Grötschel, Michael Jünger, and Gerhard Reinelt. An application of combinatorial optimization to statistical physics and circuit layout design. *Operations Research*, 36(3):493–513, 1988. doi:10.1287/opre.36.3.493.
2. Jonathan W. Berry and Mark K. Goldberg. Path optimization for graph partitioning problems. *Discrete Appl. Math.*, 90(1–3):27–50, January 1999. doi:10.1016/S0166-218X(98)00084-5.
3. Alberto Bertoni, Paola Campadelli, and Giuliano Grossi. An approximation algorithm for the maximum cut problem and its experimental analysis. *Discrete Applied Mathematics*, 110:3–12, 2001. doi:10.1016/S0166-218X(00)00299-7.
4. F. Della Croce, M.J. Kaminski, and V.Th. Paschos. An exact algorithm for MAX-CUT in sparse graphs. *Operations Research Letters*, 35(3):403–408, 2007. doi:10.1016/j.orl.2006.04.001.
5. Oliver Dolezal, Thomas Hofmeister, and Hanno Lefmann. A comparison of approximation algorithms for the maxcut-problem, May 2000. doi:10.17877/DE290R-5013.
6. Iain Dunning, Swati Gupta, and John Silberholz. What works best when? A systematic evaluation of heuristics for max-cut and QUBO. *INFORMS Journal on Computing*, 30(3):608–624, 2018. doi:10.1287/ijoc.2017.0798.
7. Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, November 1995. doi:10.1145/227683.227684.
8. Alexander Golovnev. New upper bounds for MAX-2-SAT and MAX-2-CSP w.r.t. the average variable degree. In Dániel Marx and Peter Rossmanith, editors, *Parameterized and Exact Computation*, pages 106–117, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
9. F. Hadlock. Finding a maximum cut of a planar graph in polynomial time. *SIAM Journal on Computing*, 4(3):221–225, 1975. doi:10.1137/0204019.
10. Timotej Hrga, Borut Lužar, Janez Povh, and Angelika Wiegele. BiqBin: Moving boundaries for NP-hard problems by HPC. In Ivan Dimov and Stefka Fidanova, editors, *Advances in High Performance Computing*, pages 327–339, Cham, 2021. Springer International Publishing.
11. Timotej Hrga and Janez Povh. MADAM: A parallel exact solver for max-cut based on semidefinite programming and ADMM. *Comput. Optim. Appl.*, 80(2):347–375, November 2021. doi:10.1007/s10589-021-00310-6.
12. Richard Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, volume 40, pages 85–103, January 1972. doi:10.1007/978-3-540-68279-0_8.

13 Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell. Optimal inapproximability results for MAX-CUT and other 2-variable CSPs? *SIAM J. Comput.*, 37(1):319–357, April 2007. doi:10.1137/S0097539705447372.

14 Nathan Krislock, Jérôme Malick, and Frédéric Roupin. BiqCrunch: A semidefinite branch-and-bound method for solving binary quadratic problems. *ACM Trans. Math. Softw.*, 43(4), January 2017. doi:10.1145/3005345.

15 G.I. Orlova and Ya. G. Dorfman. Finding the maximal cut in a graph. *Engineering Cybernetics*, 10(3):502–506, 1972.

16 Jan Poland and Thomas Zeugmann. Clustering pairwise distances with missing data: Maximum cuts versus normalized cuts. In Ljupčo Todorovski, Nada Lavrač, and Klaus P. Jantke, editors, *Discovery Science*, pages 197–208, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

17 Svatopluk Poljak and Zsolt Tuza. Maximum cuts and largest bipartite subgraphs. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 20, pages 181–244, 1995. doi:10.1090/dimacs/020/04.

18 Gerhard Reinelt. TSPLIB – A traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

19 Franz Rendl, Giovanni Rinaldi, and Angelika Wiegele. Solving max-cut to optimality by intersecting semidefinite and polyhedral relaxations. *Mathematical Programming*, 121:307–335, February 2010. doi:10.1007/s10107-008-0235-8.

20 Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL: https://networkrepository.com.

21 Sartaj Sahni and Teofilo Gonzalez. P-Complete approximation problems. *J. ACM*, 23(3):555–565, July 1976. doi:10.1145/321958.321975.

22 José A. Soto. Improved analysis of a max-cut algorithm based on spectral partitioning. *SIAM Journal on Discrete Mathematics*, 29(1):259–268, 2015. doi:10.1137/14099098X.

23 Luca Trevisan. Max cut and the smallest eigenvalue. *SIAM Journal on Computing*, 41(6):1769–1786, 2012. doi:10.1137/090773714.

24 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2):357–365, 2005. Automata, Languages and Programming: Algorithms and Complexity (ICALP-A 2004). doi:10.1016/j.tcs.2005.09.023.

Digraph k-Coloring Games: From Theory to Practice

Andrea D'Ascenzo
Department of Computer Science, Information Engineering and Mathematics, University of L'Aquila, Italy

Mattia D'Emidio
Department of Computer Science, Information Engineering and Mathematics, University of L'Aquila, Italy

Michele Flammini
Gran Sasso Science Institute, L'Aquila, Italy

Gianpiero Monaco
Department of Computer Science, Information Engineering and Mathematics, University of L'Aquila, Italy

Abstract

We study digraph k-coloring games where agents are vertices of a directed unweighted graph and arcs represent agents' mutual unidirectional idiosyncrasies or conflicts. Each agent can select one of k different colors, and her payoff in a given state is given by the number of outgoing neighbors with a different color. Such games model lots of strategic real-world scenarios and are related to several fundamental classes of anti-coordination games. Unfortunately, the problem of understanding whether an instance of the game admits a pure Nash equilibrium is NP-complete [33]. Therefore, in the last few years a relevant research focus has been that of designing polynomial time algorithms able to compute *approximate* Nash equilibria, i.e., states in which no agent, changing her strategy, can improve her payoff by some bounded multiplicative factor. The only two known algorithms in this respect are those in [14]. While they provide theoretical guarantees, their practical performance over real-world instances so far has not been investigated. In this paper, under the further motivation of the lack of practical approximation algorithms for the problem, we experimentally evaluate the above algorithms with the conclusion that, while they were suitably designed for achieving a bounded worst case behavior, they generally have a poor performance. Therefore, we next focus on classical best-response dynamics, and show that, despite of the fact that they might not always converge, they are very effective in practice. In particular, we provide a strong empirical evidence that they outperform existing methods, since surprisingly they quickly converge to exact Nash equilibria in almost all instances arising in practice. This also shows that, while this class of games is known to not always possess pure Nash equilibria, in almost all cases such equilibria exist and can be efficiently computed, even in a distributed uncoordinated way by a decentralized interaction of the agents.

2012 ACM Subject Classification Theory of computation → Algorithmic game theory and mechanism design; Theory of computation → Quality of equilibria; Theory of computation → Design and analysis of algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases Algorithmic Game Theory, Coloring Games, Experimental Algorithmics, Exact vs Approximate Nash Equilibria, Decentralized Dynamics

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.20

Supplementary Material *Software (Source Code)*: `https://github.com/buildfreak/sea_2022`
archived at `swh:1:dir:f09c580538ebfe5b245c15fb6a24810fbb817fa2`

1 Introduction

In this paper we consider digraph k-coloring games. We are given an unweighted directed graph where vertices represent selfish autonomous agents and arcs mutual unidirectional idiosyncrasies or conflicts. Moreover, we have a set of $k \geq 2$ available colors denoting agents'

available choices or strategies. A state of the game is a vertex coloring, induced by the choices of the agents. The objective of each agent is that of maximizing her own payoff, which is defined as the number of outgoing neighbors with a color different from hers.

Digraph k-coloring games form some of the basic payoff structures in algorithmic game theory, they fall within the fundamental classes of (anti-)coordination games (see the Related Work section) and model many relevant real-world scenarios. Typical examples are wireless networks in which radio stations wish to select the transmission frequency not used by the maximum number of neighboring stations within their range in order to limit interferences, or social networks in which members must be split in groups and want to maximize the number of enemies they do not end together with, or markets in which sellers aim to locate their activities as far as possible from their direct competitors.

A classical solution concept of stable outcomes in scenarios with selfish and autonomous agents is the *(pure) Nash equilibrium* (NE, for short) where no agent can improve her payoff by unilaterally changing her strategy. In our setting of digraph k-coloring games, a NE is a coloring where no vertex can improve her payoff by changing color. The NE is one of the most important concepts in game theory. As such, its efficient computation is one of the most important problems in algorithmic game theory [24]. In the specific setting when the input graph is undirected, the k-coloring game is a potential game [39], which implies that a NE always exists. In particular, when the graph is unweighted undirected, the dynamics (where at each step one agent performs an improving move) always converges to a NE in a polynomial number of steps [29, 33]. However, in the more general case of directed graphs, for any $k \geq 2$ it is known that even the problem of understanding whether digraph k-coloring games admit a NE is NP-complete [33]. Therefore, like in a variety of games falling in this class, where Nash equilibria do not exist or cannot be computed in polynomial time, researchers focused on the milder form of *approximate* equilibria [14]. Namely, a state is called a γ-Nash equilibrium (γ-NE, for short), for some $\gamma \geq 1$, if no agent can strictly improve her payoff by a multiplicative factor of γ by changing her strategy.

To the best of our knowledge, the only advancements in this direction about digraph k-coloring games are those presented in [14]. In details, the authors: (i) show that a pure NE (i.e., with $\gamma = 1$) is not guaranteed to exist for any number of players n and colors $k < n$; (ii) observe that, for any $k \geq 2$, a pure NE exists and can be found in polynomial time if the graph is bipartite or directed acyclic; (iii) notice that, for the case of $k = 2$, a γ-Nash equilibria might not exist for any bounded value of γ. In fact, it is easy to see that in any 2-coloring of a directed cycle with an odd number of vertices there is always at least one vertex with zero utility (and hence no NE can have bounded γ); (iv) present a deterministic polynomial time algorithm (called AP1) that, for any $k \geq 3$, returns a k-coloring that is a Δ_o-Nash equilibrium, where Δ_o is the maximum outgoing degree of the input digraph G; (v) present a randomized algorithm (denoted as LLL-SPE in what follows) that, by exploiting the constructive version of the well-known Lovász Local Lemma (LLL, for short, from here onwards) [40], computes a constant approximate Nash equilibrium in polynomial expected running time. Specifically, the algorithm works for any constant $k \geq 2$ and for the special class of n-vertex digraphs having $\Omega(\log n)$ minimum outgoing degree.

However, notwithstanding the aforementioned progress, very little is known about the true performance of existing algorithms in practice. Specifically, it remains unclear which method performs best, in terms of approximation and running time, for inputs arising from real-world application domains. On the one hand, in fact, no average case analysis is known for AP1 and hence it is difficult to capture how much the upper bound on the approximation is pessimistic

on real inputs. On the other hand, algorithm LLL-SPE guarantees constant approximation with a value of γ that is quite large, and only for a class of digraphs that appears to be quite infrequent in the application domains of the considered game. Nothing is known about the behavior of LLL-SPE on general digraphs. Moreover, to the best our knowledge, no experimental evaluations have been conducted on the subject, nor any algorithm has been shown to perform nicely in practice.

Our Contribution. In this paper we attack such research questions through algorithmic experimentation. In particular, our contribution is twofold. First, we implement and test both AP1 and LLL-SPE[1] against both artificial and real-world graphs of heterogeneous sizes and topologies, and different values of k. We observe that the measured approximation achieved by the two solutions, on the considered combinations of inputs and values of k, is unsatisfactory. Specifically, the obtained colorings are comparable, in terms of measured γ, to those one can produce by simply assigning colors to agents/vertices uniformly at random. Then, motivated by such unsatisfactory performance, under the further motivation of the lack of practical approximation algorithms for the problem, we focus on the classical best-response paradigm, where at each step an agent having an improving move is selected uniformly at random (we refer to this approach as algorithm BEST-RESP in the following). Such paradigm induces a dynamics on the coloring that does not offer any upper bound on the provided approximation and might not always stabilize. Hence, to evaluate the practical effectiveness of BEST-RESP, we implement it and compare it against AP1 and LLL-SPE, through an extensive experimental evaluation again involving large sets of heterogeneous inputs and values of k. In our study we consider, as main performance indicators, both approximation (namely measured γ) and computational time. Plus, we assess other metrics we introduce here, namely *average payoff* and *fraction of unhappy vertices*, which naturally characterize the practical effectiveness of considered algorithms in the application domains where they are intended to be applied. Our experimental results provide strong empirical evidences of BEST-RESP being the most effective solution among the tested ones, and hence advised for practical usage. In fact, in essentially all considered combinations of inputs and values of $k \geq 3$, BEST-RESP results to be the best performing method in terms of approximation. Moreover, remarkably, in the majority of the cases it converges to a pure NE (i.e. computes exact, non-approximate solutions) in a reasonably low running time, even for large graphs. In the (few) remaining cases, BEST-RESP is comparable to other methods in terms of approximation, but achieves better results in terms of average payoff and fraction of unhappy vertices. Our results trigger new theoretical questions and demand for new investigation on the possibility of achieving constant approximation for general digraphs or computing a pure NE in polynomial time for special classes of digraphs. Our study also highlights that, while this class of games is known to not always possess NE, in almost all cases such equilibria exist and can be efficiently computed, even in a distributed uncoordinated way by a decentralized interaction of the agents, such as that underlying BEST-RESP (while AP1 and LLL-SPE are not naturally suited for distributed implementations).

Related Work. k-coloring games in undirected graphs have been first investigated in [29,33], where it is shown that a NE always exists and can be computed in polynomial time if the graph is unweighted. When the graph is weighted, instead, a NE always exists but the problem of computing it is PLS-complete, i.e. conjectured difficult, even for $k = 2$ [46] (notice

[1] More precisely, a slightly modified version of LLL-SPE that incorporates a stopping criterion to apply it on general digraphs, where the convergence is not guaranteed since the LLL might not hold.

that graph 2-coloring games are exactly max cut games). In [43] it is proven that NE can be computed in polynomial time for graph 2-coloring games if the maximum degree of the graph is at most 3. A related investigation for the same class of games is presented in [5, 12], where the authors give an algorithm that, for any $\epsilon > 0$, computes a $(3+\epsilon)$-equilibrium in time polynomial in $\frac{1}{\epsilon}$ and in the instance size. All above results exploit the potential function method. Unfortunately, digraph k-coloring games (where the graph is directed) do not admit a potential function and the problem of understanding whether they admit a Nash equilibrium is NP-complete for any fixed $k \geq 2$, even in the unweighted case [33]. The performance of NE in general graph k-coloring games has been addressed in [25], while the authors of [15] consider Nash equilibria where players also have an extra profit depending on the chosen color. Finally, the authors of [13, 28] study the existence of strong NE, i.e. resistant to coalitional moves, again for graph k-coloring games.

Digraph k-coloring games are related to many fundamental games that have been widely studied in recent literature. One example is graphical games, first introduced in [32], where the payoff of each agent depends only on the strategies of her neighbours in a given social knowledge graph defined over the set of the agents. An interesting class of graphical games, related to digraph k-coloring games, is that of graphical congestion games [7]. Digraph k-coloring games can also be seen as a particular form of hedonic games with an upper bound (i.e., k) to the number of coalitions (see [4] for a brief introduction to hedonic games). Specifically, given a k-coloring, agents having a same color can be seen as members of the same coalition in the corresponding hedonic game. In order to get the equivalence between the two games, the so–called hedonic utility of an agent v has to be defined as the overall number of her neighbors minus the number of agents of her neighborhood that are in the same coalition. Issues related to NE in hedonic games have been largely investigated under several assumptions (see [6, 26, 27, 37, 38, 42] and references therein).

While coloring games are the paradigmatic class of anti-coordination games, another very active stream of research has been dedicated to coordination games, where agents instead are rewarded for choosing common strategies rather than different ones. Results about coordination games can be found in [2, 3, 44]. Finally, the authors of [41] study games where Nash equilibria are proper vertex colorings in an undirected unweighted graphs setting.

Another prominent class of games, generalizing coloring games and bearing strong connections with coalition, coordination, and anti-coordination games is the one of the so-called *polymatrix coordination games* [51]. Here each agent v must select an action in her strategy set, and the utility is given by the preference she has for her action plus, for each neighbor w, a payoff which strictly depends on the mutual actions played by v and w. Polymatrix games have been thoroughly studied both in some classical works [23, 30, 31, 36] and also, more recently, with a special focus on equilibria [11, 20, 21, 44].

2 Notation and Background

Graph Notation. We assume we are given an unweighted directed graph, or simply digraph, $G = (V, A)$, without self loops, having $|V| = n$ vertices and $|A| = m$ arcs connecting vertices of G. Any arc $(v, w) \in A$ is *directed* from vertex v to vertex w. An arc is said to be an *outgoing arc* from vertex v (*incoming arc* to vertex w, respectively) if such arc is any arc $(v, w) \in A$.

Given a vertex $v \in V$, we denote by δ_o^v (δ_i^v, respectively) the *outgoing degree* of v (the *incoming degree* of v, respectively), that is the number of outgoing arcs from v (the number of incoming arcs to v, respectively) in G. The set of *outgoing neighbors* $N_{out}(v)$ of a vertex v is the set of vertices induced by all outgoing arcs of v, i.e. $N_{out}(v) = \{w : w \in V \wedge (v, w) \in A\}$.

Moreover, we denote by $d_o = \min_{v \in V} \delta_o^v$ and $\Delta_o = \max_{v \in V} \delta_o^v$ the minimum and maximum outgoing degree of G, respectively. Similarly, we call $\overline{d_o}$ and $\overline{\overline{d_o}}$ the average and median outgoing degree, respectively, and $\Delta_i = \max_{v \in V} \delta_i^v$ the maximum incoming degree of G.

Digraph k-coloring games. In a *digraph k-coloring game* we are given a digraph $G = (V, A)$, without self loops, in which each vertex $v \in V$ is a *selfish agent*, and a set C of $|C| = k$ available *colors*. Each agent has a same *set of actions* (i.e. a same *strategy set*), which is the set of the k available colors. A *state* of the game $c = \{c_1, \ldots, c_n\}$ is a *k-coloring* for graph G (simply *coloring* when k is clear from the context), where each c_v is the color chosen by each agent $v \in V$ (i.e., a number from 1 to k). In what follows, we will use vertex and agent interchangeably. Given a coloring c, the *payoff* $\mu_c(v)$ (often also referred to as the *utility*) of an agent v is the number of outgoing neighbors whose color in c is different from that of v, i.e. $\mu_c(v) = |\{(v, w) \in A : c_v \neq c_w\}|$. Observe that, for a vertex v, having $N_{out}(v) = \emptyset$ or $c_v = c_w \ \forall w \in N_{out}(v)$ implies $\mu_c(v) = 0$.

A coloring c is a *pure Nash equilibrium* (a.k.a. *stable equilibrium*, denoted in what follows as pure NE for short), if no agent v can improve her payoff by unilaterally changing strategy (i.e., color). Formally, if we use (c_{-v}, c'_v) to denote the coloring obtained, from a coloring $c = \{c_1, \ldots, c_n\}$, by changing the strategy of agent v from c_v to c'_v, then a coloring c is a pure NE if $\mu_c(v) \geq \mu_{(c_{-v}, c'_v)}(v)$, for any possible color $c'_v \in C$ and for any vertex $v \in V$.

Unfortunately, a pure NE is not guaranteed to exist even for a large number of available colors k. In particular, while it is easy to observe that there always exists a NE with $|C| = n$ colors, by just assigning to each vertex a different color, it is also possible to prove that, for arbitrary values of $n \geq 3$, and for any fixed k such that $1 < k \leq n-1$, there exist instances of the digraph k-coloring game that do not admit any pure NE [14]. Furthermore, it is known that, for general digraphs, the problem of determining whether the digraph k-coloring game admits a pure NE is NP-complete, for all $k \geq 2$. Moreover, the cases of bipartite graphs and directed acyclic graphs are simpler and can be handled in polynomial time [14].

Therefore, from here onward we consider the notion of *approximate Nash equilibrium*, defined as follows: a state or coloring c is a γ-approximate Nash equilibrium (simply γ-NE or *γ-stable* equilibrium for short), for some $\gamma \geq 1$, if no agent can strictly improve her payoff by a multiplicative factor of γ, by changing color. More formally, we have that a coloring $c = \{c_1, \ldots, c_n\}$ is a γ-NE when, for any possible color $c'_v \in C$ and for any vertex $v \in V$, we have:

$$\gamma \cdot \mu_c(v) \geq \mu_{(c_{-v}, c'_v)}(v).$$

Each vertex in a γ-NE is said to be *γ-happy*. Viceversa, a vertex v is *γ-unhappy*, for some $\gamma \geq 1$, if and only if $\gamma \cdot \mu_c(v) < \mu_{(c_{-v}, c'_v)}(v)$ for some color $c'_v \in C$. In other words, a γ-unhappy vertex can strictly improve her payoff by a multiplicative factor of γ, by changing color. In this case, we define the *potential payoff* $\pi_c(v)$ of vertex v as the maximum payoff a vertex v can achieve by unilaterally changing its color to another color of the strategy, that is $\pi_c(v) = \max_{c'_v \in C} \mu_{(c_{-v}, c'_v)}(v)$. Consequently, we call the *potential color* of a vertex v to be the color of C inducing the potential payoff, i.e. $\psi_c(v) = argmax_{c'_v \in C} \mu_{(c_{-v}, c'_v)}(v)$.

By analogy, we introduce the special notion of (simply) unhappy vertex, which occurs for a γ-unhappy vertex when $\gamma = 1$. Specifically, given a coloring c, we say a vertex v is *unhappy* if and only if $\mu_c(v) < \mu_{(c_{-v}, c'_v)}(v)$ for some color $c'_v \in C$, i.e. when the vertex can strictly improve her payoff by changing color unilaterally (c is not a pure NE). Clearly, if a vertex v is unhappy we have that $\mu_c(v) < \pi_c(v)$ and $c_v \neq \psi_c(v)$ while, if $\pi_c(v) = \gamma \mu_c(v)$, for all vertices $v \in V$ for some $\gamma > 0$, we have a γ-NE.

3 Algorithms for Digraph k-coloring Games

In this section, we first summarize the main characteristics of known algorithms for the computation of approximate γ-NE with guarantees on the achieved γ, namely AP1 and LLL-SPE [14]. Note that we equip the latter with a stopping criterion to apply it to general digraphs. Then, we present a formal description of algorithm BEST-RESP, an iterative best-response-based approach that computes a k-coloring without any guarantee on the achieved approximation. We will show in the experimental section how this simple strategy results to be the best solution in practice.

Algorithm AP1. In this paragraph, we provide a brief description of algorithm AP1, proposed in [14]. Given a digraph G, algorithm AP1, is deterministic and, for any $k \geq 3$, returns a k-coloring such that every vertex v with $\delta_o^v \geq 1$ has payoff at least 1, in polynomial time. Clearly this corresponds to a Δ_o-NE since Δ_o is the maximum payoff any agent can achieve.

The algorithm is iterative and works as follows (see [14] for more details and for the pseudocode of the algorithm): at each iteration the algorithm visits the graph induced by the uncolored vertices and detects a cycle or a path (the latter happens only when the visit reaches a vertex with zero outgoing neighbors in the induced subgraph). Then, it colors the vertices of the cycle or the path by alternating three colors, say colors 1, 2 and 3, in a way that every vertex gets payoff of at least 1, as follows. If the induced subgraph is a cycle, then the algorithm considers vertices of the cycle in clockwise order and assigns the colors by following such order, starting by any vertex and alternating the three colors. Viceversa, if the subgraph is a path from vertex v to vertex w, then two cases can occur. If the arc $(w, u) \in A$ then it colors w by a different color with respect to the already colored vertex u. Otherwise, it means that $\delta_o^w = 0$ and we can assign any color to w. Then, it alternates colors (in this case two colors are enough) for the other vertices of the path considered in the reverse order starting from w. Observe that it is easy to show, by analyzing the pseudocode of AP1, given in [14], that the following holds.

▶ **Lemma 1.** *Algorithm AP1 runs in $O(\Delta_o nm)$ worst case time.*

Algorithm LLL-GEN. In this paragraph, we introduce algorithm LLL-GEN, a generalization of the randomized algorithm LLL-SPE presented in [14]. Observe that algorithm LLL-SPE is based on the *Lovász Local Lemma* (LLL, for short) [47], which can be used for proving that there is a positive probability that a random assignment of the k colors to the vertices of digraphs returns a constant approximate NE, for any value of $k \geq 2$, for any graph $G = (V, A)$ such that the outgoing degree of any $v \in V$ is $\delta_o^v = \Omega(\log \Delta_o + \log \Delta_i)$. More precisely, algorithm LLL-SPE is designed to compute such a NE, in polynomial expected running time, by exploiting the constructive version of the LLL provided by [40]. Specifically, the algorithm, starting from a random assignment of the colors, repeatedly and iteratively applies operations of random *resampling* of the colors of γ-unhappy vertices of the graph, and to vertices in their dependency sets, until the coloring converges to a γ-NE. The dependency set of a vertex v is the set of vertices whose status of being unhappy/happy is influenced by (influences, resp.) the status of v, namely v's neighbours, their incoming neighbors, and v itself (see [14]).

The convergence is guaranteed to occur, for LLL-SPE, w.h.p., if the LLL is satisfied for a given graph, i.e. when $\delta_o^v = \Omega(\log \Delta_o + \log \Delta_i)$. In details, in order to apply the LLL: (i) a "bad event" I_v is defined over each vertex v of the graph when v is not γ-happy; (ii) a bound to the maximum size of the dependency set, denoted as dep_v, of each bad event I_v is given

as $|dep_v| \leq \delta_o^v + \delta_i^v + \delta_o^v \delta_i^v$. If the LLL is satisfied, then algorithm LLL-SPE returns a γ-NE by performing a number of resampling operations of dependency sets of γ-unhappy vertices that is polynomial in expectation, for a constant value of γ, defined as follows:

$$\gamma = \max_{v \in V : \delta_o^v > 0} \frac{\text{max possible payoff}}{\text{min expected payoff}} \approx \max_{v \in V : \delta_o^v > 0} \frac{k}{k-1} + \left(\frac{k}{k-1}\right)^2 O\left((r - \frac{k}{k-1})^{-1}\right) \quad (1)$$

where $r = \delta_o^v / \log(\Delta_o \Delta_i)$. Note that vertices with zero out-degree are not considered in this formula, indeed they are always happy because they can always select a strategy that yields a non-zero payoff [14].

Now, observe that the above guarantees hold only for graphs whose structure satisfies the LLL. Thus, in order to generalize algorithm LLL-SPE and apply it to any digraph, motivated by the lack of algorithms to compute approximate NE in general digraphs, we introduce a stopping criterion to the maximum number of resampling operations that the algorithm can perform. This is a necessary change for our purpose of experimentally evaluating the behavior of such algorithm also in digraphs such that δ_o^v is not $\Omega(\log \Delta_o + \log \Delta_i)$, for some vertex v, as in this case LLL-SPE might not converge in polynomial time in expectation. In

Algorithm 1 Algorithm LLL-GEN.

Input: A digraph $G(V, A)$, a set C of $|C| = k$ available colors, a maximum number of iterations I
Output: A k-coloring c of G

1 $c \leftarrow$ random k–coloring of G with uniform probability $\frac{1}{k}$;
2 Compute threshold on γ as in Eq. 1;
3 $i \leftarrow 0$;
4 **while** $\exists$ *a* γ*-unhappy vertex and* $i < I$ **do**
5 Let S be the set of γ unhappy vertices; `/* c is not a γ-NE, hence S ≠ ∅ */`
6 Select randomly a vertex $v \in S$, with uniform probability $\frac{1}{|S|}$;
7 Randomly, uniformly, color vertices in dependency set dep_v of v;
8 $i \leftarrow i + 1$;
9 **return** c;

more details, besides the input digraph and the number of available colors k, the modified method LLL-GEN, summarized in Algorithm 1, takes as input also an integer value I and stops when either a γ-NE is found or a maximum number of iterations I is performed. By such modification, it is easy to see that the following holds.

▶ **Lemma 2.** *Algorithm* LLL-GEN *runs in* $O(I(n + \Delta_o + \Delta_i + \Delta_o \Delta_i))$ *worst case time.*

Proof. In each iteration, lines 5 and 7 are executed, with the former requiring $O(n)$ time while the latter performing a number of operations that is bounded by the maximum size of the dependency set of any vertex, i.e. $O(\Delta_o + \Delta_i + \Delta_o \Delta_i)$. ◀

Algorithm BEST-RESP**.** In this paragraph we describe an approach for computing approximated NE, named BEST-RESP, that is inspired to classical best-response dynamics, since they have been shown to be effective in practice to handle similar kinds of games [16, 49, 50]. More specifically, we consider what, in the literature, is sometimes referred to as myopic best-response method [16, 50], i.e. best-response dynamics where agents decide their strategy based on knowledge of their neighbors only. Such method is universally considered one of the most appealing strategies in this domain, since its update rules depend only on local

knowledge and hence they are very easy to be translated into distributed algorithms for decentralized systems of agents [8, 49]. We remark that this is a very relevant domain for digraph k-coloring games, and no distributed solution to compute approximate NE is currently known.

In more details, the idea underlying algorithm BEST-RESP is to start from a random coloring c. Then, if c is not a pure NE, the algorithm tries to iteratively improve it by applying best response strategies to unhappy vertices. More specifically, during a generic iteration the algorithm performs the following steps: (i) an unhappy vertex, say v, is selected uniformly at random; (ii) the color c_v of the unhappy vertex is set to the color in the strategy set C that maximizes her payoff (ties are broken arbitrarily), i.e. to $\pi_c(v)$. The process stops if a NE is reached, i.e. if no unhappy vertices exist in the graph, or when a maximum number of iterations I, given as part of the input, is performed. The pseudocode of procedure BEST-RESP is summarized in Algorithm 2. Given the above, the following result easily follows.

Algorithm 2 Algorithm BEST-RESP.

Input: A digraph $G(V, A)$, a set C of $|C| = k$ available colors, a maximum number I of iterations
Output: A k-coloring c of G

```
1 c ← random k–coloring of G with uniform probability 1/k;
2 i ← 0;
3 while ∃ an unhappy vertex and i < I do
4 |   Let S be the set of unhappy vertices;        /* c is not a NE, hence S ≠ ∅ */
5 |   Select randomly a vertex v ∈ S, with uniform probability 1/|S|;
6 |   c_v ← ψ_c(v);                                  // Color that maximizes payoff
7 |   i ← i + 1;
8 return c;
```

▶ **Lemma 3.** *Algorithm* BEST-RESP *runs in* $O(n\Delta_o I)$ *worst case time.*

Proof. Observe that executing line 1 takes $\Theta(n)$ time. Moreover, the block of Lines 3–7 is executed at most I times, and strictly less than I times only if a pure NE is found. To this regard, testing the existence of an unhappy vertex in each iteration requires computing the payoff of all vertices in the worst case, which takes $O(\Delta_o n)$ time since, for each vertex, we need to evaluate the colors of her outgoing neighbors. Selecting at random an unhappy vertex costs $|S|$ hence $O(n)$ time, and for said unhappy vertex an additional $\delta_o^v = O(\Delta_o)$ time in necessary to determine the color that maximizes her payoff. Thus, the claim follows. ◀

4 Experimentation

In this section, we describe the experimental study we conducted to assess the performance of algorithms for digraph k-coloring games. In particular, we implemented LLL-GEN, BEST-RESP, and AP1, and designed and deployed an experimental framework to evaluate said algorithms, with respect to various metrics of interest for the context of digraph k–coloring games, and on large set of meaningful input digraphs.

Test Environment and Implementation Details. Our entire test environment is based on NetworKit [48], a widely adopted open-source toolkit for implementing graph algorithms and performing network analysis tasks at scale. All our code is written in Python, with

some sub-routines in C++/Cython. All tests have been executed, through the Python 3.8 interpreter, under Linux (Kernel 5.3.0-53), on a workstation equipped with an Intel© Xeon© CPU E5-2643 3.40GHz and 128 GB of RAM.

Input Details. As input to our experiments, inspired by other empirical studies on graph algorithms [1,10,17–19,22], we employed a large dataset of digraphs, including: (i) real-world instances, taken from publicly available repositories [34,45]; (ii) artificial digraphs, built via well-established random generators, namely *Erdős-Rényi* and *Paley* models [9]. More details on used inputs, including sizes and main characteristics are reported in Table 1, while Figure 1 shows how outgoing degree is distributed in the considered inputs.

Table 1 Overview of used input digraphs. The first three columns contain dataset name, acronym, and type; the 4th and 5th columns show number of vertices and arcs of the digraph, respectively; columns from the 6th to the 8th report average, median and maximum outgoing degree, respectively. Finally, the 9th column highlights whether the graph is synthetic or real-world (● = true, ○ = false), while the last column specifies whether the LLL holds in the given graph (● = true, ○ = false). Inputs are sorted by $\overline{d_o}$, non-decreasing.

Dataset	Short	Type	|V|	|A|	$\overline{d_o}$	$\widetilde{d_o}$	Δ_o	S	LLL
Twitter	twi	digital social	23370	33101	1.42	0	238	○	○
facebook	fac	digital social	309717	472792	1.53	0	358	○	○
amazon	ama	ratings	80679	135336	1.68	2	9	○	○
Flight	flt	infrastructure	1226	2613	2.13	1	24	○	○
Peer2Peer	p2p	internet	62586	147892	2.36	0	78	○	○
Luxembourg	lux	road	30647	75546	2.47	3	9	○	○
rand3	rr3	random	10000	30000	3	3	3	●	●
rand4	rr4	random	10000	40000	4	4	4	●	●
Oregon-AS	ore	autonomous system	10670	44004	4.12	2	2312	○	○
rand5	rr5	random	10000	50000	5	5	5	●	●
Health	hea	human social	2539	12969	5.11	5	10	○	○
relativity	rel	collaboration	5242	28968	5.53	3	81	○	○
Linux	lin	community	30834	213424	6.92	5	243	○	○
Peer2PeerSm	spp	internet	10876	79988	7.35	5	103	○	○
Google	goo	hyperlinks (local)	15763	170335	10.81	8	852	○	○
Erdős-Rényi A	era	random	1000	12460	12.46	12	27	●	○
Blog	blg	interaction	1224	19022	15.54	7	256	○	○
Erdős-Rényi B	erb	random	1000	24943	24.94	25	45	●	●
wiki-Vote	wvt	voting	7115	201524	28.32	4	1065	○	○
Email	ema	interaction	1005	32128	31.97	21	345	○	○
Erdős-Rényi C	erc	random	1000	49924	49.92	50	74	●	●
Erdős-Rényi D	erd	random	1000	100025	100.03	100	134	●	●
Erdős-Rényi E	ere	random	1000	199443	199.44	199	238	●	●
paley601	pl1	random	601	180300	300	300	300	●	●
paley1181	pl2	random	1181	696790	590	590	590	●	●

Concerning parameter k, since no direct, well-established relationship is known between k itself and the approximation provided by the algorithms under study, our experimental trials consider carefully selected values of said parameter to investigate how the algorithms' behavior changes as k increases. Specifically, we consider both the reference case of $k = 3$ (representing a conjectured threshold on the computational hardness of the problem) and, as suggested by established guidelines for experimental algorithmics [35], to magnify the dependency on k in a reasonable number of tests, values of k ranging in the interval $[\max\{4, d_o\}, \Delta_o]$, evenly spaced as multiples of $\lfloor \frac{\Delta_o - d_o}{5} \rceil$. For iterative algorithms, we fix $I = n \log n$, since this leads in all cases to practical running times, even on the largest inputs.

Objectives of Experimentation. The purpose of our experimentation is twofold. First, we want to assess how effective are state-of-the-art solutions for digraph k-coloring games in practice. To this regard, we test our implementations of AP1 and LLL-GEN against all inputs and the mentioned values of k and compare, in terms of resulting γ, computed colorings

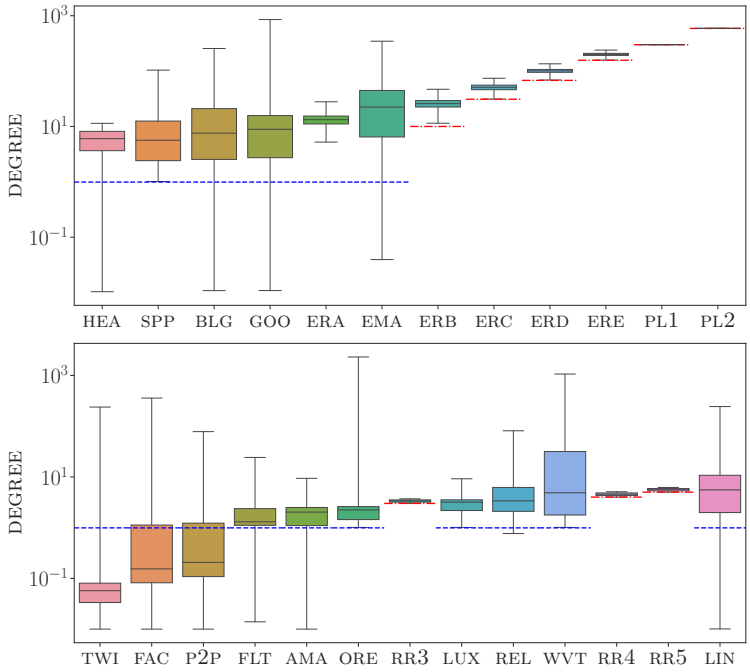


Figure 1 Distributions of outgoing vertex degrees of input graphs: each box plot shows minimum, 1st quartile, median, 3rd quartile, and maximum values. The red dotted line shows $\log \Delta_o + \log \Delta_i$ for graphs where the LLL holds. The blue line, viceversa, marks graphs where the LLL does not hold. Inputs are sorted, left to right, bottom to top, in non-decreasing order of $\overline{\overline{d_o}}$.

to randomly generated colorings, obtained by randomly, uniformly assigning a color of the strategy set C to each agent with probability $\frac{1}{|C|}$ (we denote this method by RANDOM in what follows). Second, we aim at establishing the practical effectiveness of algorithm BEST-RESP, hereby formalized. To this aim, we execute and compare obtained results to those achieved by AP1 and LLL-GEN for the same inputs and values of k. In all conducted tests, as a measure of quality of the computed colorings, we focus primarily on the obtained approximation. Specifically, for each graph G and value of k, and for each execution of each algorithm yielding a coloring c, we measure the *approximation ratio* with respect to a pure, exact NE, denoted as $\gamma(G, c)$, as follows:

$$\gamma(G,c) = \begin{cases} 0 & \textbf{if } \exists v \in V : c_v = c_u \ \forall\ u \in N_{out}(v) \\ \max_{v \in V : \delta_o^v > 0} \frac{\pi_c(v)}{\mu_c(v)} & \textbf{otherwise.} \end{cases}$$

In other words, if an algorithm computes a coloring c with $\gamma(G, c) > 1$ (a sufficient condition for the latter to happen is all vertices having strictly positive payoff, cf. Sec. 1), it follows that said coloring is a γ-NE. Moreover, $\gamma(G, c) = 1$ implies the computed coloring is a pure NE. Now, since algorithms LLL-GEN and BEST-RESP are iterative and induce dynamics on the state of the game (i.e. on the coloring), this part of the study measures also further performance indicators to better characterize the behavior of the considered algorithms. More in details, we measure: *average payoff* denoted as

$$\overline{P}(G,c) = \frac{\sum_{v \in V} \mu_c(v)}{|V|}$$

and *fraction of unhappy vertices*, denoted as

$$U(G,c) = \frac{|\{v \in V : v \text{ is unhappy}\}|}{|V|}.$$

Finally, for all algorithms, we measure the running time $T(G, c)$ spent to compute c. Notice that most of our experimental framework is written in Python, with some sub-routines in C++/Cython, with a fairly optimized code. We leave the problem open of achieving a faster version of all implementations through careful code tuning and porting to more high-performance programming languages., e.g. pure C++. In what follows, we will use LLG, BR, RND and AP1 to refer to algorithms LLL-GEN, BEST-RESP, RANDOM and AP1, respectively, for the sake of brevity.

Analysis. In Table 2 we present a summary of the results of our experimentation, for all input graphs and values of k. In details, for each considered metric, starting from the most relevant (i.e. approximation), we report the number of times each algorithm resulted to be the best performing one (2nd, 3rd, 4th best performing one, respectively) with respect to said metric. Note that, for those algorithms that resort to randomization, we executed three trials for each combination and computed average values of observed measures.

Table 2 Aggregate statistics for all tested algorithms with respect to the four considered indicators, for all combinations of inputs and values of k (for a total of 175 combinations).

metric	algorithm	best	2nd	3rd	worst
$\gamma(G, c)$	RND	4 (2.3 %)	33 (18.9 %)	81 (46.3 %)	57 (32.5 %)
	AP1	1 (0.6 %)	69 (39.4 %)	51 (29.1 %)	54 (30.9 %)
	LLG	7 (4.0 %)	62 (35.4 %)	43 (24.6 %)	63 (36.0 %)
	BR	163 (93.1 %)	11 (6.3 %)	0 (0.0 %)	1 (0.6 %)
$U(G, c)$	RND	1 (0.6 %)	42 (24.0 %)	106 (60.6 %)	26 (14.8 %)
	AP1	0 (0.0 %)	13 (7.4 %)	13 (7.4 %)	149 (85.1 %)
	LLG	6 (3.4 %)	114 (65.1 %)	55 (31.5 %)	0 (0.0 %)
	BR	168 (96.0 %)	6 (3.4 %)	1 (0.6 %)	0 (0.0 %)
$\overline{P}(G, c)$	RND	5 (2.9 %)	56 (32.0 %)	96 (54.9 %)	18 (10.2 %)
	AP1	0 (0.0 %)	7 (4.0 %)	13 (7.4 %)	155 (88.6 %)
	LLG	7 (4.0 %)	106 (60.6 %)	60 (34.3 %)	2 (1.1 %)
	BR	163 (93.2 %)	6 (3.4 %)	6 (3.4 %)	0 (0.0 %)
T (G,C)	RND	173 (98.9 %)	2 (1.1 %)	0 (0.0 %)	0 (0.0 %)
	AP1	2 (1.1 %)	152 (86.9 %)	14 (8.0 %)	7 (4.0 %)
	LLG	0 (0.0 %)	20 (11.4 %)	101 (57.7 %)	54 (30.9 %)
	BR	0 (0.0 %)	1 (0.6 %)	60 (34.3 %)	114 (65.1 %)

Our data highlight clearly that algorithm BR is the best performing one, globally, in terms of approximation. In fact, out of 175 combinations of input instances and values of k, BR computes a $\gamma(G, c)$-NE with the smallest value of $\gamma(G, c)$ in 163 of them (93.1% of the combinations). Algorithms RND, LLG and AP1, instead, behave rather badly, providing the best approximation, together, only in the remaining 6.9% of the combinations. Moreover, values of $\gamma(G, c)$ obtained by LLG and AP1 are often quite close to those of RND, as shown in Figures 2–3 (for $k = 3$) and Figure 4 (for larger k), which represents a solid evidence of the practical ineffectiveness of the two. Notice that, in such figures we report detailed measures of $\gamma(G, c)$ and of other indicators introduced in this section, for all algorithms and for a meaningful selection of input graphs and values of k (results for other inputs lead to similar considerations and hence are omitted, due to space constraints). Notice also that, in all mentioned figures, we report a default of $\gamma(G, c) = 0$ when there exist at least one vertex with zero payoff (hence γ is unbounded) while $\gamma(G, c) = 1$ and $U(G, c) = 0$ correspond to a pure NE being found. Besides BR being the best solution with respect to approximation, the most surprising outcome of our experimentation is that BR is able to compute, in almost all cases, colorings that are pure, exact NEs (see e.g. Figure 2 or 3 (middle)). This is remarkable, considering the known hardness of determining this kind of colorings in general digraphs. Specifically, BR is able to find pure Nash equilibria, often in less

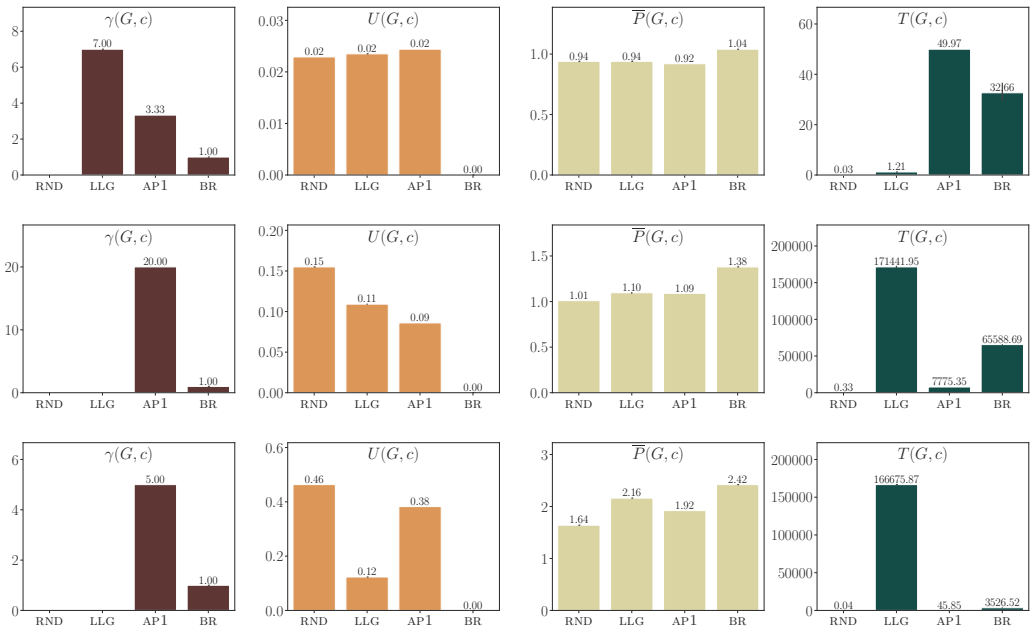


Figure 2 Performance of algorithms RND, LLG, AP1 and BR, respectively, in graphs TWI (top), FAC (middle), and LUX (bottom) with $k = 3$. Time is expressed in seconds.

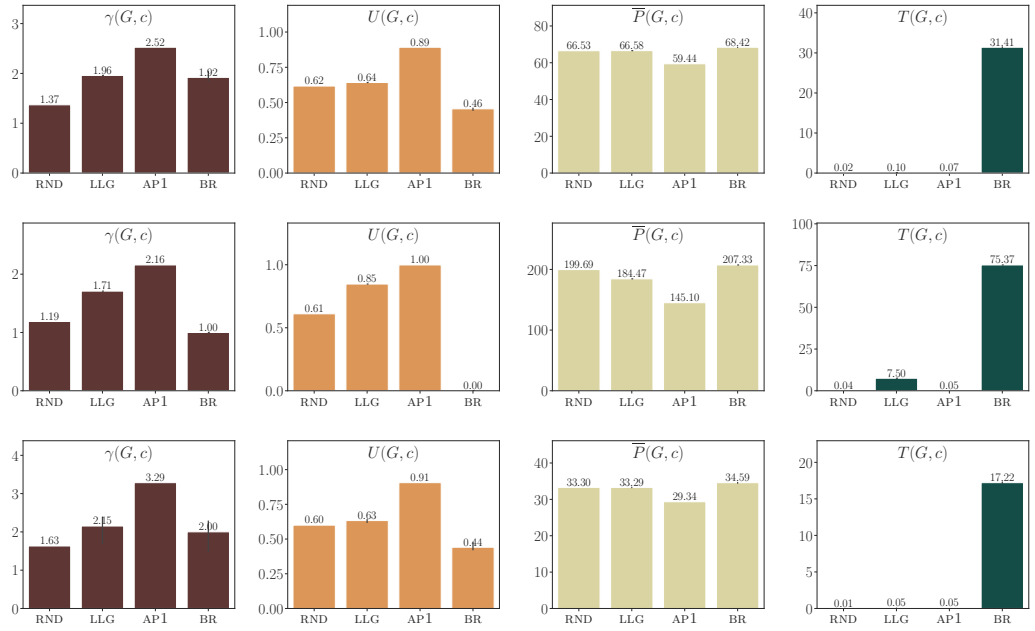


Figure 3 Performance of algorithms RND, LLG, AP1 and BR, respectively, in graphs ERD (top), PL1 (middle), and ERE (bottom), with $k = 3$. Time is expressed in seconds.

than n iterations, for all considered graphs and values of k (see, e.g., Figure 5, two left-most panels) except Erdős-Rényi instances (where however in some case still achieves the best approximation). In these latter inputs, colorings computed by BEST-RESP appears to exhibit a $\gamma(G, c)$ that becomes periodic at some point of the optimization process, around some value close to 1 (see Figure 5, two right-most panels). Note that, when BR ranks 2nd best in terms

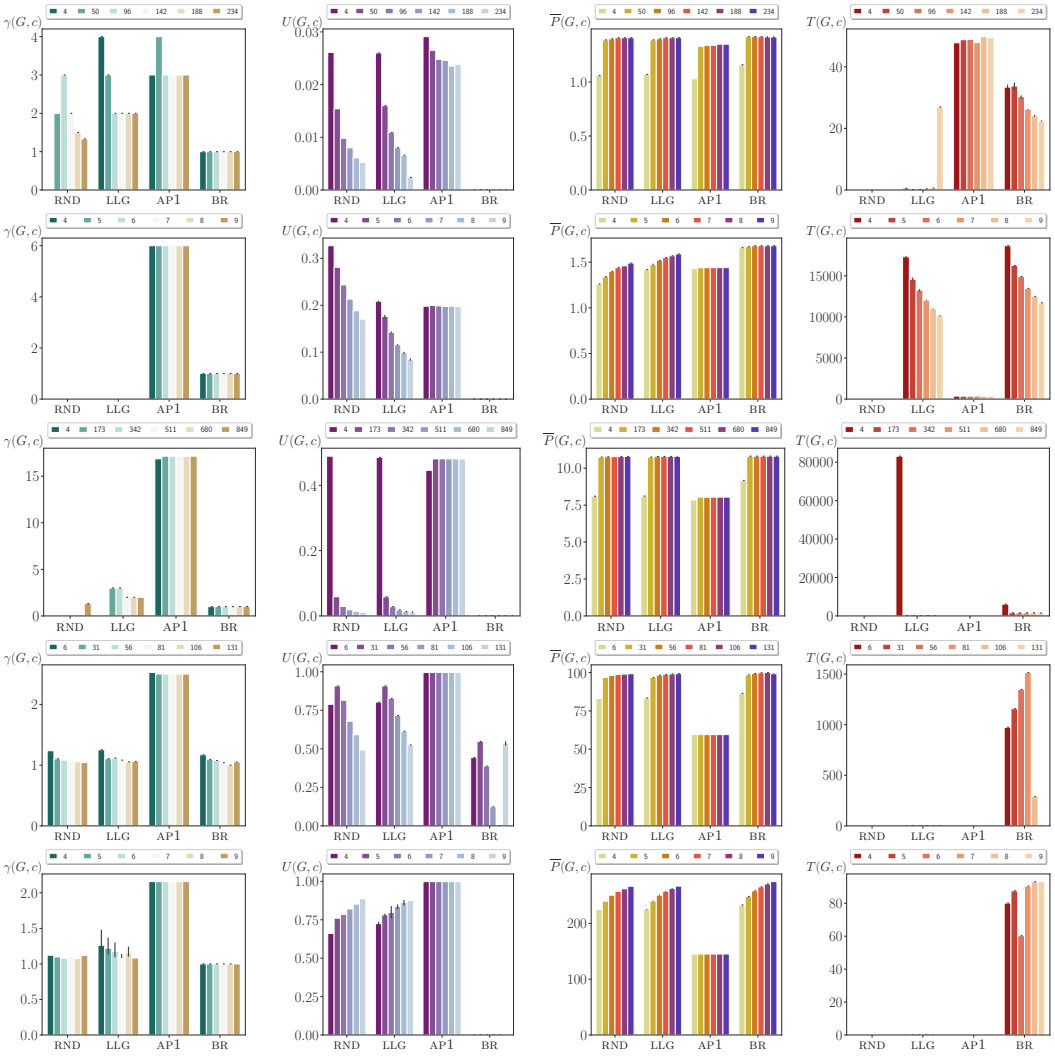


Figure 4 Performance of algorithms RND, LLG, AP1 and BR, respectively, in graphs TWI (top-most), AMA (middle-top), GOO (middle), ERD (middle-bottom) and PL1 (bottom-most), with increasing values of k. Time is expressed in seconds.

of $\gamma(G, c)$, algorithm RND results to be the best performing one (see, e.g., Figure 3, top or bottom). At the same time, BR exhibits higher average payoff and lower fraction of unhappy vertices, which suggests that random assignment might be "lucky" in picking and making happy high-degree vertices, while algorithm BR is able to achieve maximum payoff for a large fraction of the vertex set. By the above, we conjecture that there might exist an analysis for algorithm BR to prove a bounded approximation ratio for a broad class of graphs.

In this direction, it is worth noticing that, unexpectedly, LLG fails at achieving the best approximation even in graphs where the LLL is satisfied (i.e. where LLG finds constant approximation in expected polynomial time). This might be due the fact that the threshold value of γ, for which LLG stops, is rather high, often larger than Δ_o (e.g. 2690.36 for instance ERC, see Eq. 1 with $k = 3$). In this respect, to achieve better results in practice, one might

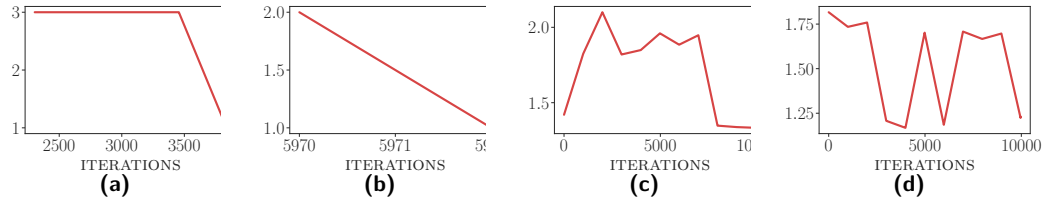


Figure 5 Results achieved by the execution of algorithm BR, in terms of $\gamma(G, c)$, on graphs BLG (a), WVT (b), ERD (c) and ERE (d) with $k = 3$, respectively. The y–axis shows the measured value of $\gamma(G, c)$ as a function of the number of iterations performed by the algorithm, reported on the x-axis (we omit iterations where $\gamma(G, c)$ is unbounded due e.g. to some vertex having zero payoff). In the two left-most panels, i.e. (a)–(b), we can observe $\gamma(G, c)$ the algorithm quickly converging to a pure NE ($\gamma(G, c)$ approaches and then stabilizes at 1 (in much less than $n \log n$ and n iterations, respectively). In the two right-most panels, , i.e. (c)–(d), instead, $n \log n$ iterations do not suffice to achieve convergence at pure NE and $\gamma(G, c)$ seems to start oscillating around a value of 1.75 and 2, respectively.

think of removing such stopping criterion from LLL-GEN to let the coloring being updated, via resampling, for a maximum number of iterations, as done by BEST-RESP. Nonetheless, as further experimentation shows (omitted due to space limitations), this does not lead to meaningful improvements, in terms of both approximation and other indicators. We can hence conclude that repeated operations of resampling of the dependency set are indeed enough to obtain constant approximation w.h.p. but result to be empirically ineffective. Thus, another outcome of our study is that the use of LLG is discouraged for practical purposes, unless more effective ways of exploiting the LLL can be determined. Nonetheless, for the sake of completeness, it is worth noticing that also the latter might not be enough. In fact, our study points out that inputs arising in real-world applications satisfying the LLL are essentially non-existent (see Figure 1).

Regarding the impact of varying k on the performance of the considered algorithms, we observe that, while BR remains the best performing in essentially all cases, approximation ratio and fraction of unhappy vertices (running time and average payoff, respectively) tend to decrease (increase, respectively) with k, for all algorithms. This might suggest that larger values of k could reduce the possibility of being unhappy, by increasing the choices of the agents in the strategy set. Further investigation is hence necessary also here to find out whether there exists some relationship between k, Δ_o, and the quality of the equilibrium that can be achieved. As a final remark, concerning running time, our data mostly confirm what it is expected, i.e. that: (i) RND is trivially the fastest method; (ii) in some cases BR is the most time consuming solution (always achieving the best approximation in these cases); (iii) in the remaining cases, either AP1 and LLG yield the largest $T(G, c)$ but they do not achieve the best approximation. The latter represents another evidence of BR being fast at converging to a pure NE, when it exists. To summarize, our experimental study identifies algorithm BR as the best performing one for digraph k-coloring games. This observation,

combined with the fact that (myopic) best response approaches are easy to implement, even in a distributed uncoordinated environment, suggests that BR is strongly advised to find good Nash equilibria in application domains where the considered game arise.

5 Conclusion and Future Work

Our experimental study adds considerable insights on digraph k-coloring games w.r.t. previous theoretical work. In fact, it provides a strong empirical evidence of the fact that theoretical results probably required very rare graphs, tailored around the worst case behaviour, and that best response heuristics outperforms algorithms with guarantees. This motivates further research effort towards proving the existence of a NE in specific classes of graphs, especially the ones typically arising from social phenomena. Moreover, our results suggest that, in general, a NE with a better approximation factor could be found. In fact, for the very few cases in which a NE is not reached by BR, the returned solution is always a γ-NE with a very low approximation ratio. This renews theoretical interest on this relevant class of games. In this direction, it is worth noticing that, a viable strategy to obtain lower bounds on the approximation factor could be that of exploiting a linear programming formulation for the problem, as done for other combinatorial problems in the past.

References

1 Eugenio Angriman, Alexander van der Grinten, Moritz von Looz, Henning Meyerhenke, Martin Nöllenburg, Maria Predari, and Charilaos Tzovas. Guidelines for experimental algorithmics: A case study in network analysis. *Algorithms*, 12(7):127, 2019. doi:10.3390/a12070127.

2 Elliot Anshelevich and Shreyas Sekar. Approximate equilibrium and incentivizing social coordination. In *Proceedings of the 28th Conference on Artificial Intelligence (AAAI 2014)*, pages 508–514, 2014.

3 Krzysztof R. Apt, Bart de Keijzer, Mona Rahn, Guido Schäfer, and Sunil Simon. Coordination games on graphs. *Int. J. Game Theory*, 46(3):851–877, 2017. doi:10.1007/s00182-016-0560-8.

4 Haris Aziz and Rahul Savani. Hedonic games. In Felix Brandt, Vincent Conitzer, Ulle Endriss, Jérôme Lang, and Ariel D. Procaccia, editors, *Handbook of Computational Social Choice*, pages 356–376. Cambridge University Press, 2016. doi:10.1017/CBO9781107446984.016.

5 Anand Bhalgat, Tanmoy Chakraborty, and Sanjeev Khanna. Approximating pure nash equilibrium in cut, party affiliation, and satisfiability games. In *Proceedings of the 11th ACM Conference on Electronic Commerce (EC 2010)*, pages 73–82, 2010.

6 Vittorio Bilò, Angelo Fanelli, Michele Flammini, Gianpiero Monaco, and Luca Moscardelli. Nash stable outcomes in fractional hedonic games: Existence, efficiency and computation. *J. Artif. Intell. Res.*, 62:315–371, 2018. doi:10.1613/jair.1.11211.

7 Vittorio Bilò, Angelo Fanelli, Michele Flammini, and Luca Moscardelli. Graphical congestion games. *Algorithmica*, 61(2):274–297, 2011.

8 Lawrence E. Blume. The statistical mechanics of best-response strategy revision. *Games and Economic Behavior*, 11(2):111–145, 1995. doi:10.1006/game.1995.1046.

9 Béla Bollobas. *Random Graphs*. Cambridge University Press, 2001.

10 Michele Borassi and Emanuele Natale. Kadabra is an adaptive algorithm for betweenness via random approximation. *ACM J. Exp. Algorithmics*, 24, February 2019. doi:10.1145/3284359.

11 Yang Cai, Ozan Candogan, Constantinos Daskalakis, and Christos H. Papadimitriou. Zero-sum polymatrix games: A generalization of minmax. *MOR*, 41(2):648–655, 2016.

12 Ioannis Caragiannis, Angelo Fanelli, and Nick Gravin. Short sequences of improvement moves lead to approximate equilibria in constraint satisfaction games. In *Proceedings of the 7th International Symposium on Algorithmic Game Theory (SAGT 2014)*, pages 49–60, 2014.

13 Raffaello Carosi, Simone Fioravanti, Luciano Gualà, and Gianpiero Monaco. Coalition resilient outcomes in max k-cut games. In *Proceedings of the 45th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2019)*, volume 11376 of *Lecture Notes in Computer Science*, pages 94–107. Springer, 2019. doi:10.1007/978-3-030-10801-4_9.

14 Raffaello Carosi, Michele Flammini, and Gianpiero Monaco. Computing approximate pure nash equilibria in digraph k-coloring games. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2017)*, pages 911–919. ACM, 2017.

15 Raffaello Carosi and Gianpiero Monaco. Generalized graph k-coloring games. *Theory Comput. Syst.*, 64(6):1028–1041, 2020. doi:10.1007/s00224-019-09961-9.

16 Matthew Cary, Aparna Das, Benjamin Edelman, Ioannis Giotis, Kurtis Heimerl, Anna R. Karlin, Scott Duke Kominers, Claire Mathieu, and Michael Schwarz. Convergence of position auctions under myopic best-response dynamics. *ACM Trans. Econ. Comput.*, 2(3), 2014.

17 Annalisa D'Andrea, Mattia D'Emidio, Daniele Frigioni, Stefano Leucci, and Guido Proietti. Experimental evaluation of dynamic shortest path tree algorithms on homogeneous batches. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA 2014)*, volume 8504 of *Lecture Notes in Computer Science*, pages 283–294. Springer, 2014. doi:10.1007/978-3-319-07959-2_24.

18 Gianlorenzo D'Angelo, Mattia D'Emidio, and Daniele Frigioni. Distance queries in large-scale fully dynamic complex networks. In Veli Mäkinen, Simon J. Puglisi, and Leena Salmela, editors, *Proceedings of the 27th International Workshop on Combinatorial Algorithms (IWOCA 2016)*, volume 9843 of *Lecture Notes in Computer Science*, pages 109–121. Springer, 2016. doi:10.1007/978-3-319-44543-4_9.

19 Gianlorenzo D'Angelo, Mattia D'Emidio, and Daniele Frigioni. Fully dynamic 2-hop cover labeling. *J. Exp. Algorithmics*, 24(1):1.6:1–1.6:36, 2019. doi:10.1145/3299901.

20 Argyrios Deligkas, John Fearnley, and Rahul Savani. Tree polymatrix games are ppad-hard. In *Proceedings of the 47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, volume 168 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.38.

21 Argyrios Deligkas, John Fearnley, Rahul Savani, and Paul Spirakis. Computing approximate nash equilibria in polymatrix games. *Algorithmica*, 77(2):487–514, 2017. doi:10.1007/s00453-015-0078-7.

22 Mattia D'Emidio. Faster algorithms for mining shortest-path distances from massive time-evolving graphs. *Algorithms*, 13(8):191, 2020.

23 B. Curtis Eaves. Polymatrix games with joint constraints. *SIAM Journal on Applied Mathematics*, 24(3):418–423, 1973.

24 Alex Fabrikant, Christos Papadimitriou, and Kunal Talwar. The complexity of pure nash equilibria. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC 2004)*, pages 604–612. ACM, 2004.

25 Michal Feldman and Ophir Friedler. A unified framework for strong price of anarchy in clustering games. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming (ICALP 2015)*, volume 9135 of *Lecture Notes in Computer Science*, pages 601–613. Springer, 2015.

26 Moran Feldman, Liane Lewin-Eytan, and Joseph (Seffi) Naor. Hedonic clustering games. *ACM Trans. Parallel Comput.*, 2(1), 2015. doi:10.1145/2742345.

27 Martin Gairing and Rahul Savani. Computing stable outcomes in hedonic games with voting-based deviations. In *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, pages 559–566, 2011.

28 Laurent Gourvès and Jérôme Monnot. The max k-cut game and its strong equilibria. In *Proceedings of the 7th Annual Conference on Theory and Applications of Models of Computation (TAMC 2010)*, volume 6108 of *Lecture Notes in Computer Science*, pages 234–246. Springer, 2010. doi:10.1007/978-3-642-13562-0_22.

29 Martin Hoefer. *Cost sharing and clustering under distributed competition*. PhD thesis, University of Konstanz, 2007.

30 Joseph T. Howson. Equilibria of polymatrix games. *Management Science*, 18(5):312–318, 1972.

31 Joseph T. Howson and Robert W. Rosenthal. Bayesian equilibria of finite two-person games with incomplete information. *Management Science*, 21(3):313–315, 1974.

32 Michael J. Kearns, Michael L. Littman, and Satinder P. Singh. Graphical models for game theory. In *Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence (UAI 2001)*, pages 253–260, 2001.

33 Jeremy Kun, Brian Powers, and Lev Reyzin. Anti-coordination games and stable graph colorings. In *Proceedings of the 6th International Symposium on Algorithmic Game Theory (SAGT 2013)*, pages 122–133, 2013.

34 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

35 Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012.

36 Douglas A. Miller and Steven W. Zucker. Copositive-plus lemke algorithm solves polymatrix games. *Operations Research Letters*, 10(5):285–290, 1991. `doi:10.1016/0167-6377(91)90015-H`.

37 Gianpiero Monaco, Luca Moscardelli, and Yllka Velaj. On the performance of stable outcomes in modified fractional hedonic games with egalitarian social welfare. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2019)*, pages 873–881. International Foundation for Autonomous Agents and Multiagent Systems, 2019.

38 Gianpiero Monaco, Luca Moscardelli, and Yllka Velaj. Stable outcomes in modified fractional hedonic games. *Auton. Agents Multi Agent Syst.*, 34(1):4, 2020. `doi:10.1007/s10458-019-09431-z`.

39 Dov Monderer and Lloyd S. Shapley. Potential games. *Games and Economic Behavior*, 14(1):124–143, 1996.

40 Robin A. Moser and Gábor Tardos. A constructive proof of the general lovász local lemma. *J. ACM*, 57(2), 2010.

41 Panagiota N. Panagopoulou and Paul G. Spirakis. A game theoretic approach for efficient graph coloring. In *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC 2008)*, pages 183–195, 2008.

42 Dominik Peters and Edith Elkind. Simple causes of complexity in hedonic games. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 617–623, 2015.

43 Svatopluk Poljak. Integer linear programs and local search for max-cut. *SIAM J. Comput.*, 24(4):822–839, 1995.

44 Mona Rahn and Guido Schäfer. Efficient equilibria in polymatrix coordination games. In *Proceedings of 40th International Symposium on Mathematical Foundations of Computer Science (MFCS 2015)*, pages 529–541, 2015.

45 Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the 29th Conference on Artificial Intelligence (AAAI 2015)*, pages 4292–4293. AAAI Press, 2015.

46 Alejandro A. Schäffer and Mihalis Yannakakis. Simple local search problems that are hard to solve. *SIAM J. Comput.*, 20(1):56–87, 1991.

47 Joel Spencer. Asymptotic lower bounds for ramsey functions. *Discrete Mathematics*, 20:69–76, 1977. `doi:10.1016/0012-365X(77)90044-9`.

48 Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016. `doi:10.1017/nws.2016.20`.

49 Brian Swenson, Ryan Murray, and Soummya Kar. On best-response dynamics in potential games. *SIAM Journal on Control and Optimization*, 56(4):2734–2767, 2018.

50 Brian Woodbury Swenson. *Myopic Best-Response Learning in Large-Scale Games*. PhD thesis, Carnegie Mellon University, 2017. doi:10.1184/R1/6720788.v1.

51 Elena B. Yanovskaya. Equilibrium points in polymatrix games. *Lithuanian Mathematical Journal*, 8(2):381–384, 1968.

Practical Performance of Random Projections in Linear Programming

Leo Liberti
LIX CNRS, Ecole Polytechnique, Institut Polytechnique de Paris, 91128 Palaiseau, France

Benedetto Manca
Department of Mathematics and Informatics, University of Cagliari, Italy

Pierre-Louis Poirion
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan

Abstract

The use of random projections in mathematical programming allows standard solution algorithms to solve instances of much larger sizes, at least approximately. Approximation results have been derived in the relevant literature for many specific problems, as well as for several mathematical programming subclasses. Despite the theoretical developments, it is not always clear that random projections are actually useful in solving mathematical programs in practice. In this paper we provide a computational assessment of the application of random projections to linear programming.

2012 ACM Subject Classification Mathematics of computing → Mathematical optimization; Theory of computation → Random projections and metric embeddings

Keywords and phrases Linear Programming, Johnson-Lindenstrauss Lemma, Computational testing

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.21

Supplementary Material *Software (Source Code)*:
`https://mega.nz/file/p8MQhbpT#0TJBUVgaBf4KPVk2fu_5k05cMy2VozJk-0fQ1PZdJ0U`

Funding *Benedetto Manca*: Partly supported by grant STAGE, Fondazione Sardegna 2018.

1 Introduction

This paper is about applying Random Projections (RP) to Linear Programming (LP) formulations. RPs are dimensional reduction operators that usually apply to data. The point of applying RPs to LPs is to obtain an approximate solution of the high-dimensional formulation by solving a related lower-dimensional one. The main goal of this paper is to discuss the pros and cons of this technique from a computational (practical) point of view.

1.1 Random Projections

In general, RPs are functions, sampled randomly from certain distributions, that map a vector in $\mathbb{R}^m$ to one in $\mathbb{R}^k$, where $k \ll m$. In this paper we restrict our attention to linear RPs, which are $k \times m$ random matrices T. The most famous result about RPs is the Johnson-Lindenstrauss Lemma [13], which we recall here in its probabilistic form. Given a finite set $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^m$ and an $\epsilon \in (0, 1)$, there exists a $\delta = O(e^{-\mathcal{C}\phi(k)})$ (with ϕ usually linear and $\mathcal{C}$ a universal constant not depending on input data) and an RP T with $k = O(\epsilon^{-2} \ln n)$ such that

$$\mathbf{Prob}\big(\forall i < j \leq n\ (1-\epsilon)\|x_i - x_j\|_2 \leq \|Tx_i - Tx_j\|_2 \leq (1+\epsilon)\|x_i - x_j\|_2\big) \geq 1 - \delta. \quad (1)$$

If T is sampled componentwise from the normal distribution $\mathsf{N}(0, 1/\sqrt{k})$, Eq. (1) holds (note that other distributions also work). The JLL is not the only result worth mentioning in RP [22, 11, 19], but it is the object of interest in this paper.

The JLL directly applies to all problems involving the Euclidean distance between points in a Euclidean space of high dimension, e.g. the design of an efficient nearest-neighbor data structure (i.e. given $X \subset \mathbb{R}^m$ and $q \in \mathbb{R}^m$ quickly return $x \in X$ closest to q) [12].

More in general, the JLL shows that RPs can transform the point set X to a lower dimensional set TX such that X and TX are "approximately congruent": the pairwise distances in X are approximately the same (multiplicatively) as the corresponding pairwise distances in TX, even if X has m dimensions and TX only k (proportional to $\epsilon^{-2} \ln |X|$). Since "approximately congruence" means "almost the same, aside from translations, rotations, and reflections", it is reasonable to hope that RPs might apply to other constructs than just sets of points, and still deliver a theoretically quantifiable approximation. In this paper we consider LP.

1.2 Applying RPs to Linear Programming

In this paper we are interested in the application of the JLL to LP in standard form:

$$\left.\begin{array}{lrcl} \min\limits_{x} & c^\top x & & \\ & Ax & = & b \\ & x & \geq & 0, \end{array}\right\} \quad \text{(LP)}$$

where $x = (x_1, \ldots, x_n)$, A is an $m \times n$ matrix, and $b \in \mathbb{R}^m$.

There are several issues in applying RPs to Mathematical Programs (MP) in general. The three foremost are:

1. RPs project vectors rather than decision variables and constraint functions;
2. RPs ensure approximate congruence of the input vectors in the lower-dimensional output: but approximation arguments in LP must instead be based on optimality and feasibility (unrelated to the ℓ_2 norm);
3. RPs only apply to finite point sets, whereas LP decision variables represent infinite point sets.

These issues pose nontrivial theoretical challenges, and the proof techniques vary considerably depending on the MP subclass being considered. The first issue mentioned above is addressed by applying RPs to the problem parameters (the input data); in the LP case, we project the linear system $Ax = b$. We speak of the *original* formulation P and the *projected* formulation TP. This yields a fourth issue: the solution of TP may be infeasible in P: in such cases, a *solution retrieval* phase is necessary in order to construct a feasible solution of P from that of TP.

The second and third issues are addressed in [25], leading to statements similar to the JLL, but concerning approximate LP feasibility and optimality. If $E(P, T)$ is a statement about the feasibility or optimality error between the LP formulations P and TP, the general structure of these results is similar to the probabilistic version of the JLL:

$$\mathbf{Prob}\big(E(P,T)\big) \geq 1 - \delta, \tag{2}$$

where δ usually depends on ϵ, k and possibly even the solution of P. We shall recall the statements of these results more precisely in Sect. 2.

1.3 Relevant literature

The main reference for RPs and LP in standard form is [25], which presents the theory addressing the above issues, and a computational study focussing on dense random LP instances. RPs were also applied to some specific LP problems: PAC learning [20] and quantile regression [26], with dimensional reduction techniques tailored to the corresponding

LP structure. Other works in applying RPs to different types of MP subclasses are [24] (quadratic programs with a ball constraint), [4] (general quadratic programs), [18] (conic programs including second-order cone and semidefinite programs).

1.4 Contributions of this paper

Although some of the relevant literature carries computational results, we think that, computationally, the application of RPs to LPs is still experimental: in practice the output on a given instance can range from accurate all the way to catastrophic.

One of the difficulties is that, in writing $k = O(\epsilon^{-2} \ln n)$, we are neglecting a constant multiplicative coefficient C related to the "big oh", the appropriate value of which is usually the fruit of guesswork. Another difficulty is that the the theoretical results in this area apply to "high dimensions", without specifying a minimum dimension above which they hold. In catastrophic cases, the theory ensures that results would improve for larger instance sizes, but just how large is unknown. At this time, in our opinion, no-one is able to justifiably foresee whether RPs will be useful or not on a given LP instance. The only existing work about practical RP usage is [23], which only focusses on computational testing of different RP matrices.

This paper will provide a computational analysis of LP cases where RPs work reasonably well, and others where they do not, and attempt to derive some guidelines for choosing appropriate values for the most critical unknown parameters. On the theoretical side, we tighten two results of [18] when applied to the LP case.

The rest of this paper is organized as follows. In Sect. 2 we recall the main theoretical results relative to the application of RPs to LP, and state the two new tightened results. In Sect. 3 we illustrate the benchmark goal, the LP structures we test, and the methodology. In Sect. 4 we discuss the benchmark results.

2 Summary of theoretical results

We apply RPs to the original formulation (LP) by reducing the number m of constraints. Let T be a $k \times m$ RP matrix. The projected formulation is:

$$\min\{c^\top x \mid TAx = Tb \wedge x \geq 0\} \quad (T\mathsf{LP}).$$

We first discuss feasibility. We note that the geometric interpretation of the feasible set $F = \{x \mid Ax = b \wedge x \geq 0\}$ of (LP) is that F is the set of conic combinations of the columns A^j of A, i.e. $F = \mathsf{cone}(A)$. We also let $\mathsf{conv}(A)$ the convex hull of the columns of A, and $\|x\|_A = \min\{\sum_j \lambda_j \mid x = \sum_j \lambda_j A^j\}$ be the A-*norm* of $x \in \mathsf{cone}(A)$. Is F is invariant w.r.t. the application of T to (LP)? If $x \in F$ then $TAx = Tb$ by linearity of T. On the other hand, it is generally false that if $x \geq 0$ but $x \notin F$, then $TAx \neq Tb$. The following approximate feasibility statement

$$b \notin \mathsf{cone}(A) \Rightarrow \mathbf{Prob}\big(Tb \notin \mathsf{cone}TA\big) \geq 1 - 2(n+1)(n+2)e^{-\mathcal{C}(\epsilon^2-\epsilon^3)k} \tag{3}$$

is proved in [25, Thm. 3] for all $\epsilon \in (0, \Delta^2/(\mu_A + 2\mu_A\sqrt{1-\Delta^2}+1))$, where $\mathcal{C}$ is the universal constant of the JLL, $\mu_A = \max\{\|x\|_A \mid x \in \mathsf{cone}(A) \wedge \|x\|_2 \leq 1\}$, and Δ is a lower bound to $\min_{x \in \mathsf{conv}(A)} \|b - x\|_2$.

Let $\mathsf{val}(\cdot)$ indicate the optimal objective function value of a MP formulation. The approximate optimality statement for (LP) derived in [25, Thm. 4] is conditional to the LP formulation being feasible and bounded, so that, if x^* is an optimal solution, there is θ (assumed w.l.o.g. ≥ 1) such that $\sum_j x^*_j < \theta$. Given $\gamma \in (0, \mathsf{val}(\mathsf{LP}))$,

$$\mathbf{Prob}\big(\mathsf{val}(\mathsf{LP}) - \gamma \leq \mathsf{val}(T\mathsf{LP}) \leq \mathsf{val}(\mathsf{LP})\big) \geq 1 - \delta, \tag{4}$$

where $\delta = 4ne^{-\mathcal{C}(\epsilon^2-\epsilon^3)k}$, $\epsilon = O(\gamma/(\theta^2\|y^*\|_2))$, and y^* is an optimal dual solution of (LP). Like other approximate optimality results in this field, some quantities in the probabilistic statement depend on the norm of a dual optimal solution. This adds a further difficulty to computational evaluations, since they cannot be computed prior to solving the problem.

Let $\bar{x}$ be a *projected solution*, i.e. an optimal solution of the projected formulation. In [25, Prop. 3], it is proved that $\bar{x}$ is feasible in the original formulation with zero probability. We therefore need to provide a solution retrieval method. A couple were proposed in [25], but the one found in [18, Eq. (6)] comes with an approximation guarantee and a good practical performance. The retrieved solution $\tilde{x}$ is defined as the projection of $\bar{x}$ on the affine subspace $Ax = b$, and computed using the pseudoinverse:

$$\tilde{x} = \bar{x} + A^\top(AA^\top)^{-1}(b - A\bar{x}). \tag{5}$$

The fact that we only project on $Ax = b$ without enforcing $x \geq 0$ is necessary, since otherwise we would need to solve the whole high-dimensional LP. On the other hand, it causes potential infeasibility errors w.r.t. $x \geq 0$. A probabilistic bound on this error is cast in general terms for conic programs in [25]. Let $\kappa(A)$ be the condition number of A; applying [25, Thm. 4.4] to LP, we obtain the following result, which bound the (negativity of) the smallest component of $\tilde{x}$ in terms of that of $\bar{x}$.

▶ **Proposition 1.** *There is a universal constant $\mathcal{C}_2$ such that, for any $u \geq 0$, we have:*

$$\mathbf{Prob}\big(\min_{j\leq n}\tilde{x}_j \geq \min_{j\leq n}\bar{x}_j - \epsilon\theta\kappa(A)(\mathcal{C}_2 + u\sqrt{2/\ln(n)})\big) \geq 1 - 2e^{-u^2}.$$

The proof is based on an improvement of [18, Eq. (7)] based on computing the Gaussian width and diameter of $\{x \geq 0 \mid \langle \mathbf{1}, x\rangle \leq 1\}$. As a corollary, we also have the following result about the difference between objective function values of the retrieved and projected solutions.

▶ **Corollary 2.** *Let $\tilde{f}$ be the objective function value of the retrieved solution $\tilde{x}$, and $\bar{f}$ be the optimal objective function value of the projected formulation. There is a universal constant $\mathcal{C}_2$ such that, for any $u \geq 0$, we have:*

$$\mathbf{Prob}\big(|\tilde{f} - \bar{f}| \leq \epsilon\theta\kappa(A)\|c\|_2(\mathcal{C}_2 + u\sqrt{2/\ln(n)})\big) \geq 1 - 2e^{-u^2}.$$

3 What we establish and how

Upon receving an LP instance to be solved using RPs, one has to at least know how to decide k (the projected dimension) so that the solution of the projected formulation is reasonably close to that of the original one.

Ideally, one would like to estimate all unknown parameters: $k, \epsilon, C, \mathcal{C}$ in function of γ and δ. This is theoretically hopeless because the theoretical bounds derived for "all LPs" are far from tight. We shall see below that it is also computationally hopeless. In practice, moreover, one might be much more interested in finding a good retrieved solution (i.e. almost feasible in the original problem), rather than finding a good approximation to the optimal objective function value, since a feasible solution can be improved by local methods, while an approximate optimal value may at best be useful as an objective cut.

Our approach will accordingly be based on solving sets of uniformly sampled LP instances (from different applications) using a standard solver, and analyse the output in terms of how the feasibility and optimality errors of the retrieved solution vary with problem size and ϵ.

3.1 The RP matrix

All componentwise sampled sub-Gaussian distributions [7] can be used to ensure the results cited in this paper. Some sparse variants also exist, along the lines of [1, 15]. We use the sparse RPs described in [4, §5.1]. For a given density $\sigma \in (0,1)$ and standard deviation $\sqrt{1/(k\sigma)}$, with probability σ we sample a component of the $k \times m$ RP T from the distribution $\mathsf{N}(0, \sqrt{1/(k\sigma)})$, and set it to zero with probability $1-\sigma$. In our computational study, we set $\sigma = d_A/2$, where d_A is the density of the constraint matrix A.

3.2 LP structures

We consider randomly generated LPs of the following four classes: Max Flow problems [8], Diet problems [6], Quantile Regression problems [16], and Basis Pursuit problems from sparse coding [3]. This choice yields a set of LP problems going from extremely sparse (Max Flow) to completely dense (Basis Pursuit), with the Diet and Quantile Regression providing cases of various intermediate densities. These four test cases arise from a diverse range of application settings: combinatorial optimization, continuous optimization, statistics, data science.

3.2.1 Maximum flow

The Max Flow formulation is defined on a weighted digraph $G = (N, \mathcal{A}, u)$ with a source node $s \in N$, a target node $t \in N$ (with $s \neq t$) and $u : \mathcal{A} \to \mathbb{R}_+$, as follows:

$$\left.\begin{array}{rrcl} \max\limits_{x \in \mathbb{R}_+^{|\mathcal{A}|}} & \sum\limits_{\substack{i \in N \smallsetminus \{s\} \\ (s,i) \in \mathcal{A}}} x_{si} & - & \sum\limits_{\substack{i \in N \smallsetminus \{s\} \\ (i,s) \in \mathcal{A}}} x_{is} \\ \forall i \in N \smallsetminus \{s,t\} & \sum\limits_{\substack{j \in N \\ (i,j) \in \mathcal{A}}} x_{ij} & = & \sum\limits_{\substack{j \in N \\ (j,i) \in \mathcal{A}}} x_{ji} \\ \forall (i,j) \in \mathcal{A} & 0 \leq x_{ij} & \leq & u_{ij}. \end{array}\right\} \quad (\mathsf{MF})$$

We generate random weighted digraphs $G = (N, \mathcal{A}, u)$ with the property that a single (randomly chosen) node s is connected (through paths) to all of the other nodes: we first generate a random tree on $N \smallsetminus \{t\}$, orient it so that s is the root, add a node t with the same indegree as the outdegree of s, and then proceed to enrich this digraph with arcs generated at random using the Erdős-Renyi model with probability 0.05. We then generate the capacities u uniformly from $[0,1]$. Finally, we compute the digraph's incidence matrix A, which has $m = |N| - 2$ rows and $|\mathcal{A}|$ columns. Instances are feasible because the graph always has a path from s to t by construction, and the zero flow is always feasible.

Although (MF) is an LP, it is not in standard form, because of the upper bounding constraints $x \leq u$. But, by [25, §4.2], we can devise a block-structured RP matrix that only projects the equations $Ax = b$, leaving the inequalities $x \leq u$ alone. In this case, A is a flow matrix with two nonzeros per column, one set to 1 the other to -1, aside from columns referring to source and target nodes s, t that only have one nonzero; and $b = 0$. The density of A is $d_A = \frac{2|\mathcal{A}|-2}{(m-2)|\mathcal{A}|} \approx 2/m$.

For our random (MF) instances, $\theta = |\mathcal{A}|$ is a valid upper bound to $\sum_{(i,j)\in\mathcal{A}} x^*_{ij}$, since $0 \leq x_{ij} \leq u_{ij} \leq 1$ for all $(i,j) \in \mathcal{A}$.

3.2.2 Diet problem

The DIET formulation is defined on an $m \times n$ nutrient-food matrix D, a food cost vector $c \in \mathbb{R}^n_+$, and a nutrient requirement vector $b \in \mathbb{R}^m$, as follows:

$$\left.\begin{array}{lll} \min\limits_{q \in \mathbb{R}^{mn}_+} & c^\top q & \\ & Dq & \geq \quad b. \end{array}\right\} \quad (\mathsf{DP})$$

We sample c, D, b uniformly componentwise in $[0,1]$, and set the density of D to $d_D = 0.5$. Instances are feasible because one can always buy enough food to satisfy all nutrient requirements. If $\|D_i\|_0 = |\text{nonzeros of row } D_i|$, then $\hat{q} = \left(\max\limits_{i \leq m}(b_i/(\|D_i\|_0 D_{ij})) \mid j \leq n\right)$ is a feasible solution.

Again, (DP) is not in standard form, but the transformation is immediate using slack variables $r_i \geq 0$ for $i \leq m$. We let $A = (D \mid -I)$, where I is $m \times m$. The decision variable vector is $x = (q, r)$. The density of A is $d_A = (d_D mn + m)/(m(n+m)) = (d_D n + 1)/(n+m)$.

For (DP), the upper bounding solution $\hat{q}$ yields slack values $\hat{r}_i = D_i \hat{q} - b_i$ for all $i \leq m$, where D_i is the i-th row of D. So we let $\theta = \sum_j \hat{q}_j + \sum_i \hat{r}_i$ be an upper bound for $\sum_j x^*_j$.

3.2.3 Quantile regression

The QUANTILE REGRESSION formulation, for a quantile $\tau \in (0,1)$, is defined over a database table D having density d_D with m records and p fields, and a further column field b. We make a statistical hypothesis $b = \sum_j \beta_j D^j$, and aim at estimating $\beta = (\beta_j \mid j \leq p)$ from the data b, D so that errors from the τ-quantile are minimized. Instances may only have nonzero optimal value if $m > p$, as is clear from the constraints of the formulation below:

$$\left.\begin{array}{lll} \min\limits_{\substack{\beta \in \mathbb{R}^p \\ u^+, u^- \in \mathbb{R}^m_+}} & \tau \mathbf{1}^\top u^+ + (1-\tau)\mathbf{1}^\top u^- & \\ & D\beta + Iu^+ - Iu^- \quad = \quad b, \end{array}\right\} \quad (\mathsf{QR})$$

where the constraint system $Ax = b$ has $A = (D|I|-I)$, $x = (\beta, u^+, u^-)$, and τ (the quantile level) is given, and fixed at 0.2 in our experiments. The data matrix (D, b) is sampled uniformly componentwise from $[-1, 1]$, with $d_D = 0.8$. Instances are all feasible because the problem reduces to solving the overconstrained linear system $D\beta = b$ with a "skewed" version of an ℓ_1 error function.

We note that (QR) is not in standard form, since the components of β are unconstrained; but this is not an issue, insofar as the problem is bounded (since it is feasible and it minimizes a weighted sum of non-negative variables), and this is enough to have the results in [25] hold. On the contrary, the lack of non-negative bounds on β is an advantage, since we need not worry about negativity errors in the β components of the retrieved solution (Prop. 1). The density of A is $d_A = (d_D mp + 2m)/(mp + 2m^2) = (d_D p + 2)/(p + 2m)$.

For (QR), given that all data is sampled uniformly from $[-1, 1]$, no optimum can ever have $|\beta_j| > 1$. As for u^+, u^-, we note that any feasible β yields an upper bound to the optimal objective function value, which only depends on u^+, u^-: we can therefore choose $\beta = 0$, and obtain $u^+_i - u^-_i = b_i$ for all $i \leq m$; we then let $u^+_i = b_i \wedge u^-_i = 0$ if $b_i > 0$, and $u^+_i = 0 \wedge u^-_i = -b_i$ otherwise. This yields an upper bound estimate $\theta = p + \sum_i |b_i|$ to $\sum_j x^*_j$.

3.2.4 Basis pursuit

The Basis Pursuit formulation aims at finding the sparsest vector x satisfying the underdetermined linear system $Ax = b$ by resorting to a well-known approximation of the zero-norm by the ℓ_1 norm [3]:

$$\left.\begin{array}{lll} \min\limits_{x,s\in\mathbb{R}^n} & \mathbf{1}^\top s & \\ & Ax = & b \\ \forall j \le n & -s_j \le x_j & \le s_j. \end{array}\right\} \quad \text{(BP)}$$

According to sparse coding theory [5], we work with a fully dense $m \times n$ matrix A sampled componentwise from $\mathsf{N}(0,1)$ (with density $d_A = 1$), a random message obtained as z/Z from a sparse $z \in (\mathbb{Z} \cap [-Z, Z])^n$ (with density 0.2) and $Z = 10$, and compute the encoded message $b = Az$. We then solve (BP) in order to recover the sparsest solution of the underconstrained system $Ax = b$, which should provide an approximation of z. Basis pursuit problems undergo a phase transition as m decreases from n down to zero [2], so it shouldn't really make sense to decrease m by using RPs, and yet some mileage can unexpectedly be extracted from this operation [17].

Similarly to (MF), in (BP) we can partition the constraints into equations $Ax = b$ and inequalities $-s \le x \le s$. Again by [25, §4.2], we devise a block-structured RP matrix which only projects the equations.

As in Sect. 3.2.3, (BP) is not in standard form, since none of the variables are non-negative. In this case, moreover, it is not easy to establish a bound θ on $\sum_j (x^*_j + s^*_j)$, since A is sampled from a normal distribution. On the other hand, for $A_{ij} \sim \mathsf{N}(0,1)$ we have $\mathbf{Prob}\big(A_{ij} \in [-3,3]\big) = 0.997$. By construction, we have $b \in [-3n, 3n]^m$, which implies a defining interval $[-n, n]$ on the components of optimal solutions, yielding $\theta = 2n^2$ with probability 0.997.

3.3 Methodology

The goal of this paper is to provide a computational assessment of RPs applied to LP.

As discussed at the beginning of Sect. 3, the actual determination of all relevant parameters is theoretically hopeless. We can certainly simplify the task a little by noting that the coefficient C can be removed since it suffices to decide a value for ϵ in order to decide k. Ideally we would like to decide γ first (see Eq. (4)), then compute ϵ as $O(\gamma/(\theta^2\|y^*\|_2))$, and sample an appropriate RP. Unfortunately, estimating θ and $\|y^*\|_2$ prior to solving the original LP leads to tiny values for ϵ (e.g. 10^{-i} for $i \in \{2, \ldots, 11\}$ in some preliminary tests), which would require the rows of A to be at least $O(10^{i^2})$ in order to yield a useful projection. Since we are interested in applying RPs to LPs with $O(10^2)$ and $O(10^3)$ rows, this "ideal" approach is inapplicable.

Instead, we repeatedly solve sets of instances of each LP structure. Each projected instance is solved with different values of $\epsilon \in \mathcal{E} = \{0.15, 0.2, 0.25, 0.3, 0.35, 0.4\}$ (these values have been found to be the most relevant in preliminary computational experiments performed over several years). Moreover, to mitigate the effect of randomness, we solve each instance with each ϵ multiple times. For each instance and ϵ we collect performance measures on objective function values, infeasibility errors, and CPU time. This allows us to illustrate the co-variability of ϵ and instance size with the performance measures.

4 The benchmark

The solution pipeline is based on Python 3 [21] and the libraries `scipy` [14] and `amplpy` [9] (besides other standard python libraries). For each problem type, we loop over instances (based on row size of the equality constraint system, varying in $\mathcal{S}$, see below), over $\epsilon \in \mathcal{E}$, and over 5 different runs for each instance and ϵ in order to amortize the result randomness depending on the choice of T. We solve all of the original and projected instances using CPLEX 20.1 [10]. We use the barrier solver, because we found this to be more efficient with large dense LPs than the simplex-based solvers in CPLEX. Our code can be downloaded here.[1] All tests have been carried out on a MacBook 2017 wih a 1.4GHz dual-core Intel Core i7 with 16GB RAM.

4.1 Choice of instances

In the case of Diet, Quantile Regression, and Basis Pursuit, we generated instances so that the number of rows of the equality constraint system $Ax = b$ is in the set $\mathcal{S} = \{500p \mid 1 \leq p \leq 5 \wedge p \in \mathbb{N}\}$. For Max Flow we used $\mathcal{S}' = \mathcal{S} \setminus \{2500\}$ because the larger size triggered a RAM-related error in a part of the solution pipeline involving the AMPL [9] interpreter.

4.1.1 The variable space

The space of original, projected, and retrieved variable values is identical for Max Flow, Quantile Regression, and Basis Pursuit, since these three structures are originally cast in an equality constraint form $Ax = b$. This desirable property fails to hold for Diet, which deserves a separate discussion.

The original formulation (DP) of Diet is in inequality form $Dq \geq b$, but the projected formulation is derived from the constraints $Ax = b$ in standard form, where $A = (D| -I)$.

The theoretical results in Sect. 2 justify a fair comparison only between original and projected solutions in standard form. Since this paper is about a *practical* comparison, however, and since no-one would convert (DP) to standard form before solving it (because the solver would do it as needed), we chose to compute objective function values and feasibility errors of the projected formulation on the space of the original formulation variables q. Thus, for a retrieved solution $\tilde{x} = (\tilde{q}, \tilde{r})$ we only considered $\tilde{q}$ in order to compute the objective function value of $\tilde{x}$.

Considering only the q variables is unproblematic if applied to the optimal solution x^* of the original formulation in standard form, because $s^* \geq 0$ and $A = (D| -I)$ ensure that q^* is a feasible solution in $Dq \geq b$. When applied to the projected formulation, however, $TA = (TD| -TI)$ yields a block matrix TI with both positive and negative entries (since T is sampled from a normal distribution). Thus, it often happens that the underdetermined $k \times m$ system $TI = Tb$ has solutions. In this case, since the objective tends to minimize $c^\top q$, the projected solution $\bar{x} = (\bar{q}, \bar{s})$ will have $\bar{q} = 0$, yielding zero projected objective function value. This, in turn, may yield $D\tilde{q} \not\geq b$. The application of RPs to Diet is therefore less successful than for other structures.

[1] The URL is `https://mega.nz/file/p8MQhbpT#0TJBUVgaBf4KPVk2fu_5k05cMy2VozJk-0fQ1PZdJ0U`.

4.2 Performance measures

At the end of each solver call we record: the optimal objective function f^* of the original problem, the optimal objective function $\bar{f}$ of the projected problem, the objective function value $\tilde{f}$ of the retrieved solution $\tilde{x}$, the feasibility error w.r.t. equation constraints $Ax = b$ (`eq`) and inequalities $x \geq 0$ (`in`), the CPU time t^* taken to solve the original formulation, and the CPU time $\bar{t}$ taken to solve the projected formulation.

The CPU time t^* takes into account: reading the instance, constructing the original formulation, and solving it. The CPU time $\bar{t}$ takes into account: reading the instance, sampling the RP, projecting the instance data, constructing the projected formulation, solving it, and performing solution retrieval.

The benchmark considers: the average objective function ratios $\bar{f}/f^*$, $\tilde{f}/f^*$, the average errors `avgeq`, `avgin` for $Ax = b$ and $x \geq 0$, the ratio k/m, the average CPU ratio $\bar{t}/t^*$: all averages are computed over 5 solution runs over a given instance size and ϵ value.

4.3 RP performance on Max Flow

The application of RPs to the Max Flow problem looks like a success story: the ratio of projected to original optimal objective function value is very close to 1.0 and constant w.r.t. ϵ ($\bar{f}/f^* \geq 1$ is normal insofar as Max Flow is a maximization problem, and TLP is a relaxation of LP). The feasibility error of the retrieved solution related to the equality constraints $Ax = b$ is very close to zero, and the error w.r.t. $x \geq 0$ decreases as m increases (a healthy behaviour in RPs) and also as ϵ increases (implying that maximum negativity error increases more slowly than the number of variables). The CPU time ratio decreases proportionally to k/m, as expected. The only issue is that the objective function value at the retrieved solution is only around 0.5 of the optimum.

4.4 RP performance on Diet

As mentioned in Sect. 4.1.1, the practical application of RPs to the Diet problem is not successful, as shown by the plots in Fig. 2. The projected cost is almost always zero, because the constraint projection allowed the solver to satisfy $(D|-I)(q,r)^\top = b$ using slack variables only. This causes sizable errors in the retrieved solutions. As expected, the CPU time taken to solve the projected formulation is a tiny fraction of the time to solve the original formulation.

We tried to experiment with a modified projected objective $(c\,|\,\mathbf{1})$ so that we would minimize the sum of the projected slack variables. This yielded quantitatively better results, as shown in Fig. 3; qualitatively, the results still look like a failure.

4.5 RP performance on Quantile Regression

The results quality on Quantile Regression is mixed. The ratio $\bar{f}/f^*$ is rather low, but we note that it is higher (better) for low sizes and low ϵ values, which is a sign that ϵ should be further decreased for all (and specially large) sizes. Interestingly, the objective value of the retrieved solution $\tilde{x}$ has better quality. The feasibility errors of $\tilde{x}$ are zero for $Ax = b$, and not negligible (around 0.2, with one outlier) for $x \geq 0$: the trend, unfortunately, is not decreasing, either with ϵ or m increasing. CPU time ratios are good.

To see whether increasing sizes and decreasing ϵ improved performances, we solved an instance with $m = 5000$ and $p = 100$ with $\epsilon = 0.1$, obtaining the following results.

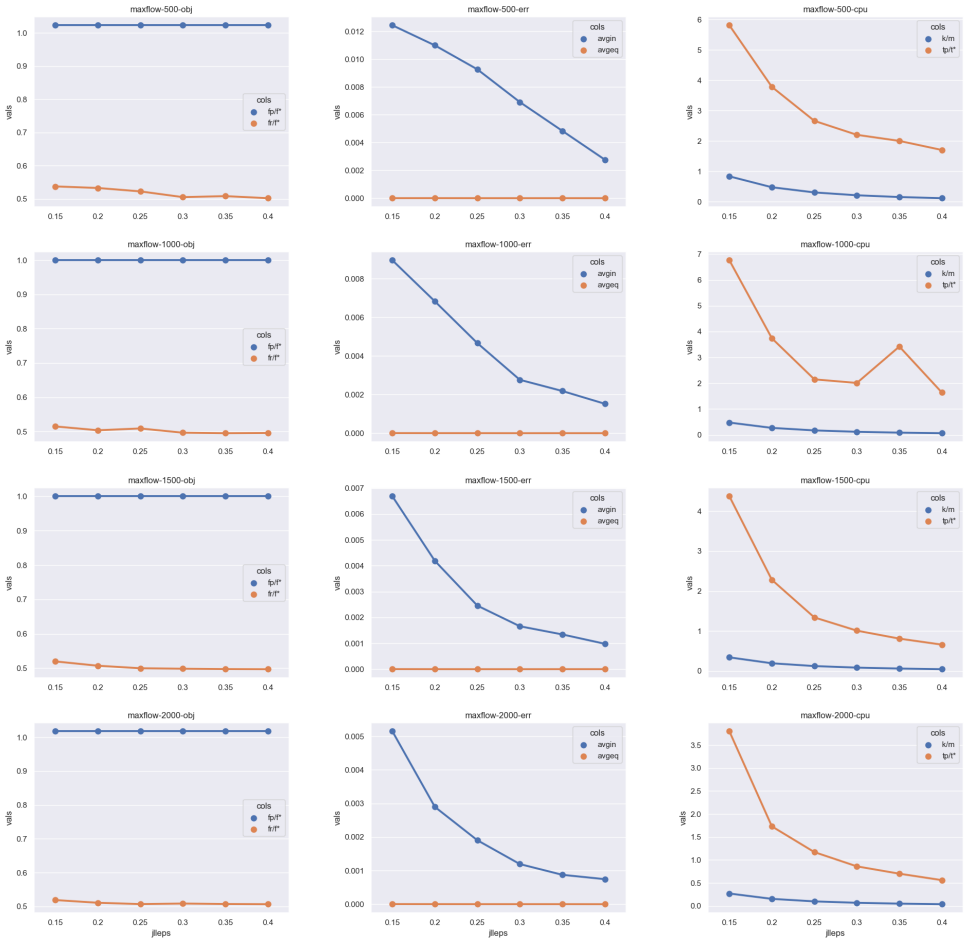


Figure 1 Max Flow plots (increasing ϵ on abscissae): instances of growing size on rows, objective function ratios on the first column, feasibility errors on the second, k/m and CPU time ratio on the third.

ϵ	$\bar{f}/f^*$	$\tilde{f}/f^*$	`avgin`	`avgeq`	k/m	$\bar{t}/t^*$
quantreg-5000						
0.10	0.1460	0.3839	0.1784	0.0000	0.18	4.43

We can see that the objective function ratios of this instance provide a definite improvement with respect to the three largest instances in Fig. 4 ($m \in \{1500, 2000, 2500\}$). The negativity error is, however, of the same magnitude as before.

4.6 RP performance on Basis Pursuit

In the Basis Pursuit problem we see an encouraging trend of the ratio $\bar{f}/f^*$, which starts off at 0.8 for $m = 500$ and $\epsilon = 0.15$, and indicates that ϵ should be decreased for larger sizes. The retrieved solution was not computed on the "sandwich" variables s (see Eq. (BP)), but as the ℓ_1 norm of $\tilde{x}$. Since there are fewer constraints in the encoding matrix A, it follows from compressed sensing theory that the sparsest solution is found less often, a fact that increases the objective value of the retrieved solution. The feasibility errors are always zero

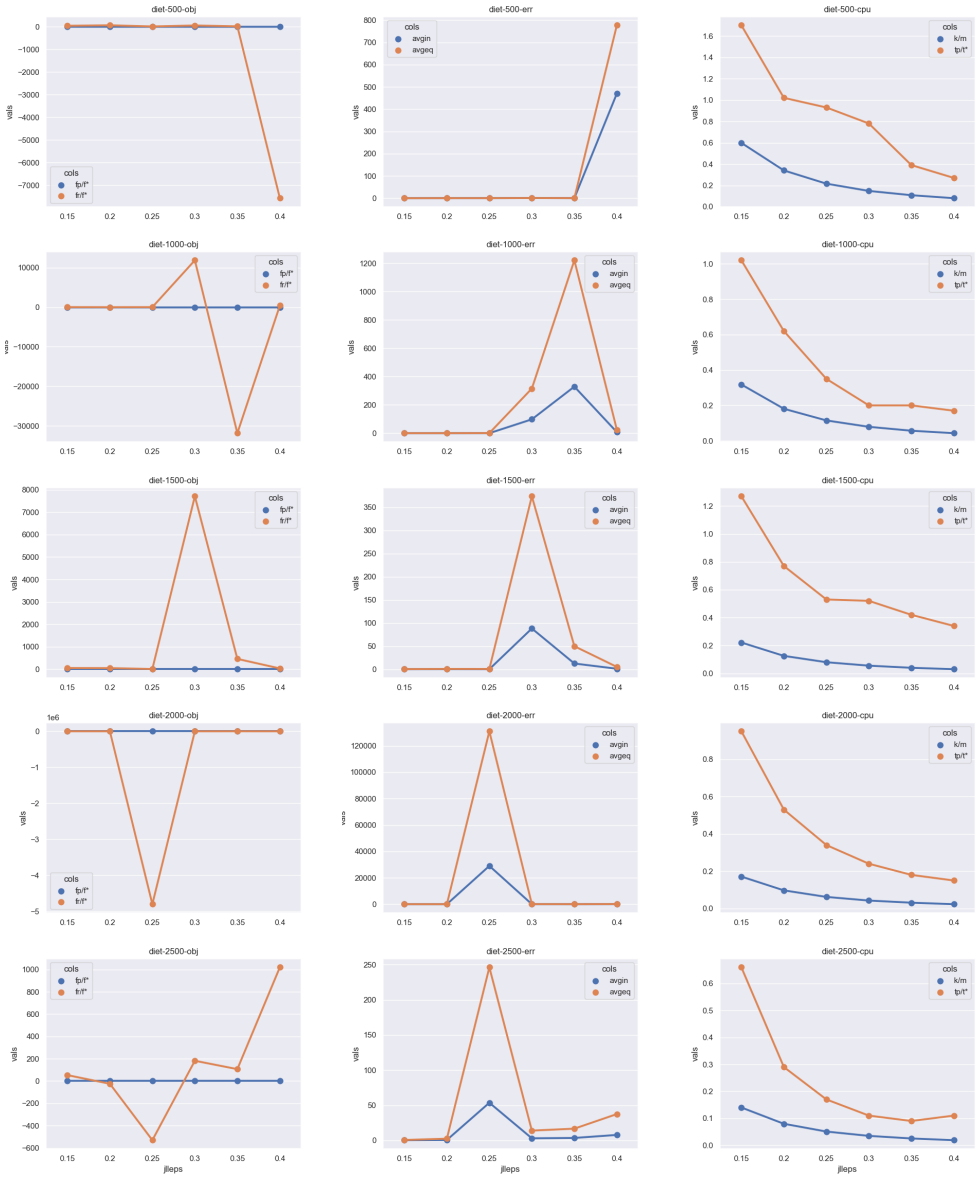


Figure 2 DIET plots (increasing ϵ on abscissae): instances of growing size on rows, objective function ratios on the first column, feasibility errors on the second, k/m and CPU time ratio on the third.

(for $Ax = b$ and $x \geq 0$), which happens because the variables x are unbounded. The CPU time ratio is not as regular as for the other structures, but still denotes a remarkable time saving when solving projected formulations.

To see whether increasing sizes and decreasing ϵ improved performances, we solved an instance with $m = 5000$ and $n = 6000$ with $\epsilon = 0.1$, obtaining the following results.

ϵ	$\bar{f}/f^*$	$\tilde{f}/f^*$	`avgin`	`avgeq`	k/m	$\bar{t}/t^*$
basispursuit-5000						
0.10	0.4925	1.5395	0.0000	0.0000	0.17	0.09

An improvement with respect to the three largest instances in Fig. 5 ($m \in \{1500, 2000, 2500\}$) is present, which points to the correct trend, albeit not substantial.

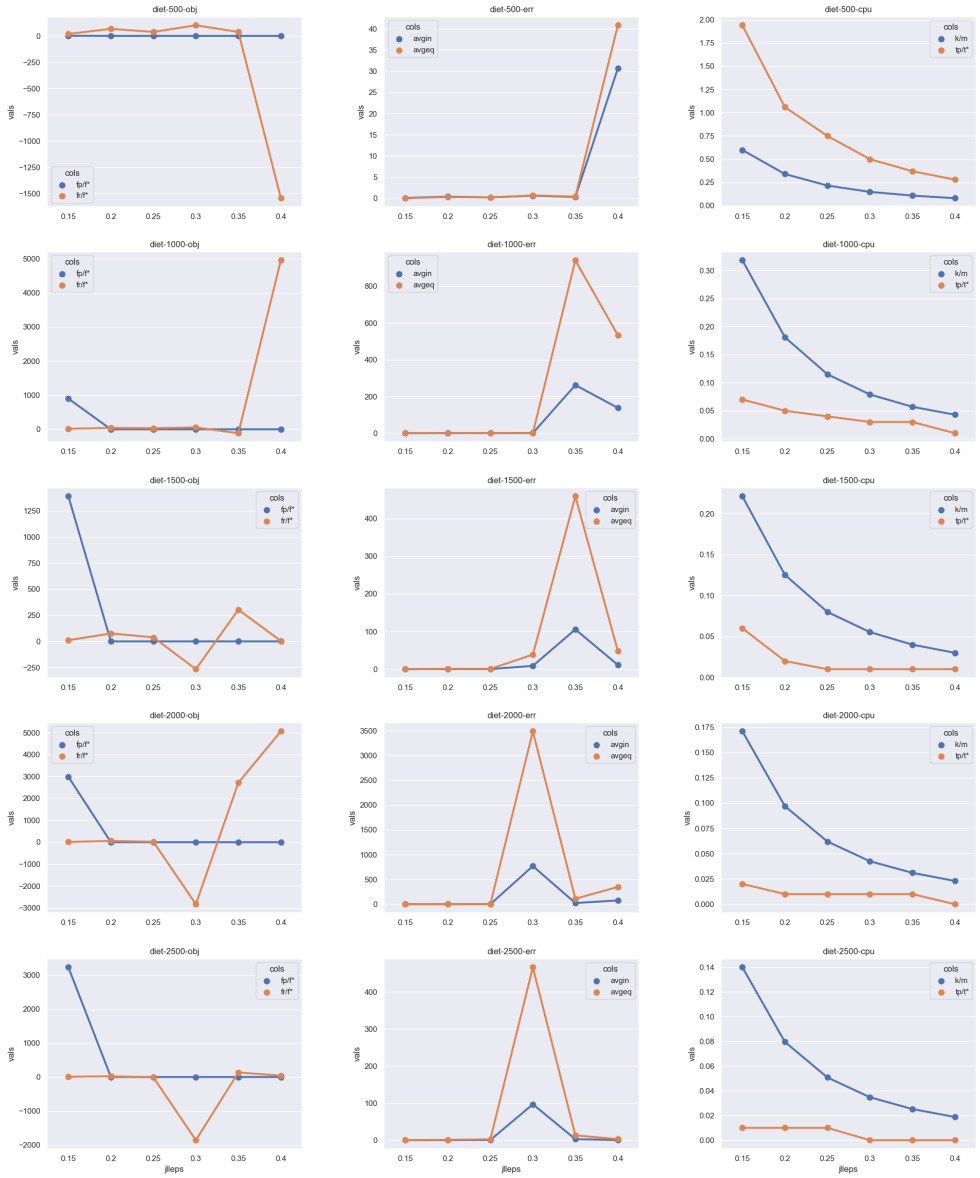


Figure 3 DIET plots with modified objective attempting to drive the slack variables to zero.

5 Conclusion

In this paper we have pursued a computational study of the application of random projections to linear program data, based on solving original and projected formulations linear program instances of various structures and sizes. We found that original formulations only involving inequalities are particularly challenging, but those that natively involve equations behave better. The sparsity of the constraint matrix does not appear to pose issues, as long as sparse RPs are used. Lastly, the sizes we considered here are possibly at the lower end of the range allowed by RPs: better results should be obtained with larger sizes and smaller values of ϵ, which in turn imply larger CPU times.

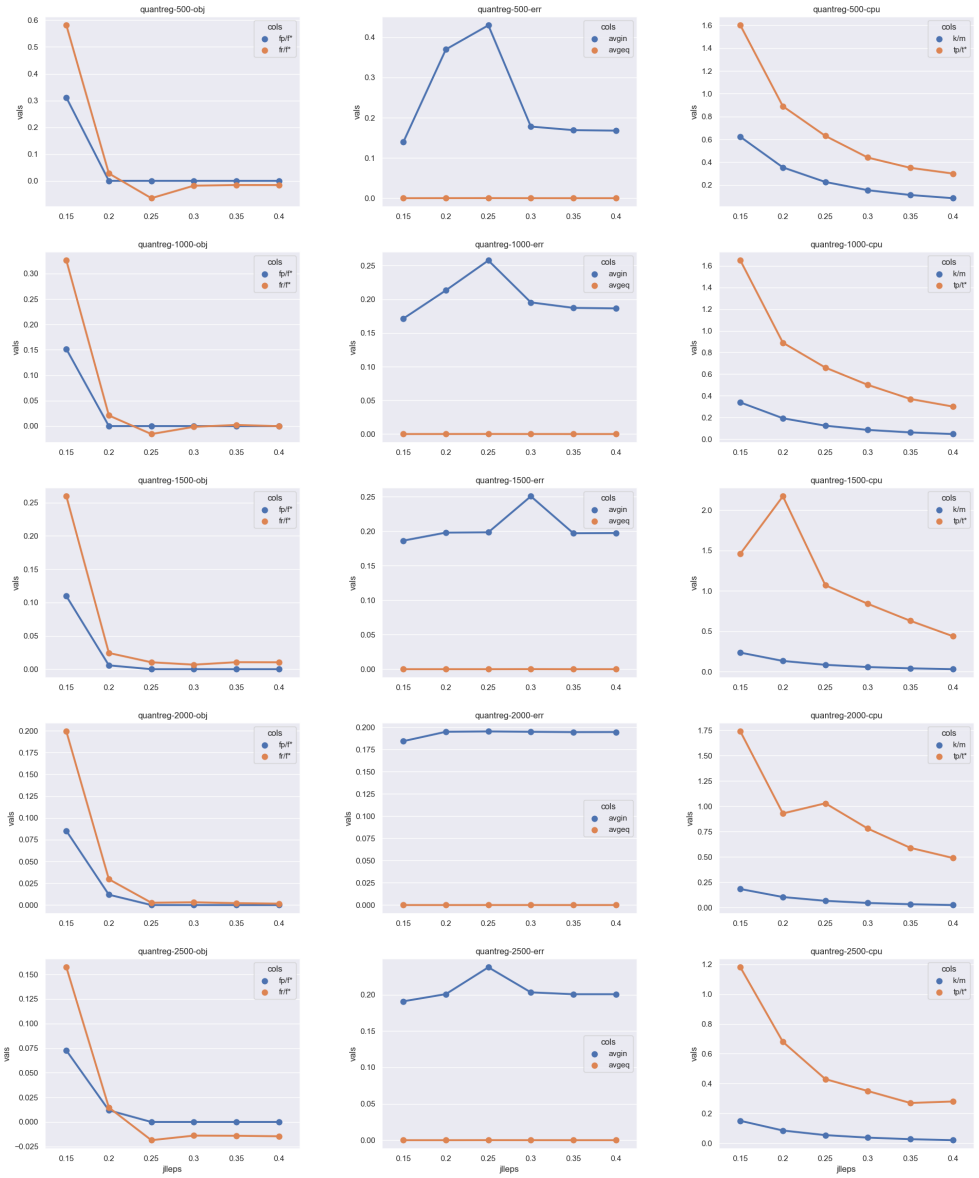


Figure 4 QUANTILE REGRESSION plots (increasing ϵ on abscissae): instances of growing size on rows, objective function ratios on the first column, feasibility errors on the second, k/m and CPU time ratio on the third.

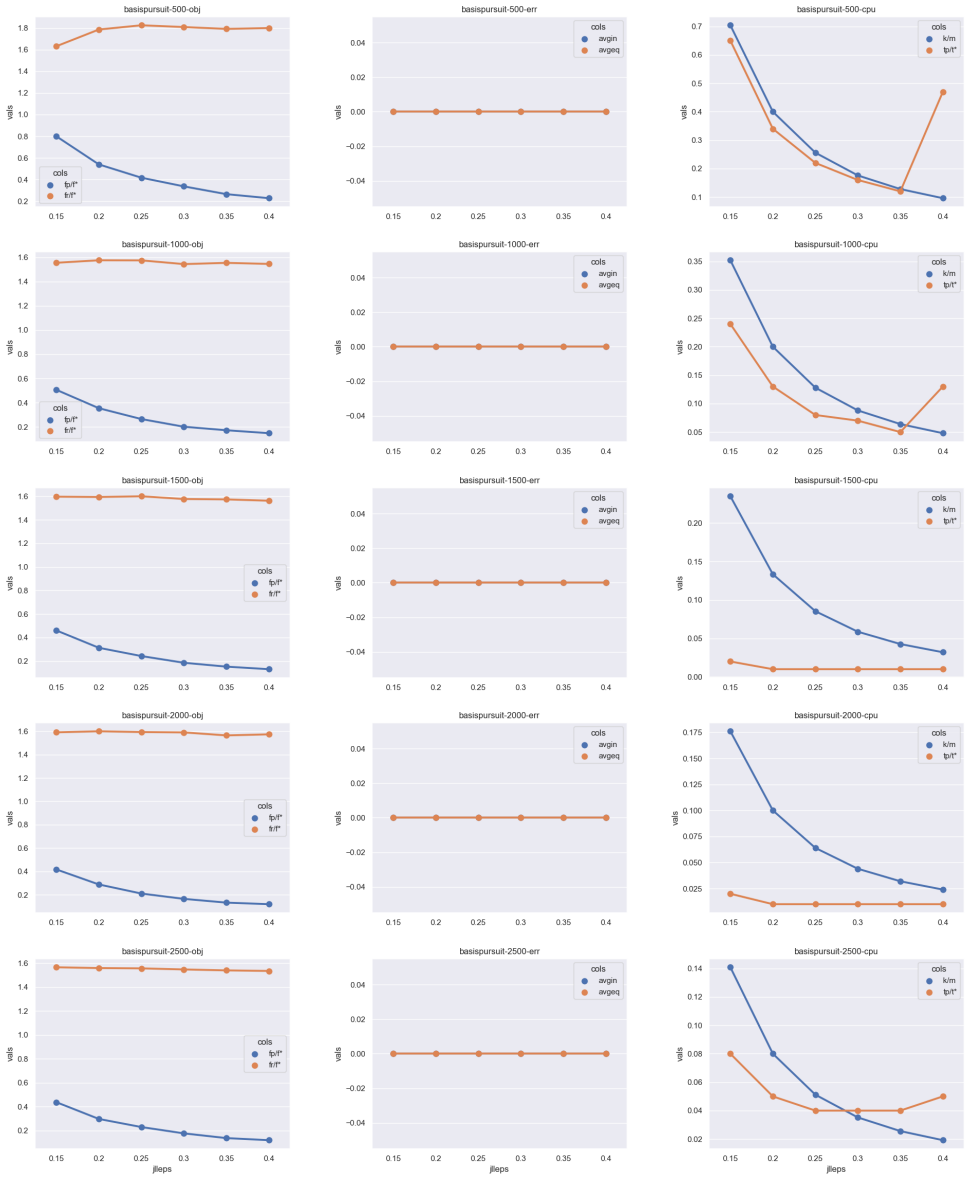


Figure 5 Basis Pursuit plots (increasing ϵ on abscissae): instances of growing size on rows, objective function ratios on the first column, feasibility errors on the second, k/m and CPU time ratio on the third.

References

1 D. Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66:671–687, 2003.

2 D. Amelunxen, M. Lotz, M. McCoy, and J. Tropp. Living on the edge: phase transitions in convex programs with random data. *Information and Inference: A Journal of the IMA*, 3:224–294, 2014.

3 E. Candès and T. Tao. Decoding by Linear Programming. *IEEE Transactions on Information Theory*, 51(12):4203–4215, 2005.

4 C. D'Ambrosio, L. Liberti, P.-L. Poirion, and K. Vu. Random projections for quadratic programs. *Mathematical Programming B*, 183:619–647, 2020.
5 S. Damelin and W. Miller. *The mathematics of signal processing*. CUP, Cambridge, 2012.
6 G. Dantzig. The Diet Problem. *Interfaces*, 20(4):43–47, 1990.
7 S. Dirksen. Dimensionality reduction with subgaussian matrices: A unified theory. *Foundations of Computational Mathematics*, 16:1367–1396, 2016.
8 L. Ford and D. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.
9 R. Fourer and D. Gay. *The AMPL Book*. Duxbury Press, Pacific Grove, 2002.
10 IBM. *ILOG CPLEX 20.1 User's Manual*. IBM, 2020.
11 P. Indyk. Algorithmic applications of low-distortion geometric embeddings. In *Foundations of Computer Science*, volume 42 of *FOCS*, pages 10–33, Washington, DC, 2001. IEEE.
12 P. Indyk and A. Naor. Nearest neighbor preserving embeddings. *ACM Transactions on Algorithms*, 3(3):Art. 31, 2007.
13 W. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In G. Hedlund, editor, *Conference in Modern Analysis and Probability*, volume 26 of *Contemporary Mathematics*, pages 189–206, Providence, RI, 1984. AMS.
14 E. Jones, T. Oliphant, and P. Peterson. `SciPy`: Open source scientific tools for Python, 2001. [Online; accessed 2016-03-01]. URL: `http://www.scipy.org/`.
15 D. Kane and J. Nelson. Sparser Johnson-Lindenstrauss transforms. *Journal of the ACM*, 61(1):4, 2014.
16 R. Koenker. *Quantile regression*. CUP, Cambridge, 2005.
17 L. Liberti. Decoding noisy messages: a method that just shouldn't work. In A. Deza, S. Gupta, and S. Pokutta, editors, *Data Science and Optimization*. Fields Institute, Toronto, pending minor revisions.
18 L. Liberti, P.-L. Poirion, and K. Vu. Random projections for conic programs. *Linear Algebra and its Applications*, 626:204–220, 2021.
19 L. Liberti and K. Vu. Barvinok's naive algorithm in distance geometry. *Operations Research Letters*, 46:476–481, 2018.
20 D. Pucci de Farias and B. Van Roy. On constraint sampling in the Linear Programming approach to approximate Dynamic Programming. *Mathematics of Operations Research*, 29(3):462–478, 2004.
21 G. van Rossum and *et al. Python Language Reference, version 3*. Python Software Foundation, 2019.
22 S. Vempala. *The Random Projection Method*. Number 65 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science. AMS, Providence, RI, 2004.
23 S. Venkatasubramanian and Q. Wang. The Johnson-Lindenstrauss transform: An empirical study. In *Algorithm Engineering and Experiments*, volume 13 of *ALENEX*, pages 164–173, Providence, RI, 2011. SIAM.
24 K. Vu, P.-L. Poirion, C. D'Ambrosio, and L. Liberti. Random projections for quadratic programs over a Euclidean ball. In A. Lodi and *et al.*, editors, *Integer Programming and Combinatorial Optimization (IPCO)*, volume 11480 of *LNCS*, pages 442–452, New York, 2019. Springer.
25 K. Vu, P.-L. Poirion, and L. Liberti. Random projections for linear programming. *Mathematics of Operations Research*, 43(4):1051–1071, 2018.
26 J. Yang, X. Meng, and M. Mahoney. Quantile regression for large-scale applications. *SIAM Journal of Scientific Computing*, 36(5):S78–S110, 2014.

Computing Maximal Unique Matches with the r-Index

Sara Giuliani
Department of Computer Science, University of Verona, Italy

Giuseppe Romana
Department of Computer Science, University of Palermo, Italy

Massimiliano Rossi
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA

Abstract

In recent years, pangenomes received increasing attention from the scientific community for their ability to incorporate population variation information and alleviate reference genome bias. Maximal Exact Matches (MEMs) and Maximal Unique Matches (MUMs) have proven themselves to be useful in multiple bioinformatic contexts, for example short-read alignment and multiple-genome alignment. However, standard techniques using suffix trees and FM-indexes do not scale to a pangenomic level. Recently, Gagie et al. [JACM 20] introduced the r-index that is a Burrows-Wheeler Transform (BWT)-based index able to handle hundreds of human genomes. Later, Rossi et al. [JCB 22] enabled the computation of MEMs using the r-index, and Boucher et al. [DCC 21] showed how to compute them in a streaming fashion.

In this paper, we show how to augment Boucher et al.'s approach to enable the computation of MUMs on the r-index, while preserving the space and time bounds. We add additional $\mathcal{O}(r)$ samples of the longest common prefix (LCP) array, where r is the number of equal-letter runs of the BWT, that permits the computation of the second longest match of the pattern suffix with respect to the input text, which in turn allows the computation of candidate MUMs. We implemented a proof-of-concept of our approach, that we call MUM-PHINDER, and tested on real-world datasets. We compared our approach with competing methods that are able to compute MUMs. We observe that our method is up to 8 times smaller, while up to 19 times slower when the dataset is not highly repetitive, while on highly repetitive data, our method is up to 6.5 times slower and uses up to 25 times less memory.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Burrows–Wheeler Transform, r-index, maximal unique matches, bioinformatics, pangenomics

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.22

Related Version *Full Version*: `https://arxiv.org/abs/2205.01576`

Supplementary Material *Software*: `https://github.com/saragiuliani/mum-phinder`
archived at `swh:1:dir:5408c1a53cd0cb26926d545f3748ad0dbb207263`

Funding *Massimiliano Rossi*: National Science Foundation NSF EAGER (Grant No. 2118251), and National Institutes of Health (NIH) NIAID (Grant No. HG011392).

Acknowledgements We thank Travis Gagie for suggesting this problem as a project for his course CSCI 6905 at Dalhousie University. We also thank the anonymous reviewers for their insightful comments.

1 Introduction

With the advent of third-generation sequencing, the quality of assembled genomes drastically increased. In the last year the Telomere-to-Telomere project released the first complete haploid human genome [19] and the Human Pangenome Reference Consortium (HPRC) plans to release hundreds of high-quality assembled genomes to be used as a pangenome reference. One important step to enable the use of these high-quality assembled genomes is to build a multiple-sequence alignment of the genomes. Tools like MUMmer [13, 18], and Mauve [5] proposed a solution to the original problem of multiple-sequence alignment by using Maximal Unique Matches (MUMs) between two input sequences as prospective anchors for an alignment. MUMs are long stretches of the genomes that are equal in both genomes and occur only once in each of them. To reduce the computational costs of computing the MUMs, progressive approaches have also been developed like progressive Mauve [6] and progressive Cactus [1] that enables the construction of pangenome graphs, among others, that have been used in recent aligners like Giraffe [21]. MUMs have also been proven useful for strain level read quantification [23], and as a computationally efficient genomic distance measure [7].

Recent advances in pangenomics [20, 3] demonstrated that it is possible to index hundreds of Human Genomes and to query such an index to find supersets of MUMs that are maximal exact matches (MEMs), which are substrings of the pattern that occur in the reference and that cannot be extended neither on the left nor on the right. The tool called MONI [20] requires two passes over the query sequence to report the MEMs. Later PHONI [3] showed how to modify the query to compute the MEMs in a streaming fashion, with only one single pass over the query string. Both MONI and PHONI are built on top of an r-index [11] and a straight-line program SLP [9]. Their main objective is to compute the so called matching statistics (see Definition 3) of the pattern with respect to the text, that can be used to compute the MEMs with a linear scan. While, MONI uses the SLP for random access to the text, and needs to store additional information to compute the matching statistics and the MEMs, PHONI uses the SLP to compute efficient longest common extension (LCE) queries which allow to compute the matching statistics and the MEMs with only one scan of the query.

We present MUM-PHINDER, a tool that is able to compute MUMs of a query pattern against an index on a commodity computer. The main observation of our approach is to extend the definition of matching statistics to include, for each suffix of the pattern, the information of the length of the second longest match of the suffix in the text, which allows to decide whether a MEM is also unique. We extended PHONI to keep track at each step of the query, the second longest match of the pattern in the index, and its length. To do this, we add $O(r)$ samples of the longest common prefix (LCP) array to PHONI.

We evaluated our algorithm on real-world datasets, and we tested MUM-PHINDER against MUMmer [18]. We measured time and memory required by both tools for sets of increasing size of haplotypes of human chromosome 19 and SARS-CoV2 genomes and queried using one haplotype of chromosome 19 and one SARS-CoV2 genome not present in the dataset. We report that MUM-PHINDER requires consistently less memory than MUMer for all experiments being up to 25 times smaller. Although MUMer is generally faster than ours (18 times faster for 1 haplotype of chromosome 19, and 6.5 times faster for 12,500 SARS-CoV2 genomes), it cannot process longer sequences due to memory limitations. Additionally, we observe that when increasing the number of sequences in the dataset, the construction time of MUM-PHINDER increases, while the query time decreases. This phenomenon is due to the

increase in the number of matches in the search process, that prevents the use of more computational-demanding operations. Note that, due to the use of the r-index, the efficiency of our method increases when the dataset is highly repetitive as in the case of pangenomes.

2 Preliminaries

Let $\Sigma = \{a_0 < a_1 < \ldots < a_{\sigma-1}\}$ be an *ordered alphabet*, where $<$ represents the lexicographical order. A *string* (or *text*) T is a sequence of characters $T[0]T[1]\cdots T[n-1]$ such that $T[j] \in \Sigma$ for all $j \in [0..n)$. The length of a string is denoted by $|T|$. We refer to the empty string with ε, that is the only substring of length 0.

We denote a factor (or substring) of T as $T[i..j) = T[i]T[i+1]\cdots T[j-1]$ if $i < j$, and $T[i..j) = \varepsilon$ otherwise. We refer to $T[0..j)$ as the $j-1$-th prefix of T and to $T[i..n)$ as the i-th suffix of T.

We assume throughout the paper that the text T is terminated by termination character \$ that does not occur in the original text and it is lexicographically smaller than all the other characters in the alphabet.

Suffix array, inverse suffix array, and longest common prefix array

The *Suffix array* (SA) of a string $T[0..n)$ is an array of length n such that $T[\mathsf{SA}[i]..n) < T[\mathsf{SA}[j]..n)$ for any $0 \le i < j < n$. The *Inverse Suffix array* (ISA) is the inverse of SA, i.e. $\mathsf{ISA}[i] = j$ if and only if $\mathsf{SA}[j] = i$. Let $lcp(u, v)$ be the length of the longest common prefix between two strings u and v, that is $u[0..lcp(u,v)) = v[0..lcp(u,v))$ but $u[lcp(u,v)] \neq v[lcp(u,v)]$ (assuming $lcp(u,v) < \min\{|u|, |v|\}$). The *Longest Common Prefix array* (LCP) of $T[0..n)$ is an array of length n such that $\mathsf{LCP}[0] = 0$ and $\mathsf{LCP}[i] = lcp(T[\mathsf{SA}[i-1]..n), T[\mathsf{SA}[i]..n))$, for any $0 < i < n$.

Burrows-Wheeler Transform, Run-Length Encoding, and r-index

The *Burrows-Wheeler Transform* (BWT) of T is a reversible transformation of the characters of T [4]. That is the concatenation of the characters preceding the suffixes of T listed in lexicographic order, i.e., for all $0 \le i < n$, $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1 \mod n]$. The LF-*mapping* is the function that maps every character in the BWT with its preceding text character, in the BWT, i.e. $\mathsf{LF}(i) = \mathsf{ISA}[\mathsf{SA}[i] - 1 \mod n]$.

The *run-length encoding* of a string T is the representation of maximal equal-letter runs of T as pairs (c, ℓ), where c is the letter of the run and $\ell > 0$ is the length of the run. For example, the run length encoding of $T = AAACAAGGGG$ is $(A,3)(C,1)(A,2)(G,4)$. We refer to the number of runs of the BWT with r.

The BWT tends to create long equal-letter runs on highly repetitive texts such as genomic datasets. The run-length encoding applied to the BWT (in short RLBWT) is the basis of many lossless data compressors and text indexes, such as the FM-index [8] which is the base of widely used bioinformatics tools such as Bowtie [14] and BWA [15]. Although the BWT can be stored and queried in compressed space [17], the number of samples of the SA required by the index grows with the length of the uncompressed text. To overcome this issue Gagie et al. [11] proposed the *r-index* whose number of SA samples grows with the number of runs r of the BWT. The r-index is a text index composed by the run-length encoded BWT and the SA sampled at run boundaries, i.e., in correspondence of the first and last character of a run of the BWT, and it is able to retrieve the missing values of the SA by using a predecessor data structure on the samples of the SA.

Grammar and straight-line program

A *context-free grammar* $\mathcal{G} = \{V, \Sigma, R, S\}$ consists in a set of *variables* V, a set of *terminal symbols* Σ, a set of *rules* R of the type $A \mapsto \alpha$, where $A \in V$ and $\alpha \in \{V \cup \Sigma\}^*$, and the *start variable* $S \in V$. The *language of the grammar* $\mathcal{L}(\mathcal{G}) \subseteq \Sigma^*$ is the set of all words over the alphabet of terminal symbols generated after applying some rules in R starting from S. When $\mathcal{L}(\mathcal{G})$ contains only one string T, that is $\mathcal{G}$ only generates T, then the grammar $\mathcal{G}$ is called *straight-line program* (SLP).

Longest Common Extension, rank, and select queries

Given a text $T[0..n)$, the *longest common extension* (LCE) query between two positions $0 \leq i, j < n$ in T is the length of the longest common prefix of $T[i..n)$ and $T[j..n)$. Thus, if $\ell = \mathsf{LCE}(i, j)$, then $T[i..i+\ell) = T[j..j+\ell)$ and either $T[i+\ell] \neq T[j+\ell]$ or either $i + \ell = n$ or $j + \ell = n$.

Given a character c and an integer i, we define $T.\mathsf{rank}_c(i)$ as the number of occurrences of the character c in the prefix $T[0..i)$, while we define $T.\mathsf{select}_c(i)$ as the position $p \in [0..n)$ of the ith occurrence of c in T if it exists, and $p = n$ otherwise.

3 Computing MUMs using MS

Given a text $T[0..n)$ and a pattern $P[0..m)$, we refer to any factor in P that also occurs in T as a *match*. A match w in P can be defined as a pair (i, ℓ) such that $w = P[i..i+\ell)$. We say that w is *maximal* if the match can not be extended neither on the left nor on the right, i.e. either $i = 0$ or $P[i-1..i+\ell)$ does not occur in T and either $i = m - \ell$ or $P[i..i+\ell+1)$ does not occur in T.

▶ **Definition 1.** *Given a text T and a pattern P, a Maximal Unique Match (*MUM*) is a maximal match that occurs exactly once in T and P.*

▶ **Example 2.** Let T = ACACTCTTAC**ACC**ATATCATCAA\$ be the text and P = AACCTAA the pattern. The factor **AA** is maximal in P and occurs only once in T, while it is repeated in P at positions 0 and 5. The factor **CT** of P starting in position 3 is a maximal match that occurs only once in P, but it is not unique in T. The factor **CC** of P starting in position 2 is unique in both T and P, but both can be extended on the left with an **A**. On the other hand, the factor $P[1..4) = T[10..13)$ =**ACC** is a MUM.

From now on, we refer to the set of all maximal unique matches between T and P as MUMs. In [3] the authors showed how to compute maximal matches (not necessarily unique neither in T nor P) in $\mathcal{O}(r + g)$ space, where r is the number of runs of the BWT of T and g is the size of the SLP representing the text T. This is achieved by computing the *matching statistics*, for which we report the definition given in [3].

▶ **Definition 3** ([3])**.** *The matching statistics* MS *of a pattern $P[0..m)$ with respect to a text $T[0..n)$ is an array of (position, length)-pairs* $\mathsf{MS}[0..m)$ *such that*

- $P[i..i + \mathsf{MS}[i].\mathsf{len}) = T[\mathsf{MS}[i].\mathsf{pos}..\mathsf{MS}[i].\mathsf{pos} + \mathsf{MS}[i].\mathsf{len})$*;*
- *either* $i = m - \mathsf{MS}[i].\mathsf{len}$ *or* $P[i..i + \mathsf{MS}[i].\mathsf{len} + 1)$ *does not occur in* T*.*

That is, $\mathsf{MS}[i].\mathsf{pos}$ *is the starting position in T of an occurrence of the longest prefix of $P[i..m)$ that occurs in T, and* $\mathsf{MS}[i].\mathsf{len}$ *is its length.*

A known property of the matching statistics is that for all $i > 0$, $\mathsf{MS}[i].\mathsf{len} \geq \mathsf{MS}[i-1].\mathsf{len} - 1$.

Our objective is to show how to further compute MUMs within the same space bound. For our purpose, we extend the definition of MS array with an additional information field to each entry.

▶ **Definition 4.** *Given a text $T = [0 \dots n)$ and a pattern $P = [0 \dots m)$, we define the extended matching statistics* eMS *as an array of* $(\mathsf{pos}, \mathsf{len}, \mathsf{slen})$*-tuples* $\mathsf{eMS}[0 \dots m)$ *such that*

- $\mathsf{eMS}[i].\mathsf{pos} = \mathsf{MS}[i].\mathsf{pos}$ *and* $\mathsf{eMS}[i].\mathsf{len} = \mathsf{MS}[i].\mathsf{len}$*;*
- $\mathsf{eMS}[i].\mathsf{slen}$ *is the largest value ℓ for which there exists $p \neq \mathsf{eMS}[i].\mathsf{pos}$ such that $P[i..i+\ell) = T[p..p+\ell)$.*

In other words, $\mathsf{eMS}[i].\mathsf{slen}$ *is the length of the second longest match of a prefix $P[i..n)$ in T.*

Note that $\mathsf{eMS}[i].\mathsf{slen} \leq \mathsf{eMS}[i].\mathsf{len}$, for any $i \in [0..m)$.

3.1 Checking Maximality and Uniqueness of matches

We now show how to compute MUMs by using the eMS array. Lemma 5 shows how to verify if a match occurs only once in T.

▶ **Lemma 5.** *Given a text T, a pattern P, and the* eMS *array computed for P with respect to T, let $w = P[i..i + \mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len})$ be a maximal match between a pattern $P[0..m)$ and a text $T[0..n)\$$. Then w occurs exactly once in T if and only if $\mathsf{eMS}[i].\mathsf{slen} < \mathsf{eMS}[i].\mathsf{len}$.*

Proof. For the *if* direction, we assume by contradiction that w is unique in T and that $\mathsf{eMS}[i].\mathsf{slen} \geq \mathsf{eMS}[i].\mathsf{len}$. By definition, $\mathsf{eMS}[i].\mathsf{slen} \leq \mathsf{eMS}[i].\mathsf{len}$, hence we assume $\mathsf{eMS}[i].\mathsf{slen} = \mathsf{eMS}[i].\mathsf{len}$. By definition of $\mathsf{eMS}[i].\mathsf{slen}$ there exists $p \neq \mathsf{eMS}[i].\mathsf{pos}$ such that $w = P[i..i + \mathsf{eMS}[i].\mathsf{slen}) = T[p..p + \mathsf{eMS}[i].\mathsf{slen}) = T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len})$, that contradicts the assumption that w occurs only once in the text T. Analogously, assume that $\mathsf{eMS}[i].\mathsf{slen} < \mathsf{eMS}[i].\mathsf{len}$ and that there exists a position $j \neq \mathsf{eMS}[i].\mathsf{pos}$ such that $T[j..j+\mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos}+\mathsf{eMS}[i].\mathsf{len})$. However, this is in contradiction with the definition of $\mathsf{eMS}[i].\mathsf{slen}$ and the assumption of $\mathsf{eMS}[i].\mathsf{slen} < \mathsf{eMS}[i].\mathsf{len}$, concluding the proof. ◀

We check the maximality of a match in the pattern using an analogous approach as in [20], that we summarize with the following lemma.

▶ **Lemma 6.** *Given a text T, a pattern P, and the* eMS *array computed for P with respect to T, let $w = P[i..i + \mathsf{eMS}[i].\mathsf{len})$ be a match with a text T. Then w is a maximal match if and only if either $i = 0$ or $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len}$.*

Proof. First we show that if $w = P[i..i + \mathsf{eMS}[i].\mathsf{len})$ is a maximal match then either $i = 0$ or $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len}$. Let us assume that w is not maximal and either $i = 0$ or $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len}$, hence either $P[i..i + \mathsf{eMS}[i].\mathsf{len} + 1)$ occurs in T or $P[i-1..i + \mathsf{eMS}[i].\mathsf{len})$ occurs in T. The former case is in contradiction with the definition of eMS, hence $P[i-1..i + \mathsf{eMS}[i].\mathsf{len})$ occurs in T. This implies that $i > 0$ and that $\mathsf{eMS}[i-1].\mathsf{len} = \mathsf{eMS}[i].\mathsf{len} + 1$ in contradiction with the hypothesis that $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len}$.

Now we show that if either $i = 0$ or $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len}$ then w is a maximal match. By definition of $\mathsf{eMS}[i].\mathsf{len}$, we know that either $i + \mathsf{eMS}[i].\mathsf{len} = m$ or $P[i..i + \mathsf{eMS}[i].\mathsf{len} + 1)$ does not occur in $T\$$, that is w cannot be extended on the right in P. If $i = 0$ we can not further extend the match w on the left, hence w is maximal. If $i > 0$, then by definition of matching statistics it holds that $\mathsf{eMS}[i-1].\mathsf{len} \leq \mathsf{eMS}[i].\mathsf{len} + 1$. Note that if there exists a

character $a \in \Sigma$ such that $P[i-1..i-1+\mathsf{eMS}[i-1].\mathsf{len}) = aw$ and aw occurs in T, then $\mathsf{eMS}[i-1] = \mathsf{eMS}[i]+1$. Hence if $\mathsf{eMS}[i-1] = \mathsf{eMS}[i]+1$ then it is easy to see that w is not maximal because it can be extended on the left. It also follows that if $\mathsf{eMS}[i-1] \leq \mathsf{eMS}[i]$ then w cannot be extended on the left, hence it is maximal and the thesis follows. ◀

Let $\mathcal{L} \subseteq [0..m)$ be the subset of positions in P such that both Lemma 5 and Lemma 6 hold, i.e. $\mathcal{L}$ contains all the positions in P where a maximal match unique in T starts. One can notice that if a match $w_i = P[i..i+\mathsf{eMS}[i].\mathsf{len})$ is a MUM, then $i \in \mathcal{L}$.

We first show that given $i \in \mathcal{L}$, if a match w_i is not unique in P, then the second occurrence of w_i in P is contained in another maximal match unique in T.

▶ **Lemma 7.** *Given a text T, a pattern P, and the* eMS *array computed for P with respect to T, let $\mathcal{L}$ be the subset of positions in P such that $w_i = P[i..i+\mathsf{eMS}[i].\mathsf{len})$ is maximal and occurs only once in T for all $i \in \mathcal{L}$. Then, w_i is not unique in P if and only if there exist $i' \in \mathcal{L} \setminus \{i\}$ and two possibly empty strings u, v such that $w_{i'} = uw_iv$ is a factor of P.*

Proof. Let us assume by contradiction that such i' does not exist, then let $j \notin \mathcal{L}$ be such that $P[j..j+|w_i|) = w_i$. Since $j \notin \mathcal{L}$ then either $P[j..j+|w_i|)$ is not unique in T, or it is not maximal. The former case it contradicts $i \in \mathcal{L}$ because $P[j..j+|w_i|) = w_i$ occurs twice in T. Hence, $P[j..j+|w_i|)$ occurs only once in T and it is not maximal, therefore there exists $k \in \mathcal{L}$ such that $k \leq j$ and $|w_k| > |w_i|$ which contradict the hypothesis. The other direction of the proof is straightforward since by definition of $w_{i'}$, either w_i occurs twice in P or it is not maximal. ◀

The following Lemma shows, for any $i \in \mathcal{L}$, if a match w_i is unique in P by using the eMS array.

▶ **Lemma 8.** *Given a text T, a pattern P, and the* eMS *array computed for P with respect to T, let $\mathcal{L}$ be the subset of positions in P such that $w_i = P[i..i+\mathsf{eMS}[i].\mathsf{len})$ is maximal and occurs only once in T, for all $i \in \mathcal{L}$. Then, w_i occurs only once in P if and only if, for all $i' \in \mathcal{L} \setminus \{i\}$, either $\mathsf{eMS}[i].\mathsf{pos} < \mathsf{eMS}[i'].\mathsf{pos}$ or $\mathsf{eMS}[i].\mathsf{len} + \mathsf{eMS}[i].\mathsf{pos} > \mathsf{eMS}[i'].\mathsf{len} + \mathsf{eMS}[i'].\mathsf{pos}$.*

Proof. We first show that if w_i occurs only once in P then for all $i' \in \mathcal{L} \setminus \{i\}$, either $\mathsf{eMS}[i].\mathsf{pos} < \mathsf{eMS}[i'].\mathsf{pos}$ or $\mathsf{eMS}[i].\mathsf{len} + \mathsf{eMS}[i].\mathsf{pos} > \mathsf{eMS}[i'].\mathsf{len} + \mathsf{eMS}[i'].\mathsf{pos}$. Since $\mathcal{L}$ contains only positions of maximal matches unique in T, then for all for $i \in \mathcal{L}$ we can map w_i to its occurrence in the text $T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos}+\mathsf{eMS}[i].\mathsf{len})$. Since w_i occurs only once in T, by Lemma 7 we have that $\mathsf{eMS}[i'].\mathsf{pos} = \mathsf{eMS}[i].\mathsf{pos} - |u|$ and $\mathsf{eMS}[i'].\mathsf{len} = \mathsf{eMS}[i].\mathsf{len} + |u| + |v|$. Hence, $\mathsf{eMS}[i'].\mathsf{pos} \leq \mathsf{eMS}[i].\mathsf{pos}$ and $\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len} \leq \mathsf{eMS}[i'].\mathsf{pos} + \mathsf{eMS}[i'].\mathsf{len}$.

We now show the other direction of the implication. If given a position $i \in \mathcal{L}$ for all $i' \in \mathcal{L} \setminus \{i\}$, either $\mathsf{eMS}[i].\mathsf{pos} < \mathsf{eMS}[i'].\mathsf{pos}$ or $\mathsf{eMS}[i].\mathsf{len} + \mathsf{eMS}[i].\mathsf{pos} > \mathsf{eMS}[i'].\mathsf{len} + \mathsf{eMS}[i'].\mathsf{pos}$ then w_i occurs only once in P. Assuming by contradiction that there exists a position $i \in \mathcal{L}$ such that for all $i' \in \mathcal{L} \setminus \{i\}$, either $\mathsf{eMS}[i].\mathsf{pos} < \mathsf{eMS}[i'].\mathsf{pos}$ or $\mathsf{eMS}[i].\mathsf{len} + \mathsf{eMS}[i].\mathsf{pos} > \mathsf{eMS}[i'].\mathsf{len} + \mathsf{eMS}[i'].\mathsf{pos}$ and w_i does not occur only once in P, then by Lemma‘7 there exist $j \in \mathcal{L}$ and two possibly empty strings u, v such that $w_j = uw_iv$ is a factor of P. It is easy to see that $\mathsf{eMS}[j].\mathsf{pos} = \mathsf{eMS}[i].\mathsf{pos} - |u|$ and $\mathsf{eMS}[j].\mathsf{len} = \mathsf{eMS}[i].\mathsf{len} + |u| + |v|$. Hence, $\mathsf{eMS}[j].\mathsf{pos} \leq \mathsf{eMS}[i].\mathsf{pos}$ and $\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len} \leq \mathsf{eMS}[j].\mathsf{pos} + \mathsf{eMS}[j].\mathsf{len}$, in contradiction with the hypothesis, concluding the proof. ◀

We can summarize the previous Lemmas in the following Theorem.

▶ **Theorem 9.** *Given a text T, a pattern P, and the* eMS *array computed for P with respect to T, for all $0 \leq i < m$, $w_i = P[i..i+\mathsf{eMS}[i].\mathsf{len})$ is a* MUM *if and only if $i \in \mathcal{L}$ and Lemma 8 holds.*

▶ **Example 10.** Let T = ACACTCTTAC**ACC**ATATCATCAA\$ be the text and P = AACCTAA the pattern. In the table below we report the values of the eMS of P with respect to T.

i	0	1	2	3	4	5	6
$P[i]$	A	A	C	C	T	A	A
eMS$[i]$.pos	21	10	11	5	6	21	8
eMS$[i]$.len	2	3	2	2	2	2	1
eMS$[i]$.slen	1	2	1	2	2	1	1

It is easy to check that $\mathcal{L} = \{0, 1, 5\}$, where $\mathcal{L}$ contains those indices i which verify both Lemma 5 (eMS$[i]$.slen $<$ eMS$[i]$.len) and Lemma 6 (either $i = 0$ or eMS$[i-1]$.len $\leq$ eMS$[i]$.len). Note that eMS[0].pos = eMS[5].pos and eMS[0].len = eMS[5].len, and by Lemma 8 we know that $P[0..2) (= P[5..7))$ is repeated in P. Since eMS[1].pos $<$ eMS[0].pos = eMS[5].pos, by Theorem 9 the match $P[1..4) = T[10..13)$ = ACC is a MUM.

3.2 Computing the second longest match

Now we show how we can compute eMS extending the algorithm presented in Boucher et al. [3] while preserving the same space-bound.

We can apply verbatim the algorithm of [3] to compute the eMS$[i]$.pos and eMS$[i]$.len while we extend the algorithm to include the computation of eMS$[i]$.slen. The following Lemma shows how to find the second longest match using the LCP array.

▶ **Lemma 11.** *Given a text T, a pattern P, and the* eMS *array of P with respect to T, let $P[i..i + \text{eMS}[i].\text{len}) = T[\text{eMS}[i].\text{pos}..\text{eMS}[i].\text{pos} + \text{eMS}[i].\text{len})$ and $q = \text{ISA}[\text{eMS}[i].\text{pos}]$. Then, for all $0 \leq q < n$, $\text{eMS}[i].\text{slen} = \max\{\text{LCP}[q], \text{LCP}[q+1]\}$, where $\text{LCP}[n] = 0$.*

Proof. Let us consider the set $\mathcal{T} = \{w_0 < w_1 < \ldots < w_n\}$ of the lexicographically sorted suffixes of T. Then, for all $i \in [0..m)$, at least one suffix of T starting with the second longest match $P[i..i + \text{eMS}[i].\text{slen})$ must be adjacent to $w_q = T[\text{eMS}[i].\text{pos}..n)$ in $\mathcal{T}$. Hence, assuming $q \neq 0$ and $q \neq n$, eMS$[i]$.slen is either the LCP value between w_{q-1} and w_q or between w_q and w_{q+1}, that are respectively LCP$[q]$ and LCP$[q+1]$. Note that if $q = 0$ then both LCP[0] and LCP[1] exist, while for the case $q = n$ only LCP$[n]$ is available, that is eMS$[i]$.slen must be LCP$[n]$. ◀

4 Algorithm description

In this section we present the algorithm that we use to compute MUMs that builds on the approach of Boucher et al. [3] for the computation of the MS array. The authors showed how to use the r-index and the SLP of [10, 9] to compute the MS array of a pattern $P[0..m)$ in $\mathcal{O}(m \cdot (t_{\text{LF}} + t_{\text{LCE}} + t_{\text{pred}}))$ time, where t_{LF}, t_{LCE}, and t_{pred} represent the time to perform respectively one LF, one LCE, and one predecessor query. Our algorithm extends Boucher et al.'s method by storing additional $\mathcal{O}(r)$ samples of the LCP array. Given a text $T[0..n)$ and a pattern $P[0..m)$, in the following, we first show how to compute the eMS array of P with respect to T using the r-index, the SLP, and the additional LCP array samples. Then we show how to apply Theorem 9 to compute the MUMs from the eMS array.

4.1 Computing the eMS array

The key point of the algorithm is to extend the last computed match backwards when possible, otherwise we search for the new longest match that can be extended on the left by using the BWT. Let q be the index such that $P[i..i + \mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{SA}[q]..\mathsf{SA}[q] + \mathsf{eMS}[i].\mathsf{len})$ is the longest match found at step i:

- if $\mathsf{BWT}[q] = P[i-1]$, then it can be extended on the left, i.e. $P[i-1..i + \mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{SA}[q]-1..\mathsf{SA}[q] + \mathsf{eMS}[i].\mathsf{len})$;
- otherwise, we want to find the longest prefix of $P[i..i + \mathsf{eMS}[i].\mathsf{len})$ that is preceded by $P[i-1]$ in the text T. As observed in Bannai et al. [2] it can be either the suffix corresponding to the occurrence of $P[i-1]$ in the BWT immediately preceding or immediately following q, that we refer to as q_p and q_s respectively. Formally, $q_p = \max\{j < q \mid \mathsf{BWT}[j] = P[i-1]\}$ and $q_s = \min\{j > q \mid \mathsf{BWT}[j] = P[i-1]\}$.

The algorithm to compute the pos and len entry of the eMS array is analogous to the procedure detailed in [3]. We use the same data structures as the one defined in [3], that are the run-length encoded BWT and the samples of the SA in correspondence of positions q such that $\mathsf{BWT}[q]$ is either the first or the last symbol of an equal-letter run of the BWT. Note that both q_p and q_s are respectively the last and the first index of their corresponding equal-letter run.

An analogous reasoning can be formulated to compute the second longest match.

▶ **Lemma 12.** *Given a text $T[0..n)$, let* LCP, SA *and* ISA *be respectively the longest common prefix array, suffix array and inverse suffix array of T. Then, for all $0 < q \leq n$, let i, j be two integers such that $q - 1 = \mathsf{LF}[i]$ and $q = \mathsf{LF}[j]$, then if $\mathsf{BWT}[i] \neq \mathsf{BWT}[j]$ then $\mathsf{LCP}[q] = 0$, otherwise $\mathsf{LCP}[q] = \mathsf{LCE}(\mathsf{SA}[i], \mathsf{SA}[j]) + 1$.*

Proof. Let w_q be the q-th suffix in lexicographic order. Note that if $w_q = \$$ then $\mathsf{LCP}[q] = \mathsf{LCP}[q+1] = 0$. For all $1 \leq q < n$, if $w_{q-1} = a u\$$ and $w_q = b v\$$ for some $a < b \in \Sigma$ and some strings u and v, then $\mathsf{LCP}[q] = 0$. On the other hand, if $w_{q-1} = a u\$$ and $w_q = a v\$$, then $\mathsf{LCP}[q] = 1 + lcp(u\$, v\$)$. The thesis follows by observing that the suffixes $u\$$ and $v\$$ respectively correspond to w_i and w_j. ◀

Note that, the second longest match can be retrieved from the LCP values in correspondence of the longest maximal match (Lemma 11). Once we have the maximal match in position q in the BWT, we can compute $\mathsf{LCP}[q]$ and $\mathsf{LCP}[q+1]$ from the LCE queries on $T[\mathsf{SA}[q]..n)$ with $T[\mathsf{SA}[q_p]..n)$ and $T[\mathsf{SA}[q_s]..n)$ (Lemma 12).

Moreover, assuming the index q_p is the greatest index smaller than q such that $\mathsf{BWT}[q_p] = \mathsf{BWT}[q]$, then $\mathsf{LF}(q_p) = \mathsf{LF}(q) - 1$. It follows that if $\mathsf{BWT}[\mathsf{LF}(q_p)] = \mathsf{BWT}[\mathsf{LF}(q) - 1] = \mathsf{BWT}[\mathsf{LF}(q)]$, then $\mathsf{LCP}[\mathsf{LF}(q)])$ is an extension of the LCE query computed between $\mathsf{SA}[q_p]$ and $\mathsf{SA}[q]$ (see Figure 1). Symmetrically, if q_s is the smallest index greater than q such that $\mathsf{BWT}[q_s] = \mathsf{BWT}[q]$, then $\mathsf{LF}(q_s) = \mathsf{LF}(q) + 1$. Thus, at each iteration, we keep track of both LCP values computed to find the second longest match.

With respect to the implementation in [3], we add $\mathcal{O}(r)$ sampled values from the LCP array. Precisely, we store the LCP values between the first and the last two suffixes in correspondence of each equal-letter run (if only one suffix corresponds to a run we simply store 0). As shown later, this allows to overcome the problem of computing the LCE queries in case a position p in T is not stored in the sampled SA, i.e. when $\mathsf{ISA}[p]$ is neither the first nor the last index of its equal-letter run.

For simplicity of exposition we ignore the cases when a select query of a symbol c in the BWT fails. However, whenever it happens, either c does not occur in T or we are attempting to find an occurrence out of the allowed range, that is between 0 and the number

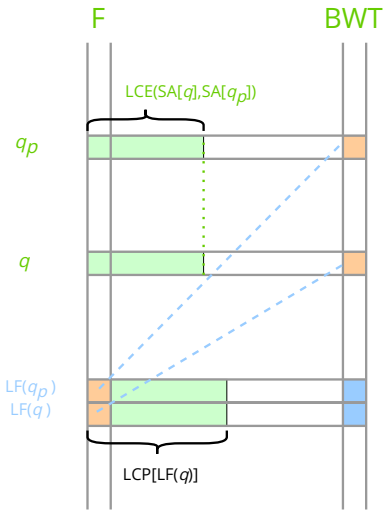


Figure 1 Application of Lemma 12 to compute $\mathsf{LCP}[\mathsf{LF}(q)]$ by extending the result of the last LCE query.

of occurrences of the character c minus 1. For the first case we can simply reset the algorithm starting from the next character of P to process, while the second occurs when we are attempting to compute an LCE query, whose result can be safely set to 0.

Algorithm 1 computes the extended matching statistics eMS of the pattern $P = [0 \dots m)$ with respect to the text $T = [0 \dots n)$ starting from the last element of the pattern (line 2). Moreover, we keep track of the first LCP values with respect to the maximal match of length 1 (line 3).

At each iteration of the loop (line 5), the algorithm tries to extend the match backwards position by position. If the match can be extended (line 7), then we use Algorithm 2 to compute the entry of the eMS. Otherwise, we use Algorithm 3 to compute the next entry of eMS (line 9).

Match case

Suppose $\mathsf{eMS}[i+1 \dots m)$ has already been processed and that $P[i] = T[\mathsf{eMS}[i+1].\mathsf{pos} - 1]$, namely we can further extend the longest match at the previous step by one position to the left. Algorithm 2 handles such scenario.

Let q be such that $\mathsf{SA}[q] = \mathsf{eMS}[i+1].\mathsf{pos} - 1$. Hence, we have that $\mathsf{eMS}[i].\mathsf{pos} = \mathsf{eMS}[i+1].\mathsf{pos} - 1$ and $\mathsf{eMS}[i].\mathsf{len} = \mathsf{eMS}[i+1].\mathsf{len} + 1$ (line 1). At this point, we search for the greatest index q_p among those smaller than q such that $\mathsf{BWT}[q_p] = P[i]$. As discussed before, when $q_p = q - 1$, then $\mathsf{LCP}[\mathsf{LF}(q)] = \mathsf{LCP}[q] + 1 = lcp_p + 1$ (line 3). Otherwise we can compute the LCE query between $\mathsf{SA}[q]$ and $\mathsf{SA}[q_p]$, to which we add 1 for the match with $P[i]$ in correspondence of $\mathsf{BWT}[q]$ and $\mathsf{BWT}[q_p]$ (line 6). Note that $\mathsf{SA}[q] = \mathsf{eMS}[i+1].\mathsf{pos}$, while q_p is the last index of its equal-letter run (and therefore $\mathsf{SA}[q_p]$ is stored).

Analogously we compute lcp_s (lines 7-10) and, by Lemmas 11 and 12, we assign to $\mathsf{eMS}[i].\mathsf{slen}$ the maximum between lcp_p and lcp_s.

Algorithm 1 Computation of eMS.

Input : Pattern $P[0, m)$
Output : Extended matching statistics $\mathsf{eMS}[0..m)$

```
1  q ← BWT.select_{P[m−1]}(1)
2  eMS[m − 1] ← (pos : SA[q] − 1, len : 1, slen : 1)
3  lcp_p ← 0, lcp_s ← 1
4  q ← LF(q)
5  for i ← m − 2 down to 0 do
6      if BWT[q] = P[i] then
7          eMS[i], lcp_p, lcp_s ← MSMatch(P[i], q, eMS[i+1].pos, eMS[i+1].pos, lcp_p, lcp_s)
8      else
9          eMS[i], lcp_p, lcp_s ←
             MSMisMatch(P[i], q, eMS[i + 1].pos, eMS[i + 1].pos, lcp_p, lcp_s)
10     q ← LF(q)
11 return eMS
```

Mismatch case

We use Algorithm 3 when q is such that $\mathsf{BWT}[q] \neq P[i]$. We search for the index q' in SA such that, among the suffixes of T preceded by $P[i]$, at position $\mathsf{SA}[q']$ in T starts the longest match with a prefix of $P[i+1..m)$. Note that $T[\mathsf{SA}[q'] - 1] = P[i]$, and that q' is either q_p or q_s.

Hence, if $q_p = q - 1$, then by Lemma 12 the longest common prefix of $T[\mathsf{SA}[q']..n)$ and $P[i+1..m)$ has length $lcp'_p = lcp_p$ computed at the previous step (line 5), otherwise we compute and store the LCE between $T[q..n)$ and $T[q_p..n)$ (line 7). A symmetric procedure is used to compute lcp'_s (lines 8-11).

Without loss of generality, we assume that $lcp'_s \geq lcp'_p$, hence $\mathsf{eMS}[i].\mathsf{pos} = \mathsf{SA}[q_s] - 1$. Then $\mathsf{eMS}[i].\mathsf{len} = lcp'_s + 1$ and $lcp_p = lcp'_p + 1$ (line 13). We add 1 to both lcp'_s and lcp'_p because both matches can be extended by one position on the left since $P[i] = \mathsf{BWT}[q_p] = \mathsf{BWT}[q_s]$. In order to compute $\mathsf{eMS}[i].\mathsf{slen}$ we need to compute the value of lcp_s with respect to q_s. To do so, we look for the smallest index q'_s greater than q_s such that $\mathsf{BWT}[q'_s] = P[i]$, and then apply a similar procedure to Algorithm 2 (lines 14-18). In this case, if $\mathsf{BWT}[q_s + 1] = P[i]$, then we can retrieve lcp_s from $\mathsf{LCP}[q_s + 1]$ since q_s is in correspondence of a run boundary. Symmetrically we handle the case $lcp'_p > lcp'_s$ (lines 20-26). Finally, we compute $\mathsf{eMS}[i].\mathsf{slen}$ by picking the maximum between lcp_p and lcp_s.

▶ **Theorem 13.** *Given a text $T[0..n)$, we can build a data structure in $\mathcal{O}(r + g)$ space that allows to compute the set MUMs between any pattern $P[0..m)$ and T in $\mathcal{O}(m \cdot (t_{\mathsf{LF}} + t_{\mathsf{LCE}} + t_{\mathsf{pred}}))$ time.*

Proof. Algorithm 1, Algorithm 2 and Algorithm 3 show how to compute the eMS array in m steps by using the data structure used in [3] of size $\mathcal{O}(r + g)$, to which we add $\mathcal{O}(r)$ words from the LCP array, preserving the space bound. Since at each step the dominant cost depends on the LF, LCE, and $\mathsf{rank}/\mathsf{select}$ queries, eMS is computed in $\mathcal{O}(m(t_{\mathsf{LF}} + t_{\mathsf{LCE}} + t_{\mathsf{pred}}))$ time. By Lemmas 5 and 6, we can build the set $\mathcal{L}$ in $\mathcal{O}(m)$ steps from the eMS array. Recall that $\mathcal{L}$ contains those indices $i \in [0..m)$ such that $P[i..i + \mathsf{eMS}[i].\mathsf{len})$ is a maximal match that occurs only once in T.

Algorithm 2 MSMatch($P[i], q, \mathsf{eMS}[i+1].\mathsf{pos}, \mathsf{eMS}[i+1].\mathsf{len}, lcp_p, lcp_s$).

```
1  pos ← eMS[i + 1].pos − 1, len ← eMS[i + 1].len + 1
2  c ← BWT.rank_{P[i]}(q)
3  if BWT[q − 1] = P[i] then
       lcp_p ← lcp_p + 1
4  else
5      q_p ← BWT.select_{P[i]}(c)
6      lcp_p ← min(lcp_p, LCE(eMS[i + 1].pos, SA[q_p])) + 1
7  if BWT[q + 1] = P[i] then
       lcp_s ← lcp_s + 1
8  else
9      q_s ← BWT.select_{P[i]}(c + 2)
10     lcp_s ← min(lcp_s, LCE(eMS[i + 1].pos, SA[q_s])) + 1
11 slen ← max(lcp_p, lcp_s)
12 return (pos, len, slen), lcp_p, lcp_s
```

Now we have to search those indices in $\mathcal{L}$ that are also unique in P. A simple algorithm is to build both the LCP and ISA array of P, and then check for each $i \in \mathcal{L}$ if both $\mathsf{LCP}[\mathsf{ISA}[i]]$ and $\mathsf{LCP}[\mathsf{ISA}[i]+1]$ (or only $\mathsf{LCP}[\mathsf{ISA}[i]]$ if $\mathsf{ISA}[i] = m$) are smaller than $\mathsf{eMS}[i].\mathsf{len}$, i.e. the same property that we use to check the uniqueness in T. Both structures can be build in $\mathcal{O}(m)$ time. The overall time is $\mathcal{O}(m(t_{\mathsf{LF}} + t_{\mathsf{LCE}} + t_{\mathsf{pred}}) + m + m)$, which collapses to $\mathcal{O}(m(t_{\mathsf{LF}} + t_{\mathsf{LCE}} + t_{\mathsf{pred}}))$. ◀

Note that both g and t_{LCE} depends on the grammar scheme chosen. In fact, if exists a data structure of size λ that supports LCE queries on a text T, then we can still compute MUMs in $\mathcal{O}(r + \lambda)$ space and $\mathcal{O}(m \cdot (t_{\mathsf{LF}} + t_{\mathsf{LCE}} + t_{\mathsf{pred}}))$ time, with t_{LCE} that depends on the data structure used.

4.2 Computing MUMs from eMS

Here we present a different approach to compute the MUMs from the eMS from the one in Theorem 13, that is of more practical use, and that does not require sorting the suffixes of P. We summarize this approach in Algorithm 4.

Let $\mathcal{L}$ be the set of indexes $i \in [0..m)$ such that $P[i..\mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len})$ is a maximal and unique match in T. By Lemmas 5 and 6, we can check in constant time if an index i belongs to $\mathcal{L}$. Note that building $\mathcal{L}$ (lines 3-4) can be also executed in streaming while computing the eMS array (for simplicity of exposition of the algorithms we have separated the procedures). Observe that a match $P[i..i + \mathsf{eMS}[i].\mathsf{len})$ such that $i \in \mathcal{L}$ is a MUM if and only if it is not fully contained into another candidate, i.e. it does not exist $j \in \mathcal{L} \setminus \{i\}$ such that (i) $\mathsf{eMS}[j].\mathsf{pos} \leq \mathsf{eMS}[i].\mathsf{pos}$ and (ii) $\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len} \leq \mathsf{eMS}[j].\mathsf{pos} + \mathsf{eMS}[j].\mathsf{len}$ (Theorem 9). Hence, we sort the elements in $\mathcal{L}$ with respect to the position in T, and starting from $\mathcal{L}[0]$, we compare every entry with the following and if both factors are not contained into the other, we store in the set MUMs the one with the smallest starting position and keep track of the other one, otherwise we simply discard the one that is repeated and continue with the following iteration.

Algorithm 3 MSMismatch$(P[i], q, \mathsf{eMS}[i+1].\mathsf{pos}, \mathsf{eMS}[i+1].\mathsf{len}, lcp_p, lcp_s)$.

$c \leftarrow \mathsf{BWT.rank}_{P[i]}(q)$
$q_p \leftarrow \mathsf{BWT.select}_{P[i]}(c)$
$q_s \leftarrow \mathsf{BWT.select}_{P[i]}(c+1)$
if $q_p = q - 1$ **then**
  $lcp'_p \leftarrow lcp_p$
else
  $lcp'_p \leftarrow \mathsf{min}(\mathsf{eMS}[i+1].\mathsf{len}, \mathsf{LCE}(\mathsf{eMS}[i+1].\mathsf{pos}, \mathsf{SA}[q_p]))$
if $q_s = q + 1$ **then**
  $lcp'_s \leftarrow lcp_s$
else
  $lcp'_s \leftarrow \mathsf{min}(\mathsf{eMS}[i+1].\mathsf{len}, \mathsf{LCE}(\mathsf{eMS}[i+1].\mathsf{pos}, \mathsf{SA}[q_s]))$
if $lcp'_p \leq lcp'_s$ **then**
  $\mathsf{pos} \leftarrow \mathsf{SA}[q_s] - 1, \mathsf{len} \leftarrow lcp'_s + 1, lcp_p \leftarrow lcp'_p + 1$
  $q'_s \leftarrow \mathsf{BWT.select}_{P[i]}(c+2)$
  if $q'_s = q_s + 1$ **then**
    $lcp_s \leftarrow \mathsf{min}(\mathsf{len}, \mathsf{LCP}[q_s + 1] + 1)$
  else
    $lcp_s \leftarrow \mathsf{min}(\mathsf{len}, \mathsf{LCE}(\mathsf{SA}[q_s], \mathsf{SA}[q'_s]) + 1)$
  $q \leftarrow q_s$
else
  $\mathsf{pos} \leftarrow \mathsf{SA}[q_p] - 1, \mathsf{len} \leftarrow lcp_p, lcp_s \leftarrow lcp'_s + 1$
  $q'_p \leftarrow \mathsf{BWT.select}_{P[i]}(c-1)$
  if $q'_p = q_p - 1$ **then**
    $lcp_p \leftarrow \mathsf{min}(\mathsf{len}, \mathsf{LCP}[q_p] + 1)$
  else
    $lcp_p \leftarrow \mathsf{min}(\mathsf{len}, \mathsf{LCE}(\mathsf{SA}[q_p], \mathsf{SA}[q'_p]) + 1)$
  $q \leftarrow q_p$
$\mathsf{slen} \leftarrow \mathsf{max}(lcp_p, lcp_s)$
return $(\mathsf{pos}, \mathsf{len}, \mathsf{slen}), lcp_p, lcp_s$

To handle the special case when two candidates $i \neq j \in \mathcal{L}$ are such that $T[\mathsf{eMS}[i].\mathsf{pos}..\mathsf{eMS}[i].\mathsf{pos} + \mathsf{eMS}[i].\mathsf{len}) = T[\mathsf{eMS}[j].\mathsf{pos}..\mathsf{eMS}[j].\mathsf{pos} + \mathsf{eMS}[j].\mathsf{len})$, we further keep track whether the current maximal match is unique. This final procedure, excluding the building time for $\mathcal{L}$ that is done in streaming, takes $\mathcal{O}(|\mathcal{L}| \log |\mathcal{L}|)$ time, since the sorting of the indexes in $\mathcal{L}$ dominates the overall cost.

5 Experimental results

We implemented our algorithm for computing MUMs and measured its performances on real biological datasets. We performed the experiments on a desktop computer equipped with 3.4 GHz Intel Core i7-6700 CPU, 8 MiB L3 cache. and 16 GiB of DDR4 main memory. The machine had no other significant CPU tasks running, and only a single thread of execution was used. The OS was Linux (Ubuntu 16.04, 64bit) running kernel 4.4.0. All programs were compiled using `gcc` version 8.1.0 with `-O3 -DNDEBUG -funroll-loops -msse4.2` options. We recorded the runtime and memory usage using the wall clock time, CPU time, and maximum resident set size from `/usr/bin/time`.

Algorithm 4 retrieveMUMs(eMS).

Input : Extended Matching Statistics $\mathsf{eMS}[0, m)$
Output : MUMs

```
1  L, MUMs ← ∅
2  for i ← 0 to m − 1 do
3      if (i = 0 or MS[i − 1].len ≤ MS[i].len) and MS[i].len > MS[i].slen then
4          L.add(i)
5  sortByPosition (L)
6  (p, ℓ) ← (eMS[L[0]].pos, eMS[L[0]].len)
7  unique ← true
8  for i ← 1 to |L| − 1 do
9      (p′, ℓ′) ← (eMS[L[i]].pos, eMS[L[i]].len)
10     if p = p′ then
11         if ℓ = ℓ′ then
12             unique ← false
13         else if ℓ < ℓ′ then
14             ℓ ← ℓ′
15             unique ← true
16     else if ℓ < ℓ′ + (p′ − p) then
17         if unique then
18             MUMs.add((p, ℓ))
19         (p, ℓ) ← (p′, ℓ′)
20         unique ← true
21 if unique then
22     MUMs.add((p, ℓ))
23 return MUMs
```

Setup

We compare our method (MUM-PHINDER) with MUMmer [18] (`mummer`). We tested two versions of `mummer`, v3.27 [13] (`mummer3`) and v4.0 [18] (`mummer4`). We executed `mummer` with the `-mum` flag to compute MUMs that are unique in both the text and the pattern, `-l 1` to report all MUMs of length at least 1, and `-n` to match only A,C,G,and T characters. We setup MUM-PHINDER to produce the same output as `mummer`. We did not test against Mauve [6] because the tool does not directly reports MUMs. We also did not consider algorithms that does not produces an index for the text that can be queried with different patterns without reconstructing the index, e.g. the algorithm described in Mäkinen et al. [16, Section 11.1.2]. The experiments that exceeded exceeded 16 GB of memory were omitted from further consideration.

Datasets

We evaluated our method using real-world datasets. We build our index for up to 512 haplotypes of human chromosome 19 from the 1000 Genomes Project [22] and up to 300,000 SARS-CoV2 genomes from EBI's COVID data portal [12]. We provide a complete list of

Table 1 Dataset used in the experiments. For each collection of datasets of the human chromosome 19 (`chr19`) dataset in Table 1a and for the SARSCoV2 (`sars-cov2`) dataset in Table 1b, we report the number of sequences (No. seqs), the length n in Megabytes (MB), and the ratio n/r, where r is the number of runs of the BWT for each number of sequences in a collection.

(a) Collections of chromosome 19.

No. seqs	n (MB)	n/r
1	59	1.92
2	118	3.79
4	236	7.47
8	473	14.78
16	946	29.19
32	1892	57.63
64	3784	113.49
128	7568	222.23
256	15,136	424.93
512	30,272	771.53

(b) Collections of SARS-CoV2 genomes.

No. seqs	n (MB)	n/r
1562	46	459.57
3125	93	515.42
6250	186	576.47
12,500	372	622.92
25,000	744	704.73
50,000	1490	848.29
100,000	2983	1060.07
200,000	5965	1146.24
300,000	8947	1218.82

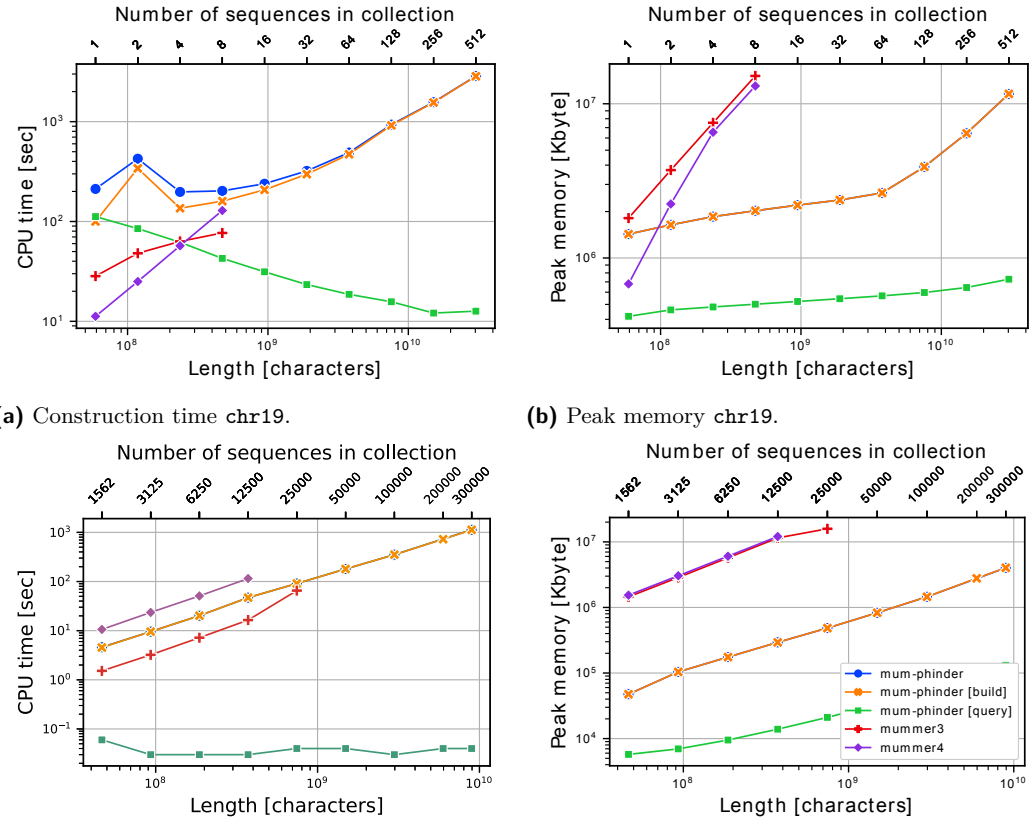
accession numbers in the repository. We divide the sequences into 11 collections of 1, 2, 3, 4, 8, 16, 32, 64, 128, 256, 512 chromosomes 19 (`chr19`) and 9 collections of 1,562, 3,125, 6,250, 1250,00, 25,000, 50,000, 100,000, 200,000, 300,000 genomes of SARS-CoV2 (`sars-cov2`). In both datasets, each collection is a superset of the previous one. In Table 1 we report the length n of each collection and the ratio n/r, where r is the number of runs of the BWT.

Furthermore, for querying the datasets, we used the first haplotype of chromosome 19 of the sample NA21144 from the 1000 Genomes Project, and the genome with accession number MZ477765 from EBI's COVID data portal [12].

Results

In Figure 2 we show the construction and query time and space for MUM-PHINDER and `mummer`. Since `mummer` is not able to decouple the construction of the suffix tree from the query, for our method we report the sum of the running times for construction and query, and the maximum resident set size of the two steps. We observe that on `chr19` `mummer3` is up to 9 times faster than MUM-PHINDER, while using up to 8 times more memory, while `mummer4` is up to 19 times faster than MUM-PHINDER, while using up to 7 times more memory. However both `mummer3` and `mummer4` cannot process more than 8 haplotypes of `chr19` due to memory limitations. MUM-PHINDER was able to build the index and query in 48 minutes for 512 haplotypes of `chr19` while using less than 11.5 GB of RAM. On `sars-cov2`, `mummer3` is up to 6.5 times faster than MUM-PHINDER, while using up to 24 times more memory, while `mummer4` is up to 1.2 times slower than MUM-PHINDER, while using up to 25 times more memory. `mummer3` was not able to process more than 25,000 genomes while `mummer4` were not able to query mote than 12,500 genomes of `sars-cov2` due to memory limitations.

In Figure 2 we also show the construction time and space for MUM-PHINDER. We observe that the construction time grows with the number of sequences in the dataset, however the query time decreases while increasing the number of sequences in the index with a 9x speedup when moving from 1 to 512 haplotypes of `chr19`. A similar phenomenon is observed in [3] and it is attributed to the increase number of match cases (Algorithm 2) while increasing the number of sequences in the index. From our profiling (data not shown) the



(a) Construction time `chr19`.

(b) Peak memory `chr19`.

(c) Construction time `sars-cov2`.

(d) Peak memory `sars-cov2`.

Figure 2 Human chromosome 19 and SARS-CoV2 genomes dataset construction CPU time and peak memory usage. We compare MUM-PHINDER with `mummer3` and `mummer4`. For MUM-PHINDER we report a breakdown of the construction (build) and query time and space. Note that for MUM-PHINDER we consider as time the sum of construction and query time, while for memory we consider the maximum between construction and query memory.

more time-demanding part of the queries are LCE queries, which are not performed in case of matches. This observation also motivates the increase in the control logic of Algorithm 3 to limit the number of LCE queries to the essential ones.

References

1 Joel Armstrong, Glenn Hickey, Mark Diekhans, Ian T. Fiddes, Adam M. Novak, Alden Deran, Qi Fang, Duo Xie, Shaohong Feng, et al. Progressive Cactus is a multiple-genome aligner for the thousand-genome era. *Nature*, 587(7833):246–251, 2020.

2 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.

3 Christina Boucher, Travis Gagie, Tomohiro I, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. PHONI: streamed matching statistics with multi-genome references. In *Proceedings of 2021 Data Compression Conference DCC*, pages 193–202. IEEE, 2021.

4 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. Technical report, DIGITAL SRC RESEARCH REPORT, 1994.

5 Aaron C. E. Darling, Bob Mau, Frederick R. Blattner, and Nicole T. Perna. Mauve: multiple alignment of conserved genomic sequence with rearrangements. *Genome Res.*, 14(7):1394–1403, 2004.

6 Aaron E. Darling, Bob Mau, and Nicole T. Perna. progressiveMauve: multiple genome alignment with gene gain, loss and rearrangement. *PLoS One*, 5(6):e11147, 2010.

7 Marc Deloger, Meriem El Karoui, and Marie-Agnès Petit. A genomic distance based on MUM indicates discontinuity between most bacterial species and genera. *J. Bacteriol.*, 191(1):91–99, 2009.

8 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *In Proceedings 41st annual Symposium on Foundations of Computer ScienceFOCS*, pages 390–398. IEEE Computer Society, 2000.

9 Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, Louisa Seelbach Benkner, and Yoshimasa Takabatake. Practical Random Access to SLP-Compressed Texts. In *Proceedings of the 27th International Symposium on String Processing and Information Retrieval (SPIRE 2020)*, volume 12303 of *LNCS*, pages 221–231. Springer, 2020.

10 Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Rescaling RePair with Rsync. In *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019*, volume 11811 of *LNCS*, pages 35–44. Springer, 2019. doi:10.1007/978-3-030-32686-9_3.

11 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.

12 Peter W. Harrison, Rodrigo Lopez, Nadim Rahman, Stefan Gutnick Allen, Raheela Aslam, Nicola Buso, Carla Cummins, Yasmin Fathy, Eloy Felix, et al. The COVID-19 Data Portal: accelerating SARS-CoV-2 and COVID-19 research through rapid open access data sharing. *Nucleic Acids Research*, 49(W1):W619–W623, 2021.

13 Stefan Kurtz, Adam Phillippy, Arthur L. Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biol.*, 5(2):R12, 2004.

14 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

15 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

16 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. *Genome-scale algorithm design*. Cambridge University Press, 2015.

17 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

18 Guillaume Marçais, Arthur L. Delcher, Adam M. Phillippy, Rachel Coston, Steven L. Salzberg, and Aleksey Zimin. MUMmer4: A fast and versatile genome alignment system. *PLoS Comput. Biol.*, 14(1):e1005944, 2018.

19 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V. Bzikadze, Alla Mikheenko, Mitchell R. Vollger, Nicolas Altemose, Lev Uralsky, et al. The complete sequence of a human genome. *bioRxiv*, 2021.

20 Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A Pangenomic Index for Finding Maximal Exact Matches. *J. Comput. Biol.*, January 2022.

21 Jouni Sirén, Jean Monlong, Xian Chang, Adam M. Novak, Jordan M. Eizenga, Charles Markello, Jonas A. Sibbesen, Glenn Hickey, Pi-Chuan Chang, et al. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science*, 374(6574):abg8871, 2021.

22 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, pages 68–74, 2015.

23 Kaiyuan Zhu, Welles Robinson, Alejandro A. Schäffer, Junyan Xu, Eytan Ruppin, A. Funda Ergun, Yuzhen Ye, and S. Cenk Sahinalp. Strain Level Microbial Detection and Quantification with Applications to Single Cell Metagenomics. *bioRxiv*, page 2020.06.12.149245, 2020.

Automatic Reformulations for Convex Mixed-Integer Nonlinear Optimization: Perspective and Separability

Meenarli Sharma[1]
Institute of Mathematics, University of Augsburg, Germany

Ashutosh Mahajan
Industrial Engineering and Operations Research, Indian Institute of Technology Bombay, India

Abstract

Tight reformulations of combinatorial optimization problems like Convex Mixed-Integer Nonlinear Programs (MINLPs) enable one to solve these problems faster by obtaining tight bounds on optimal value. We consider two techniques for reformulation: perspective reformulation and separability detection. We develop routines for automatic detection of problem structures suitable for these reformulations, and implement new extensions. Since detecting all "on-off" sets for perspective reformulation in a problem can be as hard as solving the original problem, we develop heuristic methods to automatically identify them. The LP/NLP branch-and-bound method is strengthened via "perspective cuts" derived from these automatic routines. We also provide methods to generate tight perspective cuts at different nodes in the branch-and-bound tree. The second structure, i.e., separability of nonlinear functions, is detected by means of the computational graph of the function. Our routines have been implemented in the open-source Minotaur solver for general convex MINLPs. Computational results show an improvement of up to 45% in the solution time and the size of the branch-and-bound tree for convex instances from benchmark library MINLPLib. On instances where reformulation using function separability induces structures that are amenable to perspective reformulation, we observe an improvement of up to 88% in the solution time.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Mathematics of computing → Solvers; Applied computing → Operations research

Keywords and phrases Convex MINLP, perspective reformulation, branch-and-bound, outer approximation, function separability

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.23

Related Version *Full Version*: `http://www.optimization-online.org/DB_HTML/2022/04/8888.html`

1 Introduction

We study convex MINLPs that are optimization problems of the form

$$\left.\begin{array}{ll} \underset{x}{\text{minimize}} & c^T x \\ \text{subject to} & g_i(x) \leq 0,\ i = 1, \ldots, m, \\ & x_i \in \mathbb{Z},\ i \in \mathcal{I}. \end{array}\right\} \tag{P}$$

Here, variables with indices in set $\mathcal{I}$ are restricted to take only integer values and constraint functions $g_i : \mathbb{R}^n \to \mathbb{R},\ i = 1, \ldots, m$ are convex and twice continuously differentiable. Convex MINLPs arise in a several real-world applications and are also solved as subproblems in nonconvex MINLPs [19, 22], and mixed-integer PDE constrained optimization problems [25].

[1] corresponding author

In a branch-and-bound framework for solving a convex MINLP (P), iteratively tightened relaxations of the problem are solved to obtain lower bounds on optimal objective value, Z^*. Since tight relaxations often give good bounds, one seeks to generate tight relaxations at different nodes in a tree-search. Reformulation of the problem is one of the ways to tighten relaxations. In this work, we consider two useful reformulations (i) Perspective Reformulation (PR) [9, 13] (ii) and a reformulation using the "separability" property of functions in nonlinear constraints.

It is shown in [9, 13] that, for some special disjunctive sets ("on-off" sets) convex hull description can be given in the space of the original variables using the perspective function. Problems to which PR can be applied occur in many applications, and they are shown to be solved better in terms of solution time and branch-and-bound tree size by using PR [2, 8, 13]. Although PR is useful, a bottleneck in its implementation is detecting the on-off sets in a given problem. Moreover, the reformulation involves a nonlinear constraint that can cause numerical difficulty due to possible division by zero. Depending on how this nonlinear constraint is handled, there are different ways to solve the reformulated problem [10, 12, 13]. We first introduce structures (in the form of collections of constraints) that indicate disjunctions suitable for PR, and then provide computationally economical ways to automatically detect these structures in a problem. More specifically, we present some new structures that imply "semi-continuous" variables, an important component in defining on-off sets. To obtain tight linear inequalities that outer-approximate the perspective reformulation, we propose novel line search approaches.

In the context of outer-approximation based algorithms, it is demonstrated in [17, 27] that, if a convex function is decomposed into its convex sub-expressions then outer-approximating these decomposed components separately gives better approximation of the original function than outer-approximating the original function directly. Exploiting separability in functions defining nonlinear convex constraints allows such a decomposition. The effectiveness of exploiting separability in specific models is further demonstrated in [14], where such algorithms are shown to require orders of magnitude fewer cuts to converge using separability. We implement routines to automatically detect separable functions using their "computational graphs" that are available in some solvers to store nonlinear functions.

All presented methods have been implemented within the open-source solver Minotaur[2] [20]. Our computational experiments show the improved performance of Minotaur on convex instances in MINLPLib [5] achieved by these reformulation techniques. To the best of our knowledge, PR has only recently been implemented in SCIP [3] for both convex and nonconvex problems. Also, a reformulation based on separability exists in the convex MINLP solver SHOT [18].

The rest of the paper is organized as follows. In Section 2 and Section 3, we present the perspective reformulation and the separability based reformulation, respectively, and their impact on the performance of a branch-and-cut algorithm in Minotaur. The combined effect of these reformulations is reported in Section 4. Section 5 presents our conclusions.

[2] Available at `http://github.com/minotaur-solver/minotaur`

2 Perspective Reformulation

A disjunctive set of the form (S)

$$\left\{(x,z) \in \mathbb{R}^n \times \{0,1\} \;\middle|\; \begin{array}{l} x \in \Gamma_0, \text{ if } z = 0 \\ x \in \Gamma_1, \text{ if } z = 1 \end{array} \right\}, \tag{S}$$

where Γ_0 is a singleton set and Γ_1 is a bounded convex set, is called an "on-off" set. The roles of $z = 0$ and $z = 1$ in (S) can be swapped without losing generality. The binary variable z "controls" variables x in the sense that when $z = 0$, x takes a fixed value $\hat{x} \in \mathbb{R}^n$, and $z = 1$ implies that x lies in a compact convex set. Such variables x are called semi-continuous variables [9, 10] and they appear in many real-world applications [11, 15, 29].

The convex hull of an "on-off" set can be represented using a function $\check{f}$ in the space of original variables [7]. Given a function $f(x) : \mathbb{R}^n \to \mathbb{R}$, $\check{f}(x,\lambda) : \mathbb{R}^{n+1} \to \mathbb{R}$ is defined as

$$\check{f}(x,\lambda) = \begin{cases} \lambda f\Big(\dfrac{x-(1-\lambda)\hat{x}}{\lambda}\Big), & \text{if } \lambda > 0, \\ 0, \quad \text{if } \lambda = 0, \\ \infty, \quad \text{otherwise}, \end{cases} \tag{PF}$$

where $\hat{x}$ is some fixed vector. It can be easily shown that if f is convex, then $\check{f}$ is also convex. When $\hat{x} = 0$, the function $\check{f}$ is well known as the perspective function of f. Two sets that conform to the form (S) are as follows.

1. This set referred to as (S_1), equals $\Gamma_0 \cup \Gamma_1$ with

$$\left.\begin{array}{l} \Gamma_0 := \{(x,z) \in \mathbb{R}^p \times \{0,1\} : x = \hat{x},\ z = 0\} \\ \Gamma_1 := \{(x,z) \in \mathbb{R}^p \times \{0,1\} : g_i(x) \le 0,\ A(x,z) \le a, z = 1\} \end{array}\right\} \tag{S_1}$$

2. This set, referred to as (S_2), is defined as $\Gamma_0 \cup \Gamma_1$, with

$$\left.\begin{array}{l} \Gamma_0 := \{(x,v,z) \in \mathbb{R}^{p+q} \times \{0,1\} : x = \hat{x}, d^T v \le 0, z = 0\}. \\ \Gamma_1 := \{(x,v,z) \in \mathbb{R}^{p+q} \times \{0,1\} : g_i(x) + d^T v \le 0, A(x,z) \le a, z = 1\}. \end{array}\right\} \tag{S_2}$$

   Even though (S_2) is not an on-off set by definition, its convex hull can still be described by a perspective function like an on-off set in the space of original variables.

In both sets $\hat{x}$ is a fixed vector, g_i is a convex nonlinear function, and Γ_1 is a compact convex set. In the first set Γ_0 is a singleton and in the second it is a halfspace. The polyhedral set defined by $A(x,z) \le a$ is compact (A and a are a matrix and a vector of the corresponding dimension) and contains $(\hat{x}, 0)$, enforcing the on-off relation between x and z. The convex hulls ($conv(.)$) of these sets (Lemmas 1–2) can be shown to lie in the space of the original variables x and z [13].

▶ **Lemma 1.** *$conv(S_1) = closure(\tilde{S}_1)$, where*

$$\tilde{S}_1 = \Big\{(x,z) \in \mathbb{R}^{p+1} : z g_i\Big(\frac{x-(1-z)\hat{x}}{z}\Big) \le 0, \\ A(x-(1-z)\hat{x}) \le az,\ 0 < z \le 1\Big\}.$$

▶ **Lemma 2.** *$conv(S_2) = closure(\tilde{S}_2)$, where*

$$\tilde{S}_2 = \Big\{(x,v,z) \in \mathbb{R}^{p+q+1} : z g_i\Big(\frac{x-(1-z)\hat{x}}{z}\Big) + d^T v \le 0, \\ A(x-(1-z)\hat{x}) \le az,\ 0 < z \le 1\Big\}.$$

In sets (S_1) and (S_2), x are semi-continuous variables. Given a convex MINLP, to find if x_i is a semi-continuous variable controlled by z, where $z \in \{0,1\}$, two MINLPs have to be solved to check if x_i can be fixed to $\hat{x}_i$. In these two MINLPs, z is fixed to 0 and the objective functions are $max\ x_i$ and $min\ x_i$, respectively (the rest remains same as the original problem). If the optimal value of these two MINLPs is equal to $\hat{x}_i$, then it implies $z = 0$ fixes x_i to $\hat{x}_i$. Since detecting semi-continuous variables in a given problem requires solving MINLPs (which can be as difficult as solving the original problem), there is a trade-off between the number of (S_1) and (S_2) sets detected and the time spent in detecting them. A less time-consuming alternative is to heuristically find collections of constraints during presolve, that indicate semi-continuous variables and binary variables controlling them. Some such collections ((C_1), (C_2), and (C_3)) are presented below, and our computational study show that these collections appear as small blocks in many optimization problems. The techniques to detect them are similar to probing for MILPs [24]. Henceforth, $\hat{x}$ indicates the fixed value that semi-continuous variables x takes when the corresponding binary variable has value 0.

1. Linear inequalities in at most two variables of the form,

$$\left.\begin{aligned} &l^1 z + l^0(1-z) \le x \le u^1 z + u^0(1-z), \\ &z \in \{0,1\},\ x \in \mathbb{R}, \end{aligned}\right\} \qquad (C_1)$$

where $l^0, l^1, u^0, u^1 \in \mathbb{R}$ and $l^j \le u^j, j = 0, 1$. If $l^0 = u^0$, then x is a semi-continuous variable controlled by z. Similarly, if $l^1 = u^1$, then $(1-z)$ controls x. A simple example of (C_1) that appears in many problems is, $lz \le x \le uz, z \in \{0,1\},\ x \in \mathbb{R}$.

2. Single constraint with an indicator

$$\left.\begin{aligned} &a^T x + d_1 z \le d_2, \\ &l \le x \le u, \\ &z \in \{0,1\}, \end{aligned}\right\} \qquad (C_2)$$

where d_1 and d_2 are scalars, and $l, u \in \mathbb{R}^p$ with $l_i \le u_i, \forall i$, with the additional property that if $a_i > 0$, then $l_i = 0$, and if $a_i < 0$, then $u_i = 0$.

a. If $d_2 = 0$ and $d_1 < 0$, every component of x is semi-continuous variable controlled by z such that $\hat{x} = 0$. If $d_2 = 0$ and $d_1 > 0$, $z = 1$ is infeasible and therefore, z can be fixed to 0.

b. When $d_1 = d_2$, x is semi-continuous variable controlled by $1-z$ and $\hat{x} = 0$. Additionally, if $d_1 < 0$, then $z = 0$ becomes infeasible and z can be fixed to 1. A simple example of this case is $\sum_{i=1}^{p} z_i \le 1, z \in \{0,1\}^p$ where any $(1 - z_i)$ controls all the other variables.

3. (C_1) or (C_2) with an extra equality constraint

$$\left.\begin{aligned} &d^T \overline{x} + d_3 \tilde{x} = d_4, \\ &\tilde{x} \in \mathbb{R},\ \overline{x} \in C_1 \text{ or } C_2, \end{aligned}\right\} \qquad (C_3)$$

where $d \in \mathbb{R}^p$, and d_3 and d_4 are any scalars. If $\overline{x}$ is controlled by z or $(1-z)$ ($\hat{\overline{x}}$ being the corresponding fixed value of $\overline{x}$), then so is $\tilde{x}$ if $\tilde{l} \le \dfrac{d_4 - d^T \hat{\overline{x}}}{d_3} \le \tilde{u}$, where $\tilde{l}$ and $\tilde{u}$ are lower and upper bounds respectively on $\tilde{x}$, otherwise, z can be fixed to 1 or 0, respectively.

Out of 374 convex MINLP instances in MINLPLib, 274 instances have at least one binary variable remaining after Minotaur's presolve routine. Table 9 in Appendix B reports the number of instances out of these 274 with above mentioned collections. Out of these 274 instance, 220 have at least one of these three collections, and set of these instances is referred to as TS_c, which is used for detecting sets amenable to PR. Details of instances in test set TS_c are presented in the Table 6 in Appendix A. Note that these are the instances that have semi-continuous variables, and some of them may not be suitable for the perspective reformulation.

2.1 Structures Amenable to Perspective Reformulation

Given the problem (P), following structures conform with sets of the form (S_1) or (S_2), and thus, are amenable to perspective reformulation.

1. A constraint in which all variables in the nonlinear function are semi-continuous, that is,

$$\left.\begin{aligned} &g_i(x) \leq 0, \\ &(x, z) \in \bar{C}, \end{aligned}\right\} \tag{PS_1}$$

   where $\bar{C}$ is a union of at least one of C_1 or C_2 or C_3. This structure conforms with (S_1).
2. A constraint in which all variables in only the nonlinear part of the function are semi-continuous. Let $\widetilde{g}_i$ and $\overline{g}_i$ denote nonlinear and linear parts of function g_i in disjoint set of variables $\tilde{x}$ and $\overline{x}$, respectively. If z exists in the constraint, it should be considered a part of $\tilde{g}_i$. This structure conforms with set (S_2).

$$\left.\begin{aligned} &\widetilde{g}_i(\tilde{x}) + \overline{g}_i(\overline{x}) \leq 0, \\ &(\tilde{x}, z) \in \bar{C}, \end{aligned}\right\} \tag{PS_2}$$

The PR amenable structure specified in [3] is of form (PS_2), with semi-continuous variables recognized by the constraints (C_1). A reformulation of the problem that results by replacing structures (PS_1) and (PS_2) by their convex hull description (as mentioned in Lemma 1 and Lemma 2, respectively) is referred to as perspective reformulation of the problem.

2.2 Detecting Structures (PS_1) and (PS_2)

Given a problem (P), we have a straightforward two-phase algorithm for detecting nonlinear constraints amenable to PR. In the first phase, the algorithm iterates through linear inequalities to find blocks of constraints (C_1) and (C_2). Then it iterates through all the linear equalities to detect (C_3). The outcome of the first phase is either a set of semi-continuous variables (and binary variables controlling them) or an indication that there is none. If there are semi-continuous variables, then in the second phase, the algorithm iterates through nonlinear constraints and checks if it has form required for (PS_1) or (PS_2). In case a nonlinear constraint conforms to either of the forms, it is declared amenable to PR.

Our computational results show that 104 instances (all mixed-binary nonlinear programs) in the test set TS_c have structures amenable to PR. We refer to the set of these 104 instances as TS_{pr} and Table 7 in Appendix A reports more details of these instances. Out of these, 103 instances have all PR amenable constraints of type (PS_1), and instance `synthes3` has one constraint each of type (PS_1) and (PS_2). Moreover, we found that all instances (except `synthes2` and `synthes3`) in TS_{pr} have all nonlinear constraints amenable to PR.

As this algorithm iterates through linear constraints for finding semi-continuous variables, it might take more time on instances with a large number of linear constraints. In our

experiments, the time taken to detect these structures (including detection of semi-continuous variables) in any instance in the set TS_{pr} is negligible (less than half a second).

2.3 Solving Perspective Reformulation

We solve the reformulated problem using perspective cuts in the LP/NLP based branch-and-bound method [23]. This method, also known as the QG algorithm, is based on a branch-and-cut framework and is a state-of-art method for convex MINLPs. It is implemented and practically enhanced in many MINLP solvers [1, 4, 16, 21, 26].

Perspective cuts (PCs) are outer-approximation cuts to constraints after PR is applied to them [13]. For the nonlinear constraint in structure (PS_1), the outer-approximation cut at (x', z') and $\left(\frac{x'}{z'}, 1\right)$ are the same and is given by

$$x^\top s + z\Big(g_i\Big(\frac{x'}{z'}\Big) + s\Big(\hat{x} - \frac{x'}{z'}\Big)\Big) \leq s^\top \hat{x},\ s \in \partial_x g_i\Big(\frac{x'}{z'}\Big). \tag{1}$$

Adding infinitely many PCs to the defining function in a PR amenable structure gives its convex hull. Also, note that all PCs pass through the point $(\hat{x}, 0)$. Similarly, a PC for a reformulated constraint in (PS_2), a perspective cut is given by

$$\tilde{x}^\top s + z\Big(\widetilde{g}_i\Big(\frac{\tilde{x}'}{z'}\Big) + \tilde{s}\Big(\hat{\tilde{x}} - \frac{\tilde{x}'}{z'}\Big)\Big) + \overline{g}_i(\overline{x}) \leq s^T \hat{\tilde{x}},\ s \in \partial_{\tilde{x}} \widetilde{g}_i\Big(\frac{x'}{z'}\Big). \tag{2}$$

In QG, cuts (gradient inequalities) are added at nodes where associated linear programs yield integer optimal solutions. Traditionally, the solution to the continuous relaxation of the root node, say (x^0, z^0), is used to create the initial linear relaxation by linearizing the nonlinear constraints at (x^0, z^0). Here, z^0 represents the vector of binary variables associated with semi-continuous variables appearing in the structures amenable to PR. These linearizations to the constraints active at (x^0, z^0) are supporting for P^c (the feasible region of the continuous relaxation of problem), but not necessarily for P^r (the feasible region of the continuous relaxation of the perspective reformulated problem). This scenario arises when some $z_i^0 \in (0, 1)$ satisfies the original nonlinear constraint but not the reformulated constraint. Moreover, this can also happen at other nodes in the branch-and-bound tree.

We found that in 68 instances in TS_{pr}, at least one reformulated nonlinear constraint is violated at (x^0, z^0) and 20 of them have more than 50% of the reformulated constraints violated. This observation motivated us to generate tight PCs for the reformulated problem at a point (x', z') that is not in P^r. We study the problem of generating perspective cuts at such a point (x', z') under the following two cases.

1. $(x', z') \in P^c$ and $(x', z') \notin P^r$: This happens when (x^0, z^0) does not lie in P^r.
2. $(x', z') \notin P^c$ and $(x', z') \notin P^r$: This can happen at nodes other than root node yielding fractional optimal solutions.

Given such a point $(x', z') \notin P^r$, we propose the following two methods that find another point (x'', z'') in P^r (or at least at the boundary of the violated constraint) such that the linearizations at (x'', z'') cut off (x', z').

1. SimLS Method: This is a simple line search that considers each violated constraint and search for a point that satisfies the reformulated constraint at equality. That is, given

$$z'_i g_i\Big(\frac{x' - (1 - z'_i)\hat{x}}{z'_i}\Big) > 0,$$

this method finds a point (x'', z'') such that ${z_i}'' g_i\left(\frac{x''-(1-{z_i}'')\hat{x}}{{z_i}''}\right) = 0$. Given the point $(x', z') \in P^c$, if $(x', 1) \in P^r$, then (x'', z'') is such that $x'' = x'$ and $z_i'' = (1-\lambda)z_i' + \lambda$ for some $\lambda \in (0, 1]$.
Also, if $(\hat{x}, 1) \in P^c$ (and thus, in P^r), then for every $(x', z') \in P^c$, $(x', 1) \in P^c$ (and thus, in P^r). Verifying $(\hat{x}, 1) \in P^c$ amounts to evaluating whether the nonlinear constraint satisfies at $(x', 1)$. Also, for the structure (PS_1), if the associated binary variable does not exist in the defining nonlinear constraint, then $(\hat{x}, 1) \in P^c$.
We found that in 98 instances in TS_{pr}, $(\hat{x}, 1)$ belongs to P^c for all the PR amenable constraints and in 50 of them, the binary variables controlling the semi-continuous variables do not appear in the constraint functions. The 6 instances in which this condition is not satisfied for any of the PR amenable constraints are of the type `clay*`.

2. CenLS Method: This method performs a line search between the given point and (x^C, z^C) (an approximation of the center of P^c) to obtain a point (x'', z'') at the boundary. The point (x^C, z^C) is obtained by solving the following nonlinear problem (NLPI), in which all the nonlinear inequalities in the original problem are modified using an auxiliary variable, ν, which also forms the objective of (NLPI). All the linear constraints remain unchanged. Let the optimal solution of (NLPI) be $(\tilde{\nu}, \tilde{x}, \tilde{z})$. If $\tilde{\nu} < 0$, then we set $(x^C, z^C) = (\tilde{x}, \tilde{z})$. If $\tilde{\nu} = 0$, then no point in the feasible region of the original problem exists at which all the nonlinear constraints are inactive. In this case, we terminate the method. If (NLPI) is unbounded, then we add ν to the linear inequalities in the same way as the nonlinear constraints and then re-solve.

$$\left.\begin{array}{lll} \underset{x,\nu}{\text{minimize}} & \nu & \\ \text{subject to} & g_i(x) \leq \ \nu, & i \in M, \\ & \nu \leq 0. & \end{array}\right\} \tag{NLPI}$$

If (x^C, z^C) is obtained, then it lies in P^r.

The following two sections present the computational experiments that compare the default implementation of QG in Minotaur, referred to as *qg*, to *qg* with PCs on overall solution time and size of the tree (in terms of the number of nodes processed). All the computational experiments have been carried out on a system with two 64-bit Intel(R) Xeon(R) E5-2670 v2, 2.50GHz CPUs having 10 cores each and sharing 128GB RAM. Our schemes are available in the development version of Minotaur[3]. All codes are complied with GCC-4.9.2 compiler. IPOPT-3.12 with MA27 linear-systems solver is used as the NLP solver. CPLEX-12.8 has been used as the LP solver. We have set a time limit of one hour for all our experiments and reported all the solution times in seconds.

2.3.1 Adding Perspective Cuts at Root Node

First we add PCs only at the root node with the following three settings.

1. *root_reg*: adds PCs to PR amenable constraints violated at (x^0, z^0).
2. *root_cenls*: adds PCs to violated PR amenable using CenLS method in addition to PCs from *root_reg*.
3. *root_bothls*: adds additional PCs using SimLS method, wherever applicable, in addition to PCs from *root_reg* and *root_cenls*.

[3] Available at `http://github.com/minotaur-solver/minotaur`

In these experiments, we commonly add the following cuts:

1. PCs to PR constraints at corresponding points $(\hat{x}, 0)$, where $z = 0$ implies $x = \hat{x}$.
2. If the perspective reformulated constraint is inactive at (x^0, z^0) and $(x^0, 1)$ does not lie in P^r, then we find a point on the boundary by moving along the direction $-e_z$ (which is always feasible), a vector whose components associated with z are -1 and the rest are 0.

Table 1 and Table 2 show a comparison of default *qg* and *qg* with settings $s \in \{$*root_reg, root_cenls, root_bothls*$\}$ for instances in test set TS_{pr}. These results show the distribution of performance across instances with varying difficulty. Each row corresponds to an experimental setting (s). Each row in Table 1 (Top) corresponds to the results of instances solved by *qg* and *qg* with setting s, and in Table 1 (Bottom), Table 2, it corresponds to instances that are solved by both, but where at least one of the methods takes more than 10, 100, and 500 seconds, respectively. The first column under the headings "time" and "nodes" shows the shifted geometric mean (SGM) of these measures reported by the reference solver (*qg* in this case) for the instances solved by both. The second column under these headings show the relative SGM ("rel.") under the setting s for the same instances. The relative SGM of a measure is computed as the ratio of the SGM value of the proposed scheme (here, *qg* under setting s) to the SGM value of the reference solver (*qg*). If this ratio, say r, is less than one, it implies that the proposed solver has performed better than the reference solver. More specifically, the proposed solver has shown an improvement over the reference solver with a factor $(1 - r)$ on the considered performance measure. One instance (`rsyn0830m04m`) on which *qg* reached the time limit took 56.22s with the setting *root_reg*, 53.93s with *root_cenls*, and 47.17s with *root_bothls*.

Table 1 (Top) Comparison of *qg* and *qg* with setting s on 103 instances that are solved by both the methods. (Bottom) Performance on 26 instances for *qg* and *qg* with first two settings, and 25 instances with *root_bothls* that are solved by both methods, but at least one method took more than 10 seconds.

	time		nodes	
setting (s)	*qg*	rel.	*qg*	rel.
root_reg	8.60	0.67	505.44	0.69
root_cenls	8.60	0.63	505.44	0.69
root_bothls	8.60	0.57	505.44	0.59

	time		nodes	
setting (s)	*qg*	rel.	*qg*	rel.
root_reg	67.23	0.48	14956.08	0.41
root_cenls	67.23	0.43	14956.08	0.41
root_bothls	72.39	0.34	17005.06	0.29

Table 2 (Top) Comparison of *qg* and *qg* with setting s on 9 instances that are solved by both the methods but at least one method took more than 100 seconds. (Bottom) Similar comparison on 2 instances that are solved by both the methods, but at least one method took more than 500 seconds.

time		nodes	
qg	rel.	*qg*	rel.
294.93	0.28	55751.27	0.24
294.93	0.24	55751.27	0.26
294.93	0.19	55751.27	0.19

time		nodes	
qg	rel.	*qg*	rel.
1249.47	0.10	595448.0	0.09
1249.47	0.10	595448.0	0.09
1249.47	0.05	595448.0	0.04

Our computational results show improvements in both the considered measures under all three settings. The highest improvement is reported by *qg* with *root_bothls*. Overall, it improved the solution time and tree size by about 43.19% and 41.45%, respectively. Even higher improvements (about 81% and 95%, respectively) are observed for instances in with default *qg* took more than 100 seconds and 500 seconds, respectively.

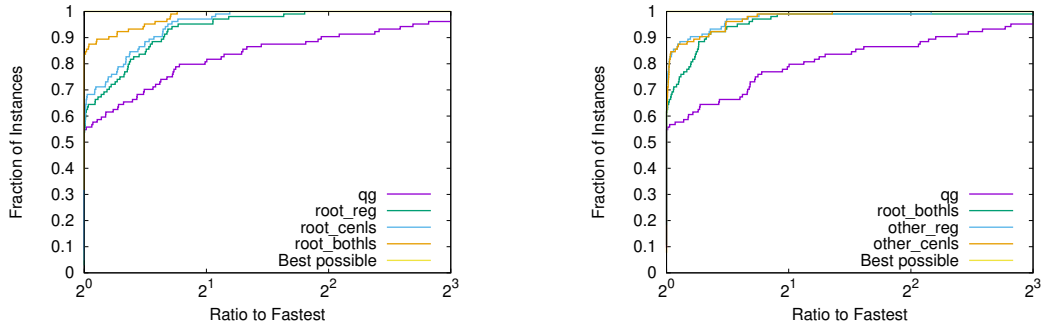


Figure 1 Performance profiles comparing solution times of *qg* and *qg* with *root_reg*, *root_cenls*, *root_bothls* (on left), and *qg* and *qg* with *root_bothls*, *other_reg*, *other_cenls* (on right).

Furthermore, we use performance profiles [6] that graphically demonstrate the relative performance of different solvers for a particular performance measure over a given set of instances.

Figure 1 shows the performance profiles of *qg* and *qg* with settings *root_reg*, *root_cenls*, *root_bothls* using the solution times of the instances in test set TS_{pr}. It shows that on nearly 85% of these instances, *root_cenls* is not slower than the rest and it solved all instances within 2 times of the best solvers among the considered ones. On the other hand, on 20% of the instances *qg* took more than double the time taken by the fastest method.

2.3.2 Adding Perspective Cuts at Other Nodes

Next, we generate PCs at other nodes yielding integer feasible solutions in addition to the root node. The nodes that we have selected for generating perspective cuts are the same as in default *qg* (the ones yielding integer optimal LP solution). But using the fixed-NLP solution as in QG algorithm may not produce tight inequalities for the reformulated problem for the same reason as mentioned for the case of the root node. Here, we employ CenLS method for finding points for generating tighter perspective cuts. Let (x', z') be an integer solution of the fixed-NLP at any node. If the fixed-NLP is infeasible, (x', z') is a solution to the feasibility problem. When the fixed-NLP is optimal, a reformulated constraint is always feasible. However, some constraints could be inactive. In the test set TS_{pr}, in 6 instances, at least one reformulated nonlinear constraint is violated at some node yielding an integer optimal solution. In 10 instances, at least one reformulated constraint is inactive at the fixed-NLP optimal solution. Thus, keeping the best setting at the root node, following computational experiments have been performed for adding PCs at integer optimal nodes.

- *other_reg*: This setting generates perspective cuts to every nonlinear constraint in the reformulated problem at (x', z') if $z' \neq 0$.
- *other_cenls*: This setting employs CenLS method for generating perspective cuts.

We have experimented *other_cenls* method with and without adding constraints for inactive perspective amenable constraints in the same manner as in the root node. We found better results by not adding additional constraints for inactive constraints and thus report the same. Table 3 and Table 4 summarize the results of *qg* with these settings in comparison to the default *qg* on instances in the test set TS_{pr}.

Our computational results show improvements in both the considered measures under all the three settings. The highest improvement is reported by *qg* with *other_cenls*. Overall, it improved the solution time and the tree size by about 47.40% and 44.62%, respectively, but even higher improvements (around 82% and 95% for both the measures) are observed

Table 3 (Top) Comparison of *qg* and *qg* with setting *s* on 103 instances that are solved by both the methods. (Bottom) Performance on instances (25 for *qg* and *qg* with first two settings, and 26 with *other_cenls*) that are solved by both but at least one method took more than 10 seconds.

	time		nodes	
setting (*s*)	*qg*	rel.	*qg*	rel.
root_bothls	8.60	0.57	505.44	0.59
other_reg	8.60	0.53	505.44	0.56
other_cenls	8.60	0.53	505.44	0.55

	time		nodes	
setting (*s*)	*qg*	rel.	*qg*	rel.
root_bothls	72.39	0.34	17005.06	0.29
other_reg	72.39	0.30	17005.06	0.26
other_cenls	67.61	0.31	15645.84	0.27

Table 4 (Top) Comparison of *qg* and *qg* with setting *s* on 9 instances that are solved by both the methods but at least one method took more than 100 seconds. (Bottom) Similar comparison on 2 instances that are solved by both the methods, but at least one method took more than 500 seconds.

time		nodes	
qg	rel.	*qg*	rel.
294.93	0.19	55751.27	0.19
294.93	0.18	55751.27	0.17
294.93	0.19	55751.27	0.17

time		nodes	
qg	rel.	*qg*	rel.
1249.47	0.05	595448.0	0.04
1249.47	0.06	595448.0	0.05
1249.47	0.06	595448.0	0.05

for instances in with default *qg* took more than 100 seconds and 500 seconds, respectively. Figure 1 shows the performance profiles of *qg* and *qg* with settings *root_bothls*, *other_reg*, *other_cenls* using the solution times of the instances in test set TS_{pr}.

One can also add the perspective cuts at the fractional nodes in the tree. However, since we compare with traditional QG, we limit our PR related computational experiments to the root node and the nodes yielding integer optimal solutions only.

3 Reformulation Based on Function Separability

A function $f : \mathbb{R}^n \to \mathbb{R}$ is called "group separable" or separable if there exist functions $f^i : \mathbb{R}^{n_i} \to \mathbb{R},\ i = 1, \ldots, m$, such that

$$f(x) = \sum_{i=1}^{m} f^i(x^i), \tag{3}$$

where f^i and f^j for $i \neq j$ have no variables in common [28]. That is, f can be written as a sum of functions with a disjoint set of variables. A function is said to be fully separable if every f^i is a univariate function and partially separable if some f^i are not univariate functions.

▶ **Proposition 3.** *If f as defined in (3) is convex, then every $f^i(x^i), i = 1, \ldots, m$ is also convex.*

Given a nonlinear separable constraint of the form $\sum_{i=1}^{m} f^i(x^i) \leq b$, where $b \in \mathbb{R}$ is a scalar, using function separability it can be reformulated as

$$\left.\begin{array}{l} \sum_{i=1}^{m} \tilde{\gamma}_i \leq b, \\ f^i(x^i) \leq \tilde{\gamma}_i,\ i = 1, \ldots, m, \\ \tilde{\gamma}_i \in \mathbb{R}, i = 1, \ldots, m. \end{array}\right\} \tag{SepCon}$$

Proposition 3 ensures that the reformulated problem is also a convex MINLP.

To detect separability of a nonlinear function, we use its computational graph (CG) that represents the function as a directed acyclic graph for computational purposes. A CG of a nonlinear function f is constructed as a combination of unary, binary, or other operations carried out on the input variables, constants, and intermediate variables, which themselves are created using these operations.

A node in a CG represents either a variable, a constant, or an operation. An edge e_{ij} from node i to node j implies that i is a parent of j, or j is an operand of operation represented by i. A node with no child is called an independent (or leaf) node and it represents either a constant or a variable. Other nodes are called dependent nodes. A node that represents a binary operation has two child nodes. A node representing a unary operation has only one child. Let E_i^o and E_i^t denote the sets of edges originating from a node i and terminating at node i, respectively. Node i with $E_i^t = \emptyset$ is called the root node and is unique in a CG. If $E_i^o = \emptyset$ then i is a leaf node. For an edge e_{ij}, let N_{ij}^o and N_{ij}^t represent the origin and terminal nodes, respectively. In our implementation, a node representing a constant can have only one parent.

Computational Subgraph

We use a notion of computational subgraph (CSG) in finding separable parts in the CG of a nonlinear function. Let $G_f(C, E)$ be a CG of a nonlinear function f where C and E refer to the sets of nodes and edges in the CG, respectively. A graph $G_f^s(V, F)$ is called a subgraph of $G_f(C, E)$ if the following conditions hold.

1. $V \subseteq C$ and $F \subseteq E$.
2. For each $v \in V$, $E_v^o \in F$, and for each $e_{ij} \in F$, $N_{ij}^t \in V$ and $N_{ij}^o \in V$.
3. A node with no parent node in V should not represent operations $+$, $-$, or unary minus.
4. $G_f^s(V, F)$ is connected.

A subgraph can have more than one node with no parent. Every CG is also its CSG. We define "maximal subgraph" as a CSG that is not a part of any CSG other than the original CG. Figure 2 shows the CG of a separable function and its maximal subgraphs.

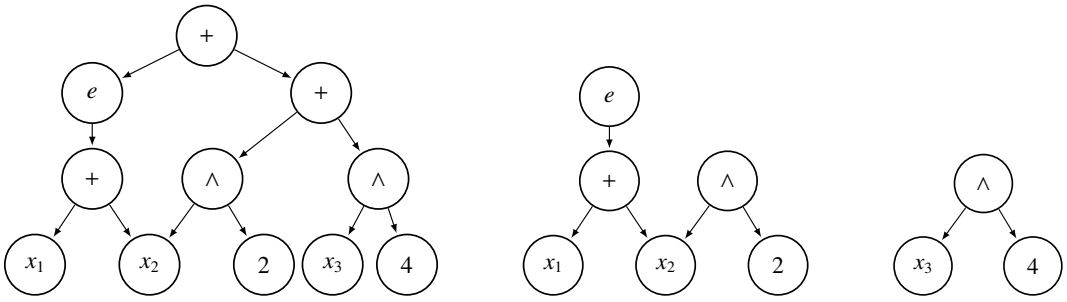


Figure 2 Computational graph of $f = e^{x_1+x_2} + {x_2}^2 + {x_3}^4$ (left) and its two maximal subgraphs.

Let f be a separable function (which cannot be further simplified symbolically) and let G_f be its CG.

▶ **Proposition 4.** *The number of maximal subgraphs in G_f is equal to the number of separable parts f and vice-versa.*

▶ **Proposition 5.** *f is not separable if and only if G_f has only one maximal subgraph.*

3.1 Detection of Function Separability

Given a nonlinear convex function $f : \mathbb{R}^n \to \mathbb{R}$ and its CG G_f, checking whether f is not separable is easier than checking otherwise. We start with employing simple rules to check if a function is not separable. If a function does not follow these rules, we use more extensive checking for separability.

Let r be the root node of G_f. Function f is not considered separable if any of the following conditions is met:

1. r represents a unary operation like $log, exp, sin, cos, \sqrt{(.)}$, etc., other than unary minus.
2. r represents the binary operation $\times$ and both its children represent a nonconstant expression.
3. r represents the binary operation $\div$ and its right child (divisor) represents a nonconstant expression.

If r represents the operation $\times$ with any of its children representing a constant, we compute the tree rooted at the nonconstant node and check again. Similarly, if r represents $\div$ with the right child representing a constant, we further analyze the graph rooted at its left child.

If a function does not satisfy the above conditions, G_f is iteratively searched for its maximal subgraphs. If there is only one maximal subgraph, the function is not separable. Otherwise, it is separable into as many parts as the number of maximal subgraphs.

3.2 Some Implementation Details

As our algorithm for detecting separability relies on the CG of the function, we prefer to express the function as explicitly as possible. Different separable expressions (in an explicit form) that can be identified by our algorithms include (1) $a \times (\sum_{i=1}^{m} f^i(x^i))$ (2) $\sum_{i=1}^{m} a_i \times f^i(x^i)$ (3) $\sum_{i=1}^{m} \frac{f^i(x^i)}{a_i}$, where $a,\ a_i,\quad b \in \mathbb{R}, \forall i = 1, \ldots, m$. However, if an expression can be simplified symbolically, e.g., $\sqrt{(x_1^2 + x_2^2)^2}$, then the proposed algorithm cannot recognize it as separable, even if it is technically a separable function.

In some instances of MINLPLib, we found that different separable constraints have common separable parts (f^i). Our implementation reuses variables corresponding to different separable parts in different constraint expressions, thus avoids creating an extra variable and an additional constraint. For example, the set of constraints of the form

$$a_1 f^1(x^1) + a_2 f^2(x^2) \leq b_1, \quad d_1 f^1(x^1) + d_2 f^3(x^3) \leq b_2,$$

is reformulated as

$$a_1\gamma_1 + a_2\gamma_2 \leq b_1, \quad d_1\gamma_1 + d_2\gamma_3 \leq b_2,$$
$$f^1(x^1) \leq \gamma_1, \quad f^2(x^2) \leq \gamma_2, \quad f^3(x^3) \leq \gamma_3.$$

In Minotaur, we carry out separability detection before the presolving step. We have found that 126 instances have at least one separable nonlinear function (either in constraint or objective) out of 374 convex instances in MINLPLib. Out of 126 such instances, 79 have separability only in the nonlinear objective function. In 45 of the remaining 47 instances, all the nonlinear constraints are separable, and 2 have 40% of the nonlinear constraints with separability property. Also, 108 out of these 126 instances have at least one integer constrained variable. These 108 instances constitute our test set, TS_{sep} for analyzing the impact of exploiting separability in QG. More details on these instances are provided in the Table 8 in Appendix A. Using the same performance measures as before, Table 5 reports a

comparison of default *qg* and *qg* with separability based reformulation (denoted *qgsep*) on instances in test set TS_{sep}. The time required by the proposed routine to detect function separability is a very small fraction of the total solution time in all instances of the test set. Overall, we achieve about 40% improvement in the solution time and the tree size; even

Table 5 Comparison of *qg* and *qgsep* on instances in TS_{sep}. The second column indicates the number of instances solved by both methods, where at least one method took more than the number of seconds indicated in the first column.

		time		nodes	
time	# of inst.	*qg*	rel.	*qg*	rel.
>= 0	76	13.68	0.60	700.04	0.50
>= 10	26	94.54	0.42	5902.23	0.32
>= 100	11	714.61	0.20	24311.92	0.28
>= 500	7	1900.22	0.12	54480.5	0.26

better improvements are obtained for difficult instances. Using this reformulation, 8 instances that reached the time limit earlier with *qg* could be solved. Figure 3 in Appendix B shows the performance profiles of *qg* and *qgsep* using the solution times of the instances in test set TS_{sep}.

4 Combined Effects of the Two Reformulations

Reformulation using separability sometimes results in structures amenable to PR. For example, consider the following uncapacitated facility location problem.

$$\left.\begin{array}{ll} \underset{x,\ z,\ \eta}{\text{minimize}} & \eta \\ \text{subject to} & \sum_{i\in\mathcal{F}} c_i z_i + \sum_{i\in\mathcal{F}, j\in\mathcal{C}} t_{ij} x_{ij}^2 \leq \eta, \\ & 0 \leq x_{ij} \leq z_i,\ i \in \mathcal{F},\ j \in \mathcal{C}, \\ & \sum_{i\in\mathcal{F}} x_{ij} = 1,\ j \in \mathcal{C}, \\ & x_{ij} \geq 0, z_i \in \{0,1\},\ i \in \mathcal{F},\ j \in \mathcal{C}. \end{array}\right\} \tag{UFL_1}$$

The function in the nonlinear constraint is separable and on reformulating (UFL_1), we get

$$\left.\begin{array}{ll} \underset{x,\ z,\ \eta}{\text{minimize}} & \eta \\ \text{subject to} & \sum_{i\in\mathcal{F}} c_i z_i + \sum_{i\in\mathcal{F}, j\in\mathcal{C}} t_{ij} \gamma_{ij} \leq \eta, \\ & x_{ij}^2 \leq \gamma_{ij}, i \in \mathcal{F},\ j \in \mathcal{C}, \\ & 0 \leq x_{ij} \leq z_i,\ i \in \mathcal{F},\ j \in \mathcal{C}, \\ & \sum_{i\in\mathcal{F}} x_{ij} = 1,\ j \in \mathcal{C}, \\ & x_{ij} \geq 0, z_i \in \{0,1\},\ i \in \mathcal{F},\ j \in \mathcal{C}. \end{array}\right\} \tag{UFL_2}$$

(UFL_2) now has structures of the form (PS_2) and thus, becomes amenable to PR.

Our results show that 26 instance in TS_{sep} become amenable to PR after separability based reformulation. These instances comprise test set TS_{ps} and are reported in Table 8 in Appendix A. Results on TS_{ps} using *qg*, *qgsep*, and *qgprsep* (*qg* with both separability and perspective reformulations) are reported in Table 10 and Table 11, respectively, in Appendix B. Overall, there is a significant improvement of about 88% in both the solution

time and the tree size, and 4 more instances that reached the time limit with even separability based reformulation could now be solved. Figure 3 in Appendix B shows the performance profiles of *qg*, *qgsep*, and *qgprsep* using the solution times of the instances in test set TS_{ps}.

5 Conclusions

Our study concludes that perspective reformulation and exploitation of separability of nonlinear constraint functions help generate better polyhedral-approximations of the feasible region. This is observed even when these reformulations are deployed using automatic routines that detect corresponding structures heuristically. We see improvement in both the solution time and the size of the tree in the branch-and-cut framework of the QG method on our test instances. The improvements are even higher for difficult instances and for those that became amenable to PR after separability based reformulation. We believe that such automatic routines can also reduce the effort required to model convex MINLPs.

References

1 K. Abhishek, S. Leyffer, and J. T. Linderoth. FilMINT: An outer-approximation-based solver for nonlinear mixed integer programs. Preprint ANL/MCS-P1374-0906, Mathematics and Computer Science Division, Argonne National Laboratory, 2006.

2 S. Aktürk, A. Atamtürk, and S. Gürel. A strong conic quadratic reformulation for machine-job assignment with controllable processing times. *Operations Research Letters*, 37:187–191, 2009.

3 Ksenia Bestuzheva, Ambros Gleixner, and Stefan Vigerske. A computational study of perspective cuts. *arXiv preprint arXiv:2103.09573*, 2021.

4 Pierre Bonami and Jon Lee. BONMIN user's manual. *Numer Math*, 4:1–32, 2007.

5 Michael R Bussieck, Arne Stolbjerg Drud, and Alexander Meeraus. MINLPLib - a collection of test models for mixed-integer nonlinear programming. *INFORMS Journal on Computing*, 15(1):114–119, 2003.

6 Elizabeth Dolan and Jorge Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.

7 A. Frangioni and C. Gentile. Perspective cuts for a class of convex 0-1 mixed integer programs. *Mathematical Programming*, 106:225–236, 2006.

8 Antonio Frangioni, Fabio Furini, and Claudio Gentile. Approximated perspective relaxations: a project and lift approach. *Computational Optimization and Applications*, 63(3):705–735, 2016.

9 Antonio Frangioni and Claudio Gentile. Perspective cuts for a class of convex 0–1 mixed integer programs. *Mathematical Programming*, 106(2):225–236, 2006.

10 Antonio Frangioni and Claudio Gentile. A computational comparison of reformulations of the perspective relaxation: SOCP vs. cutting planes. *Operations Research Letters*, 37(3):206–210, 2009.

11 Antonio Frangioni, Claudio Gentile, and Fabrizio Lacalandra. Solving unit commitment problems with general ramp constraints. *International Journal of Electrical Power & Energy Systems*, 30(5):316–326, 2008.

12 Kevin C Furman, Nicolas W Sawaya, and Ignacio E Grossmann. A computationally useful algebraic representation of nonlinear disjunctive convex sets using the perspective function. *Computational Optimization and Applications*, pages 1–26, 2020.

13 Oktay Günlük and Jeff Linderoth. Perspective reformulations of mixed integer nonlinear programs with indicator variables. *Mathematical programming*, 124(1-2):183–205, 2010.

14 Hassan Hijazi, Pierre Bonami, and Adam Ouorou. An outer-inner approximation for separable mixed-integer nonlinear programs. *INFORMS Journal on Computing*, 26(1):31–44, 2014.

15 Norbert J Jobst, Michael D Horniman, Cormac A Lucas, Gautam Mitra, et al. Computational aspects of alternative portfolio selection models in the presence of discrete asset choice constraints. *Quantitative finance*, 1(5):489–501, 2001.

16 KNITRO. *KNITRO Documentation*. Ziena Optimization., December 2012.

17 Jan Kronqvist, Andreas Lundell, and Tapio Westerlund. Reformulations for utilizing separability when solving convex MINLP problems. *Journal of Global Optimization*, 71(3):571–592, 2018.

18 Andreas Lundell, Jan Kronqvist, and Tapio Westerlund. The supporting hyperplane optimization toolkit for convex minlp. *Journal of Global Optimization*, pages 1–41, 2022.

19 Andreas Lundell and Tapio Westerlund. Solving global optimization problems using reformulations and signomial transformations. *Computers & Chemical Engineering*, 116:122–134, 2018.

20 Ashutosh Mahajan, Sven Leyffer, Jeff Linderoth, James Luedtke, and Todd Munson. Minotaur: A mixed-integer nonlinear optimization toolkit. *Mathematical Programming Computation*, pages 1–38, 2020.

21 Wendel Melo, Marcia Fampa, and Fernanda Raupp. An overview of minlp algorithms and their implementation in muriqui optimizer. *Annals of Operations Research*, 286(1):217–241, 2020.

22 Ivo Nowak, Norman Breitfeld, Eligius MT Hendrix, and Grégoire Njacheun-Njanzoua. Decomposition-based inner-and outer-refinement algorithms for global optimization. *Journal of Global Optimization*, 72(2):305–321, 2018.

23 Ignacio Quesada and Ignacio E Grossmann. An LP/NLP based branch and bound algorithm for convex MINLP optimization problems. *Computers & chemical engineering*, 16(10-11):937–947, 1992.

24 M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

25 Meenarli Sharma, Mirko Hahn, Sven Leyffer, Lars Ruthotto, and Bart van Bloemen Waanders. Inversion of convection–diffusion equation with discrete sources. *Optimization and Engineering*, pages 1–39, 2020.

26 Meenarli Sharma, Prashant Palkar, and Ashutosh Mahajan. Linearization and parallelization schemes for convex mixed-integer nonlinear optimization. *Computational Optimization and Applications*, pages 1–56, 2022.

27 Mohit Tawarmalani and Nikolaos V Sahinidis. A polyhedral branch-and-cut approach to global optimization. *Mathematical Programming*, 103(2):225–249, 2005.

28 Stephen J Wright, Robert D Nowak, and Mário AT Figueiredo. Sparse reconstruction by separable approximation. *IEEE Transactions on Signal Processing*, 57(7):2479–2493, 2009.

29 Juan M Zamora and Ignacio E Grossmann. A global MINLP optimization algorithm for the synthesis of heat exchanger networks with no stream splits. *Computers & Chemical Engineering*, 22(3):367–384, 1998.

A Description of Test Sets

Table 6 Description of instances with collections (C_1), (C_2), and (C_3). First column shows instance name and the entries $(bv, tv, fv, b0, b1, b01, v0, v1, v01)$ in the second column are: bv denotes the number of binary variables, tv indicates the total number of variables, fv reports the number of binary variables that are fixed as part of structure identification, $b0$ and $b1$ represent the number of binary variables z and $1-z$, respectively, controlling at least one other variable, $b01$ denotes number of binary variables z such that both z and $1-z$ control another variable, $v0$ and $v1$ report the number of variables controlled by a binary variable z and $1-z$ respectively, $v01$ is the number of variables controlled by a binary variable z and also $1-z$.

Instance	$(bv, tv, fv, b0, b1, b01, v0, v1, v01)$
alan	(4, 4, 0, 4, 0, 0, 4, 0, 0)
batch0812	(60, 60, 28, 0, 32, 0, 0, 70, 0)
batchdes	(9, 9, 1, 0, 8, 0, 0, 12, 0)
batch	(24, 24, 2, 0, 22, 0, 0, 30, 0)
batchs101006m	(120, 120, 0, 0, 120, 0, 0, 140, 0)
batchs121208m	(191, 191, 0, 0, 191, 0, 0, 215, 0)
batchs151208m	(188, 188, 0, 0, 188, 0, 0, 212, 0)
batchs201210m	(225, 225, 0, 0, 225, 0, 0, 249, 0)
clay0203h	(18, 18, 0, 0, 0, 18, 60, 12, 6)
clay0203m	(18, 18, 0, 0, 12, 6, 0, 12, 6)
clay0204h	(32, 32, 0, 0, 0, 32, 112, 24, 8)
clay0204m	(32, 32, 0, 0, 24, 8, 0, 24, 8)
clay0205h	(50, 50, 0, 0, 0, 50, 180, 40, 10)
clay0205m	(50, 50, 0, 0, 40, 10, 0, 40, 10)
clay0303h	(21, 21, 0, 0, 0, 21, 66, 21, 0)
clay0303m	(21, 21, 0, 0, 21, 0, 0, 21, 0)
clay0304h	(36, 36, 0, 0, 0, 36, 120, 36, 0)
clay0304m	(36, 36, 0, 0, 36, 0, 0, 36, 0)
clay0305h	(55, 55, 0, 0, 0, 55, 190, 55, 0)
clay0305m	(55, 55, 0, 0, 55, 0, 0, 55, 0)
color_lab2_4x0	(28920, 28920, 0, 0, 28680, 240, 28680, 240, 0)
color_lab6b_4x20	(27730, 27730, 0, 0, 27495, 235, 27495, 235, 0)
enpro48pb	(92, 92, 0, 0, 92, 0, 0, 108, 0)
enpro56pb	(73, 73, 0, 0, 73, 0, 0, 85, 0)
fac1	(6, 6, 0, 2, 0, 4, 16, 0, 4)
fac2	(12, 12, 0, 3, 0, 9, 54, 0, 9)
fac3	(12, 12, 0, 3, 0, 9, 54, 0, 9)
flay02h	(4, 4, 0, 0, 0, 4, 32, 4, 0)
flay02m	(4, 4, 0, 0, 4, 0, 0, 4, 0)
flay03h	(12, 12, 0, 0, 0, 12, 96, 12, 0)
flay03m	(12, 12, 0, 0, 12, 0, 0, 12, 0)
flay04h	(24, 24, 0, 0, 0, 24, 192, 24, 0)
flay04m	(24, 24, 0, 0, 24, 0, 0, 24, 0)
flay05h	(40, 40, 0, 0, 0, 40, 320, 40, 0)
flay05m	(40, 40, 0, 0, 40, 0, 0, 40, 0)
flay06h	(60, 60, 0, 0, 0, 60, 480, 60, 0)
flay06m	(60, 60, 0, 0, 60, 0, 0, 60, 0)
gams01	(110, 110, 0, 0, 100, 0, 0, 100, 0)
hybriddynamic_fixed	(1, 1, 0, 0, 0, 1, 8, 0, 2)
ibs2	(1500, 1500, 0, 0, 0, 1500, 1500, 1500, 0)
meanvarx	(12, 12, 0, 2, 0, 10, 12, 10, 0)
meanvarxsc	(22, 22, 0, 12, 0, 10, 12, 10, 0)
netmod_dol1	(462, 462, 0, 0, 0, 462, 1524, 462, 0)
netmod_dol2	(455, 455, 0, 0, 79, 367, 973, 440, 6)
netmod_kar1	(136, 136, 0, 0, 15, 121, 255, 136, 0)
netmod_kar2	(136, 136, 0, 0, 15, 121, 255, 136, 0)
pedigree_ex1058	(49386, 49386, 0, 112, 48387, 865, 48387, 112, 865)
pedigree_ex485_2	(7136, 7136, 0, 110, 6710, 294, 6710, 110, 294)
pedigree_ex485	(7136, 7136, 0, 110, 6710, 294, 6710, 110, 294)
pedigree_sim400	(11226, 11226, 0, 51, 11076, 99, 11076, 51, 99)
pedigree_sp_top4_250	(11694, 11694, 0, 243, 10981, 414, 10981, 243, 414)
pedigree_sp_top4_300	(5969, 5969, 0, 160, 5496, 244, 5496, 160, 244)
pedigree_sp_top4_350tr	(3100, 3100, 0, 105, 2838, 145, 2838, 105, 145)
pedigree_sp_top5_200	(32120, 32120, 0, 336, 30862, 871, 30862, 336, 871)
pedigree_sp_top5_250	(17028, 17028, 0, 243, 16193, 536, 16193, 243, 536)
portfol_buyin	(8, 8, 0, 8, 0, 0, 8, 0, 0)
portfol_card	(8, 8, 0, 8, 0, 0, 8, 0, 0)
portfol_classical050_1	(50, 50, 0, 50, 0, 0, 50, 0, 0)
portfol_classical200_2	(200, 200, 0, 200, 0, 0, 200, 0, 0)
procurement2mot	(60, 60, 0, 19, 3, 38, 77, 18, 23)
ravempb	(53, 53, 0, 0, 53, 0, 0, 65, 0)
risk2bpb	(12, 12, 0, 0, 12, 0, 0, 183, 0)
rsyn0805h	(37, 37, 0, 3, 0, 34, 84, 32, 26)
rsyn0805m02h	(148, 148, 0, 3, 0, 145, 171, 37, 166)
rsyn0805m02m	(148, 148, 0, 3, 64, 81, 19, 53, 118)
rsyn0805m03h	(222, 222, 0, 3, 0, 219, 255, 42, 264)
rsyn0805m03m	(222, 222, 0, 3, 96, 123, 27, 66, 192)
rsyn0805m04h	(296, 296, 0, 3, 0, 293, 339, 47, 362)
rsyn0805m04m	(296, 296, 0, 3, 128, 165, 35, 79, 266)
rsyn0805m	(37, 37, 0, 3, 32, 2, 8, 32, 2)
rsyn0810h	(41, 41, 0, 3, 0, 38, 95, 34, 26)
rsyn0810m02h	(166, 166, 0, 3, 0, 163, 187, 47, 182)
rsyn0810m02m	(166, 166, 0, 3, 64, 99, 35, 63, 134)
rsyn0810m03h	(249, 249, 0, 3, 0, 246, 278, 57, 289)
rsyn0810m03m	(249, 249, 0, 3, 96, 150, 50, 81, 217)
rsyn0810m04h	(332, 332, 0, 3, 0, 329, 369, 67, 396)
rsyn0810m04m	(332, 332, 0, 3, 128, 201, 65, 99, 300)
rsyn0810m	(41, 41, 0, 3, 32, 6, 19, 34, 2)
rsyn0815h	(44, 44, 0, 3, 0, 41, 105, 35, 27)
rsyn0815m02h	(182, 182, 0, 3, 0, 179, 204, 57, 197)
rsyn0815m02m	(182, 182, 0, 3, 64, 115, 52, 73, 149)
rsyn0815m03h	(273, 273, 0, 3, 0, 270, 303, 72, 312)
rsyn0815m03m	(273, 273, 0, 3, 96, 174, 75, 96, 240)
rsyn0815m04h	(364, 364, 0, 3, 0, 361, 402, 87, 427)
rsyn0815m04m	(364, 364, 0, 3, 128, 233, 98, 119, 331)
rsyn0815m	(44, 44, 0, 3, 32, 9, 29, 35, 3)
rsyn0820h	(49, 49, 0, 3, 0, 46, 116, 35, 29)
rsyn0820m02h	(202, 202, 0, 3, 0, 199, 223, 67, 214)
rsyn0820m02m	(202, 202, 0, 3, 64, 135, 71, 83, 166)
rsyn0820m03h	(303, 303, 0, 3, 0, 300, 330, 87, 339)
rsyn0820m03m	(303, 303, 0, 3, 96, 204, 102, 111, 267)
rsyn0820m04h	(404, 404, 0, 3, 0, 401, 437, 107, 464)
rsyn0820m04m	(404, 404, 0, 3, 128, 273, 133, 139, 368)
rsyn0820m	(49, 49, 0, 3, 32, 14, 40, 35, 5)
rsyn0830h	(58, 58, 0, 6, 0, 52, 136, 37, 30)
rsyn0830m02h	(240, 240, 0, 6, 0, 234, 259, 90, 243)

Instance	$(bv, tv, fv, b0, b1, b01, v0, v1, v01)$
rsyn0830m02m	(240, 240, 0, 6, 64, 170, 107, 106, 195)
rsyn0830m03h	(360, 360, 0, 6, 0, 354, 381, 120, 387)
rsyn0830m03m	(360, 360, 0, 6, 96, 258, 153, 144, 315)
rsyn0830m04h	(480, 480, 0, 6, 0, 474, 503, 150, 531)
rsyn0830m04m	(480, 480, 0, 6, 128, 346, 199, 182, 435)
rsyn0830m	(58, 58, 0, 6, 32, 20, 60, 37, 6)
rsyn0840h	(66, 66, 0, 6, 0, 60, 157, 38, 33)
rsyn0840m02h	(276, 276, 0, 6, 0, 270, 295, 110, 275)
rsyn0840m02m	(276, 276, 0, 6, 64, 206, 143, 126, 227)
rsyn0840m03h	(414, 414, 0, 6, 0, 408, 433, 150, 437)
rsyn0840m03m	(414, 414, 0, 6, 96, 312, 205, 174, 365)
rsyn0840m04h	(552, 552, 0, 6, 0, 546, 571, 190, 599)
rsyn0840m04m	(552, 552, 0, 6, 128, 418, 267, 222, 503)
rsyn0840m	(66, 66, 0, 6, 32, 28, 81, 38, 9)
slay04h	(24, 24, 0, 0, 0, 24, 96, 24, 0)
slay04m	(24, 24, 0, 0, 24, 0, 0, 24, 0)
slay05h	(40, 40, 0, 0, 0, 40, 160, 40, 0)
slay05m	(40, 40, 0, 0, 40, 0, 0, 40, 0)
slay06h	(60, 60, 0, 0, 0, 60, 240, 60, 0)
slay06m	(60, 60, 0, 0, 60, 0, 0, 60, 0)
slay07h	(84, 84, 0, 0, 0, 84, 336, 84, 0)
slay07m	(84, 84, 0, 0, 84, 0, 0, 84, 0)
slay08h	(112, 112, 0, 0, 0, 112, 448, 112, 0)
slay08m	(112, 112, 0, 0, 112, 0, 0, 112, 0)
slay09h	(144, 144, 0, 0, 0, 144, 576, 144, 0)
slay09m	(144, 144, 0, 0, 144, 0, 0, 144, 0)
slay10h	(180, 180, 0, 0, 0, 180, 720, 180, 0)
slay10m	(180, 180, 0, 0, 180, 0, 0, 180, 0)
squfl010-025	(10, 10, 0, 10, 0, 0, 250, 0, 0)
squfl010-040	(10, 10, 0, 10, 0, 0, 400, 0, 0)
squfl010-080	(10, 10, 0, 10, 0, 0, 800, 0, 0)
squfl015-060	(15, 15, 0, 15, 0, 0, 900, 0, 0)
squfl015-080	(15, 15, 0, 15, 0, 0, 1200, 0, 0)
squfl020-040	(20, 20, 0, 20, 0, 0, 800, 0, 0)
squfl020-050	(20, 20, 0, 20, 0, 0, 1000, 0, 0)
squfl020-150	(20, 20, 0, 20, 0, 0, 3000, 0, 0)
squfl025-025	(25, 25, 0, 25, 0, 0, 625, 0, 0)
squfl025-030	(25, 25, 0, 25, 0, 0, 750, 0, 0)
squfl025-040	(25, 25, 0, 25, 0, 0, 1000, 0, 0)
squfl030-100	(30, 30, 0, 30, 0, 0, 3000, 0, 0)
squfl030-150	(30, 30, 0, 30, 0, 0, 4500, 0, 0)
squfl040-080	(40, 40, 0, 40, 0, 0, 3200, 0, 0)
sssd08-04	(44, 44, 0, 0, 32, 12, 12, 44, 0)
sssd12-05	(75, 75, 0, 0, 60, 15, 15, 75, 0)
sssd15-04	(72, 72, 0, 0, 60, 12, 12, 72, 0)
sssd15-06	(108, 108, 0, 0, 90, 18, 18, 108, 0)
sssd15-08	(144, 144, 0, 0, 120, 24, 24, 144, 0)
sssd16-07	(133, 133, 0, 0, 112, 21, 21, 133, 0)
sssd18-06	(126, 126, 0, 0, 108, 18, 18, 126, 0)
sssd18-08	(168, 168, 0, 0, 144, 24, 24, 168, 0)
sssd20-04	(92, 92, 0, 0, 80, 12, 12, 92, 0)
sssd20-08	(184, 184, 0, 0, 160, 24, 24, 184, 0)
sssd22-08	(200, 200, 0, 0, 176, 24, 24, 200, 0)
sssd25-04	(112, 112, 0, 0, 100, 12, 12, 112, 0)
sssd25-08	(224, 224, 0, 0, 200, 24, 24, 224, 0)
st_miqp2	(2, 2, 0, 2, 0, 0, 2, 0, 0)
st_miqp4	(2, 2, 0, 2, 0, 0, 2, 0, 0)
stockcycle	(432, 432, 0, 0, 432, 0, 0, 480, 0)
st_test3	(5, 5, 0, 3, 0, 0, 3, 0, 0)
syn05h	(5, 5, 0, 3, 0, 2, 8, 0, 2)
syn05m02h	(20, 20, 0, 3, 0, 17, 19, 13, 14)
syn05m02m	(20, 20, 0, 3, 0, 17, 19, 13, 14)

Instance	$(bv, tv, fv, b0, b1, b01, v0, v1, v01)$
syn05m03h	(30, 30, 0, 3, 0, 27, 27, 18, 24)
syn05m03m	(30, 30, 0, 3, 0, 27, 27, 18, 24)
syn05m04h	(40, 40, 0, 3, 0, 37, 35, 23, 34)
syn05m04m	(40, 40, 0, 3, 0, 37, 35, 23, 34)
syn05m	(5, 5, 0, 3, 0, 2, 8, 0, 2)
syn10h	(9, 9, 0, 3, 0, 6, 19, 2, 2)
syn10m02h	(38, 38, 0, 3, 0, 35, 35, 23, 30)
syn10m02m	(38, 38, 0, 3, 0, 35, 35, 23, 30)
syn10m03h	(57, 57, 0, 3, 0, 54, 50, 33, 49)
syn10m03m	(57, 57, 0, 3, 0, 54, 50, 33, 49)
syn10m04h	(76, 76, 0, 3, 0, 73, 65, 43, 68)
syn10m04m	(76, 76, 0, 3, 0, 73, 65, 43, 68)
syn10m	(9, 9, 0, 3, 0, 6, 19, 2, 2)
syn15h	(12, 12, 0, 3, 0, 9, 29, 3, 3)
syn15m02h	(54, 54, 0, 3, 0, 51, 52, 33, 45)
syn15m02m	(54, 54, 0, 3, 0, 51, 52, 33, 45)
syn15m03h	(81, 81, 0, 3, 0, 78, 75, 48, 72)
syn15m03m	(81, 81, 0, 3, 0, 78, 75, 48, 72)
syn15m04h	(108, 108, 0, 3, 0, 105, 98, 63, 99)
syn15m04m	(108, 108, 0, 3, 0, 105, 98, 63, 99)
syn15m	(12, 12, 0, 3, 0, 9, 29, 3, 3)
syn20h	(17, 17, 0, 3, 0, 14, 40, 3, 5)
syn20m02h	(74, 74, 0, 3, 0, 71, 71, 43, 62)
syn20m02m	(74, 74, 0, 3, 0, 71, 71, 43, 62)
syn20m03h	(111, 111, 0, 3, 0, 108, 102, 63, 99)
syn20m03m	(111, 111, 0, 3, 0, 108, 102, 63, 99)
syn20m04h	(148, 148, 0, 3, 0, 145, 133, 83, 136)
syn20m04m	(148, 148, 0, 3, 0, 145, 133, 83, 136)
syn20m	(17, 17, 0, 3, 0, 14, 40, 3, 5)
syn30h	(26, 26, 0, 6, 0, 20, 60, 5, 6)
syn30m02h	(112, 112, 0, 6, 0, 106, 107, 66, 91)
syn30m02m	(112, 112, 0, 6, 0, 106, 107, 66, 91)
syn30m03h	(168, 168, 0, 6, 0, 162, 153, 96, 147)
syn30m03m	(168, 168, 0, 6, 0, 162, 153, 96, 147)
syn30m04h	(224, 224, 0, 6, 0, 218, 199, 126, 203)
syn30m04m	(224, 224, 0, 6, 0, 218, 199, 126, 203)
syn30m	(26, 26, 0, 6, 0, 20, 60, 5, 6)
syn40h	(34, 34, 0, 6, 0, 28, 81, 6, 9)
syn40m02h	(148, 148, 0, 6, 0, 142, 143, 86, 123)
syn40m02m	(148, 148, 0, 6, 0, 142, 143, 86, 123)
syn40m03h	(222, 222, 0, 6, 0, 216, 205, 126, 197)
syn40m03m	(222, 222, 0, 6, 0, 216, 205, 126, 197)
syn40m04h	(296, 296, 0, 6, 0, 290, 267, 166, 271)
syn40m04m	(296, 296, 0, 6, 0, 290, 267, 166, 271)
syn40m	(34, 34, 0, 6, 0, 28, 81, 6, 9)
synthes1	(3, 3, 0, 0, 1, 1, 1, 2, 0)
synthes2	(5, 5, 0, 1, 1, 3, 3, 2, 2)
synthes3	(8, 8, 0, 3, 1, 4, 8, 3, 2)
tls12	(489, 489, 0, 12, 465, 12, 0, 504, 129)
tls2	(31, 31, 0, 2, 29, 0, 0, 18, 17)
tls4	(85, 85, 0, 4, 81, 0, 0, 76, 25)
tls5	(131, 131, 0, 5, 126, 0, 0, 125, 31)
tls6	(165, 165, 0, 6, 159, 0, 0, 156, 45)
tls7	(278, 278, 0, 7, 271, 0, 0, 266, 61)
unitcommit1	(427, 427, 0, 9, 235, 179, 310, 196, 83)
unitcommit_200_100_1_mod_8	(4380, 4380, 0, 3843, 0, 537, 13245, 398, 190)
unitcommit_200_100_2_mod_8	(4400, 4400, 0, 3969, 0, 431, 13148, 530, 230)
unitcommit_50_20_2_mod_8	(1093, 1093, 0, 991, 0, 102, 3259, 132, 58)
watercontamination0202	(7, 7, 0, 7, 0, 0, 521, 0, 0)
watercontamination0202r	(7, 7, 0, 7, 0, 0, 188, 0, 0)
watercontamination0303	(14, 14, 0, 14, 0, 0, 1046, 0, 0)
watercontamination0303r	(14, 14, 0, 14, 0, 0, 370, 0, 0)

Table 7 Description of test set T_{pr} of 104 instances with structures, (PS_1) and (PS_2), amenable to perspective reformulation in the Section 2. The entry in the first column is the instance name and for each instance the entries in the second column are as follows: ts denotes total number of nonlinear constraints, pc shows the number of PR amenable constraints, $s1$ and $s2$ report number of constraints (out of pc) of type (S_1) and (S_2), respectively, and the last entry ub denotes the number of unique variables associated with PR amenable constraints.

Instance	$(tc, pc, s1, s2)$
clay0203h	$(24, 24, 24, 0)$
clay0204h	$(32, 32, 32, 0)$
clay0205h	$(40, 40, 40, 0)$
clay0303h	$(36, 36, 36, 0)$
clay0304h	$(48, 48, 48, 0)$
clay0305h	$(60, 60, 60, 0)$
rsyn0805h	$(3, 3, 3, 0)$
rsyn0805m02h	$(6, 6, 6, 0)$
rsyn0805m02m	$(6, 6, 6, 0)$
rsyn0805m03h	$(9, 9, 9, 0)$
rsyn0805m03m	$(9, 9, 9, 0)$
rsyn0805m04h	$(12, 12, 12, 0)$
rsyn0805m04m	$(12, 12, 12, 0)$
rsyn0805m	$(3, 3, 3, 0)$
rsyn0810h	$(6, 6, 6, 0)$
rsyn0810m02h	$(12, 12, 12, 0)$
rsyn0810m02m	$(12, 12, 12, 0)$
rsyn0810m03h	$(18, 18, 18, 0)$
rsyn0810m03m	$(18, 18, 18, 0)$
rsyn0810m04h	$(24, 24, 24, 0)$
rsyn0810m04m	$(24, 24, 24, 0)$
rsyn0810m	$(6, 6, 6, 0)$
rsyn0815h	$(11, 11, 11, 0)$
rsyn0815m02h	$(22, 22, 22, 0)$
rsyn0815m02m	$(22, 22, 22, 0)$
rsyn0815m03h	$(33, 33, 33, 0)$
rsyn0815m03m	$(33, 33, 33, 0)$
rsyn0815m04h	$(44, 44, 44, 0)$
rsyn0815m04m	$(44, 44, 44, 0)$
rsyn0815m	$(11, 11, 11, 0)$
rsyn0820h	$(14, 14, 14, 0)$
rsyn0820m02h	$(28, 28, 28, 0)$
rsyn0820m02m	$(28, 28, 28, 0)$
rsyn0820m03h	$(42, 42, 42, 0)$
rsyn0820m03m	$(42, 42, 42, 0)$

Instance	$(tc, pc, s1, s2)$
rsyn0820m04h	$(56, 56, 56, 0)$
rsyn0820m04m	$(56, 56, 56, 0)$
rsyn0820m	$(14, 14, 14, 0)$
rsyn0830h	$(20, 20, 20, 0)$
rsyn0830m02h	$(40, 40, 40, 0)$
rsyn0830m02m	$(40, 40, 40, 0)$
rsyn0830m03h	$(60, 60, 60, 0)$
rsyn0830m03m	$(60, 60, 60, 0)$
rsyn0830m04h	$(80, 80, 80, 0)$
rsyn0830m04m	$(80, 80, 80, 0)$
rsyn0830m	$(20, 20, 20, 0)$
rsyn0840h	$(28, 28, 28, 0)$
rsyn0840m02h	$(56, 56, 56, 0)$
rsyn0840m02m	$(56, 56, 56, 0)$
rsyn0840m03h	$(84, 84, 84, 0)$
rsyn0840m03m	$(84, 84, 84, 0)$
rsyn0840m04h	$(112, 112, 112, 0)$
rsyn0840m04m	$(112, 112, 112, 0)$
rsyn0840m	$(28, 28, 28, 0)$
syn05h	$(3, 3, 3, 0)$
syn05m02h	$(6, 6, 6, 0)$
syn05m02m	$(6, 6, 6, 0)$
syn05m03h	$(9, 9, 9, 0)$
syn05m03m	$(9, 9, 9, 0)$
syn05m04h	$(12, 12, 12, 0)$
syn05m04m	$(12, 12, 12, 0)$
syn05m	$(3, 3, 3, 0)$
syn10h	$(6, 6, 6, 0)$
syn10m02h	$(12, 12, 12, 0)$
syn10m02m	$(12, 12, 12, 0)$
syn10m03h	$(18, 18, 18, 0)$
syn10m03m	$(18, 18, 18, 0)$
syn10m04h	$(24, 24, 24, 0)$
syn10m04m	$(24, 24, 24, 0)$
syn10m	$(6, 6, 6, 0)$

Instance	$(tc, pc, s1, s2)$
syn15h	$(11, 11, 11, 0)$
syn15m02h	$(22, 22, 22, 0)$
syn15m02m	$(22, 22, 22, 0)$
syn15m03h	$(33, 33, 33, 0)$
syn15m03m	$(33, 33, 33, 0)$
syn15m04h	$(44, 44, 44, 0)$
syn15m04m	$(44, 44, 44, 0)$
syn15m	$(11, 11, 11, 0)$
syn20h	$(14, 14, 14, 0)$
syn20m02h	$(28, 28, 28, 0)$
syn20m02m	$(28, 28, 28, 0)$
syn20m03h	$(42, 42, 42, 0)$
syn20m03m	$(42, 42, 42, 0)$
syn20m04h	$(56, 56, 56, 0)$
syn20m04m	$(56, 56, 56, 0)$
syn20m	$(14, 14, 14, 0)$
syn30h	$(20, 20, 20, 0)$
syn30m02h	$(40, 40, 40, 0)$
syn30m02m	$(40, 40, 40, 0)$
syn30m03h	$(60, 60, 60, 0)$
syn30m03m	$(60, 60, 60, 0)$
syn30m04h	$(80, 80, 80, 0)$
syn30m04m	$(80, 80, 80, 0)$
syn30m	$(20, 20, 20, 0)$
syn40h	$(28, 28, 28, 0)$
syn40m02h	$(56, 56, 56, 0)$
syn40m02m	$(56, 56, 56, 0)$
syn40m03h	$(84, 84, 84, 0)$
syn40m03m	$(84, 84, 84, 0)$
syn40m04h	$(112, 112, 112, 0)$
syn40m04m	$(112, 112, 112, 0)$
syn40m	$(28, 28, 28, 0)$
synthes2	$(3, 1, 1, 0)$
synthes3	$(4, 2, 1, 1)$

Table 8 Description of the instances in the test set TS_{sep} for separability based reformulation in the Section 3. First column shows the instance name and the entries (nc, sp, os, us, rs) in the second column are: nc and sc number of nonlinear constraints and number of separable nonlinear constraints, respectively, os indicates whether objective function is separabe (1) or not (0), us is the number of unique separable parts considering all separable constraints and objective function, rs is the number of separable parts that are repeated. 26 instances also belonging to the test set TS_{ps} (that became amenable to perspective reformulation after the reformulation based on separability of nonlinear constraints and objective) are highlighted in bold.

Instance	(nc, sp, os, us, rs)
ball_mk2_10	(1, 1, 0, 10, 0)
ball_mk2_30	(1, 1, 0, 30, 0)
ball_mk3_10	(1, 1, 0, 10, 0)
ball_mk3_20	(1, 1, 0, 20, 0)
ball_mk3_30	(1, 1, 0, 30, 0)
ball_mk4_05	(1, 1, 0, 5, 0)
ball_mk4_10	(1, 1, 0, 10, 0)
ball_mk4_15	(1, 1, 0, 15, 0)
batch0812	(2, 2, 0, 20, 0)
batchdes	(2, 2, 0, 5, 0)
batch	(2, 2, 0, 11, 0)
batchs101006m	(2, 2, 0, 29, 0)
batchs121208m	(2, 2, 0, 35, 0)
batchs151208m	(2, 2, 0, 38, 0)
batchs201210m	(2, 2, 0, 43, 0)
clay0203m	(24, 24, 0, 24, 24)
clay0204m	(32, 32, 0, 32, 32)
clay0205m	(40, 40, 0, 40, 40)
clay0303m	(36, 36, 0, 36, 36)
clay0304m	(48, 48, 0, 48, 48)
clay0305m	(60, 60, 0, 60, 60)
enpro48pb	(2, 2, 0, 13, 0)
enpro56pb	(2, 2, 0, 12, 0)
ex1223a	(5, 2, 0, 6, 0)
ex1223b	(5, 5, 0, 12, 5)
ex1223	(5, 5, 0, 12, 5)
ex4	(26, 26, 0, 125, 2)
fac1	(1, 0, 1, 2, 0)
fac2	(1, 0, 1, 3, 0)
fac3	(1, 0, 1, 3, 0)
gams01	(111, 0, 1, 10, 0)
hybriddynamic_fixed	(1, 0, 1, 11, 0)
immun	(1, 0, 1, 6, 0)
netmod_dol1	(1, 0, 1, 6, 0)
netmod_dol2	(1, 0, 1, 6, 0)
netmod_kar1	(1, 0, 1, 4, 0)
netmod_kar2	(1, 0, 1, 4, 0)
nvs03	(2, 0, 1, 2, 0)
nvs10	(3, 0, 1, 2, 0)
pedigree_ex1058	(1, 1, 0, 28, 0)
pedigree_ex485_2	(1, 1, 0, 28, 0)
pedigree_ex485	(1, 1, 0, 28, 0)
pedigree_sp_top4_250	(1, 1, 0, 58, 0)
pedigree_sp_top4_300	(1, 1, 0, 74, 0)
pedigree_sp_top4_350tr	(1, 1, 0, 17, 0)
pedigree_sp_top5_200	(1, 1, 0, 54, 0)
pedigree_sp_top5_250	(1, 1, 0, 58, 0)
portfol_classical050_1	(1, 1, 0, 50, 0)
portfol_classical200_2	(1, 1, 0, 200, 0)
risk2bpb	(1, 0, 1, 3, 0)
slay04h	(1, 0, 1, 8, 0)
slay04m	(1, 0, 1, 8, 0)
slay05h	(1, 0, 1, 10, 0)
slay05m	(1, 0, 1, 10, 0)

Instance	(nc, sp, os, us, rs)
slay06h	(1, 0, 1, 12, 0)
slay06m	(1, 0, 1, 12, 0)
slay07h	(1, 0, 1, 14, 0)
slay07m	(1, 0, 1, 14, 0)
slay08h	(1, 0, 1, 16, 0)
slay08m	(1, 0, 1, 16, 0)
slay09h	(1, 0, 1, 18, 0)
slay09m	(1, 0, 1, 18, 0)
slay10h	(1, 0, 1, 20, 0)
slay10m	(1, 0, 1, 20, 0)
squfl010-025	(1, 0, 1, 250, 0)
squfl010-040	(1, 0, 1, 400, 0)
squfl010-080	(1, 0, 1, 800, 0)
squfl015-060	(1, 0, 1, 900, 0)
squfl015-080	(1, 0, 1, 1200, 0)
squfl020-040	(1, 0, 1, 800, 0)
squfl020-050	(1, 0, 1, 1000, 0)
squfl020-150	(1, 0, 1, 3000, 0)
squfl025-025	(1, 0, 1, 625, 0)
squfl025-030	(1, 0, 1, 750, 0)
squfl025-040	(1, 0, 1, 1000, 0)
squfl030-100	(1, 0, 1, 3000, 0)
squfl030-150	(1, 0, 1, 4500, 0)
squfl040-080	(1, 0, 1, 3200, 0)
st_e14	(5, 5, 0, 12, 5)
st_miqp1	(1, 0, 1, 5, 0)
st_miqp2	(1, 0, 1, 2, 0)
st_miqp4	(1, 0, 1, 3, 0)
st_miqp5	(1, 0, 1, 2, 0)
stockcycle	(1, 0, 1, 48, 0)
st_test1	(1, 0, 1, 4, 0)
st_test2	(1, 0, 1, 5, 0)
st_test3	(1, 0, 1, 5, 0)
st_test4	(1, 0, 1, 2, 0)
st_test5	(1, 0, 1, 7, 0)
st_test6	(1, 0, 1, 10, 0)
st_test8	(1, 0, 1, 24, 0)
st_testgr1	(1, 0, 1, 10, 0)
st_testgr3	(1, 0, 1, 20, 0)
st_testph4	(1, 0, 1, 3, 0)
synthes2	(4, 0, 1, 3, 0)
synthes3	(5, 2, 0, 6, 1)
tls12	(12, 12, 0, 144, 0)
tls2	(2, 2, 0, 4, 0)
tls4	(4, 4, 0, 16, 0)
tls5	(5, 5, 0, 25, 0)
tls6	(6, 6, 0, 36, 0)
tls7	(7, 7, 0, 49, 0)
unitcommit1	(1, 0, 1, 240, 0)
unitcommit_200_100_1_mod_8	(1, 0, 1, 4662, 0)
unitcommit_200_100_2_mod_8	(1, 0, 1, 4639, 0)
unitcommit_50_20_2_mod_8	(1, 0, 1, 1152, 0)
watercontamination0202	(1, 0, 1, 4017, 0)
watercontamination0303	(1, 0, 1, 4521, 0)

B Computational Results

Table 9 Summary of collections of type $C_i, i = 1, 2, 3$ in instances in test set TS_c. The second column reports the number of instances containing at least one collection of the type mentioned in the first column. In the last column, the first sub-column corresponds to the number of instances (out of the number of instances mentioned under the second column) in which at least 50% of the total number of variables are found to be semi-continuous. The second sub-column shows the number of instances in which the total number of semi-continuous variables is less than 10%.

		# inst. with semi-continuous variables	
type	# inst.	$\geq 50\%$	$\leq 10\%$
C_1	194	151	9
C_2	132	41	5
C_1 and C_2	220	203	0
C_1 and C_3	194	154	7
C_2 and C_3	132	43	5
C_1 and C_2 and C_3	220	208	0

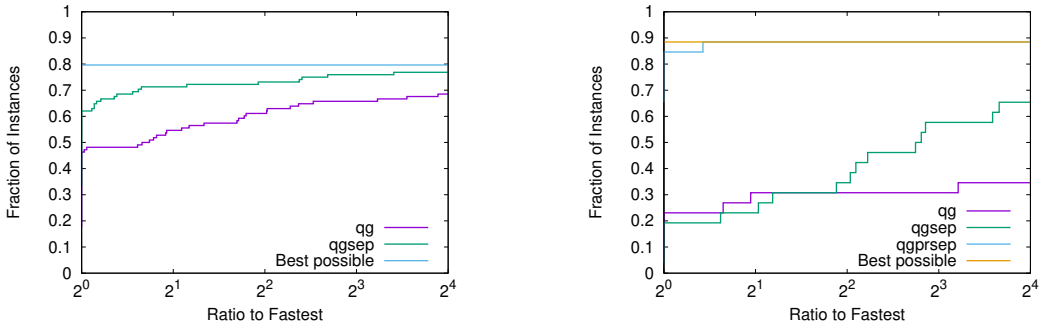


Figure 3 Performance profiles comparing solution times of *qg* and *qgsep* (on left), and of *qg*, *qgsep*, and *qgprsep* (on right).

Table 10 (Top) Comparison of *qg* and methods (M) on 15 instances in TS_{ps} that are solved by both the techniques. (Bottom) Performance on ten instances that are solved by both, but at least one technique took more than 10 seconds.

	time		nodes	
Method (M)	*qg*	rel.	*qg*	rel.
qgsep	78.24	0.32	1213.03	0.42
qgprsep	78.24	0.12	1213.03	0.12

	time		nodes	
Method (M)	*qg*	rel.	*qg*	rel.
qgsep	251.01	0.22	4418.17	0.31
qgprsep	251.01	0.07	4418.17	0.06

Table 11 (Top) Comparison of *qg* and methods (M) on instances that are solved by both, but at least one method took more than 100 seconds. (Bottom) Similar comparisons for instances that are solved by both methods, but at least one took more than 500 seconds.

# solved	time		nodes	
by both	*qg*	rel.	*qg*	rel.
7	654.78	0.15	10605.48	0.27
5	1454.86	0.01	15711.76	0.02

# solved	time		nodes	
by both	*qg*	rel.	*qg*	rel.
4	2389.38	0.04	27068.51	0.15
4	2389.38	0.01	27068.51	0.01

An Adaptive Refinement Algorithm for Discretizations of Nonconvex QCQP

Akshay Gupte[1]
School of Mathematics, The University of Edinburgh, UK

Arie M. C. A. Koster
Lehrstuhl II für Mathematik, RWTH Aachen University, Germany

Sascha Kuhnke
Lehrstuhl II für Mathematik, RWTH Aachen University, Germany

Abstract

We present an iterative algorithm to compute feasible solutions in reasonable running time to quadratically constrained quadratic programs (QCQPs), which form a challenging class of nonconvex continuous optimization. This algorithm is based on a mixed-integer linear program (MILP) which is a restriction of the original QCQP obtained by discretizing all quadratic terms. In each iteration, this MILP restriction is solved to get a feasible QCQP solution. Since the quality of this solution heavily depends on the chosen discretization of the MILP, we iteratively adapt the discretization values based on the MILP solution of the previous iteration. To maintain a reasonable problem size in each iteration of the algorithm, the discretization sizes are fixed at predefined values. Although our algorithm did not always yield good feasible solutions on arbitrary QCQP instances, an extensive computational study on almost 1300 test instances of two different problem classes – box-constrained quadratic programs with complementarity constraints and disjoint bilinear programs, demonstrates the effectiveness of our approach. We compare the quality of our solutions against those from heuristics and local optimization algorithms in two state-of-the-art commercial solvers and observe that on one instance class we clearly outperform the other methods whereas on the other class we obtain competitive results.

2012 ACM Subject Classification Theory of computation → Mixed discrete-continuous optimization; Mathematics of computing → Nonconvex optimization

Keywords and phrases Quadratically Constrained Quadratic Programs, Mixed Integer Linear Programming, Heuristics, BoxQP, Disjoint Bilinear

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.24

Supplementary Material *Software (Source Code)*: `https://github.com/skuhnke/qcqp_sourcecode`

Funding The second and third authors' visit was supported by a grant from DAAD, the German Academic Exchange Service.
Akshay Gupte: Initial phase of this research supported by NSF grant DMS-1913294.

Acknowledgements This research was initiated during the second and third authors' visit to Clemson University, USA, in 2019, where the first author was a faculty member.

1 Introduction

A quadratically constrained quadratic program (QCQP) is the optimization problem

$$\begin{aligned} z^* = \max_x \quad & x^\top Q_0\, x + c_0^\top\, x \\ \text{s.t.} \quad & x^\top Q_r\, x + c_r^\top\, x \le b_r, \qquad r \in R \\ & x \in P \end{aligned}$$

[1] Corresponding author

where $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, the set $P \subseteq \mathbb{R}^n$ is a polytope, the index set R is allowed to be empty so that only the objective is quadratic, and all the data are of conformable dimensions. We assume that the matrices Q_r are symmetric for all $r \in R \cup \{0\}$. When at least one of these matrices has at least one negative eigenvalue (i.e., is not positive semidefinite), then the QCQP instance is nonconvex, which are of interest to us in this paper since they are global optimization problems and solving them to optimality is NP-hard in general. It is common practice in global optimization to assume finite lower and upper bounds on the variables, these bounds may either be pre-specified in the problem or implied by the quadratic constraints or P. The assumption of P being in the nonnegative orthant can be easily achieved by translating the variables. We assume feasible instances.

1.1 Background

QCQPs are a rich and important class of nonconvex optimization with a wide-range of applications. There has been active research for many decades on solution algorithms for them, but a majority of the literature is devoted to convex relaxations and cutting planes that can be used in spatial branch-and-cut algorithms to find a global optimum. The focus of this paper is on generating strong primal (upper) bounds on z^*. A standard choice for doing this is to employ a nonlinear solver to search for a local optima and possibly improve on this local optimum [23]. Although local search heuristics work well sometimes on specific problems, such as box-constrained QPs [4] and also those arising from independent set problem in graphs [22], in general, local solutions can be far away from the global optimum and local solvers have been shown to either fail or produce not-so-good solutions when there are many difficult-to-satisfy nonconvex quadratic constraints. Employing primal heuristics, such as [2, 3, 7, 8, 18, 20], that generate feasible solutions of general mixed-integer nonlinear problems is another approach, but these primarily use integrality of variables and are not always effective for continuous nonconvex problems such as in QCQP. A third way is to solve a convex relaxation of the problem and then round its solution so that it becomes feasible to QCQP; however, a general rounding scheme is not known and difficult to characterise.

Another method for primal bounds is to develop restrictions (i.e., inner approximations) of the feasible region and optimize the objective over it. A common choice for generating restrictions is *variable discretization*, wherein the domain of a subset of variables is constrained to take values in a pre-defined finite set of reals. Although this idea is elementary, to the best of our knowledge, we have not seen it being widely employed for QCQP, except for the aforementioned pooling problem [16, 1, 24, 15, 10], for applications in energy systems [13] and water treatment [17], and for BoxQP [12], which is a subclass of QCQP wherein $R = \emptyset$ (no quadratic constraints) and $P = [0, 1]^n$. Discretising a QCQP yields a mixed-integer QCQP which can be reformulated as a mixed-integer LP (MILP) using different ways of expressing an integer variable as a linear function of binary variables [14]. Convexification studies for such binarization transformations have been carried out for MILPs [9]. The MILP approach to obtaining primal bounds for QCQP is attractive due to many decades of advances in solving MILPs and very sophisticated and powerful commercial solvers available for them. It also complements MILP-based relaxation techniques [5].

Variable discretization algorithms do not fix variables values, and in that sense they bear an advantage to some primal heuristics that obtain feasible solutions by fixing a subset of the variables to certain values [3] (typically obtained through a convex relaxation of the feasible set) and then optimizing over the variables that were left unfixed. However, as they are currently designed and implemented, they have the drawback of requiring that the discretised variables and values be pre-specified. Thus, these methods are "static" in the

sense that they solve only once and do not adapt to the solution that has been found, i.e., once a MILP has been solved and feasible solution obtained, there is no guidance on how to solve another MILP to improve on the current solution. An arbitrary discretization can likely miss close-to-optimal values that may be suggested by a convex relaxation. Of course, one could employ immense computing power and parallelise the solve of multiple MILPs, each with a different discretization scheme, and then take the best primal bound out of all these runs. However, such a brute-force approach is undesirable since it relies on a great amount of computing power and time that may not always be available, and can be rendered needless if instead one adopts a dynamic/adaptive approach where the discretization scheme gets updated according to the previous solution. We know of only two studies [13, 17] that take an adaptive discretization approach for nonconvex problems, but they are focused on specific applications. There are also other areas of optimization where adaptive schemes are used, for e.g., in stochastic programming [25].

1.2 Contributions of this Paper

We devise an adaptive variable discretization algorithm for any QCQP and test it extensively on different problem classes. Our iterative algorithm solves in each iteration an MILP restriction of the original QCQP based on discretization. The discretizations of the MILP restrictions are adapted after each iteration based on the previous solution. The numerical experiments show that our adaptive discretization algorithm yields very good feasible solutions that are competitive and frequently superior than those obtained from global solvers (which employ local solvers and also MILP heuristics) and other discretizations from literature, not only in terms of objective value but also in terms of the running time required for obtaining them. We remark that dynamic updating has been recently carried out for MILP *relaxations* [6, 19] of nonconvex problems, and so our algorithm serves as a complement and lays the groundwork for future work on a global optimization algorithm where both dual and primal bounds are computed by adapting MILP problems.

Section 2.1 introduces the procedure to discretize the quadratic terms. Then, we present in Section 2.2 a procedure to decide which variables in the quadratic terms to discretize. Section 2.3 explains the adaption of the discretization which we apply in each iteration of the algorithm. In Section 2.4, we present the whole adaptive discretization algorithm. Our computational experiments are described in section 3. The algorithms used in this study are introduced in Section 3.1 while details about their implementations are given in Section 3.2. Subsequently, we apply the algorithms to two different problem classes. Then, we perform in Section 3.3 calculations on a huge test set of 1230 BoxQPcc instances. Finally, in Section 3.4, we evaluate the performance of the algorithms on 60 DisjBLP instances.

2 Adaptive Discretization Algorithm

2.1 Discretization of Quadratic Terms

To solve nonconvex QCQPs to optimality, a global optimization algorithm is required which handles the quadratic terms $x_i x_j$. Global solvers such as `BARON` sometimes struggle to compute good feasible solutions in reasonable running time for several mid-size instances. Therefore, a common approach to obtain strong feasible solutions for such problems is discretization. In this paper, we reduce the solution space of the QCQP by restricting at least one of the continuous variables in each quadratic term to certain discrete values. This restriction can be equivalently reformulated into an MILP which is likely to be easier to solve than the original QCQP.

We propose a discretization method which is similar to the unary expansion [14]. To introduce this discretization method, we consider a single quadratic term $w_{ij} = x_i\, x_j$ where $x_i \in [\ell_i, u_i]$ and $x_j \in [\ell_j, u_j]$ with $\ell_i < u_i$ and $\ell_j < u_j$. This term can also be strictly quadratic, i.e., $i = j$. Let $u \geq 2$ be a positive integer which defines the size of the discretization. Then, we allow the originally continuous variable x_i to assume only one of the predefined equidistant values $x_{i1} < \cdots < x_{iu}$ with $x_{i1} \geq \ell_i$ and $x_{iu} \leq u_i$. This means we have $x_i \in \{x_{in} \mid n \in N\}$ where $N := \{1, \ldots, u\}$ is the set of indices of the discretized values. In the following, we express w_{ij} using linear expressions instead of a quadratic one. To this end, we introduce additional binary variables $z_{i1}, \ldots, z_{iu}$ which are used to set x_i equal to one of the values x_{in}:

$$x_i = \sum_{n \in N} x_{in}\, z_{in}, \qquad \sum_{n \in N} z_{in} = 1, \qquad z_{in} \in \{0,1\} \quad \forall\, n \in N. \tag{1}$$

We first consider the case where w_{ij} is bilinear, i.e., $i \neq j$. In this case, we add non-negative continuous variables $y_{ij1}, \ldots, y_{iju}$ defined by $y_{ijn} := x_j\, z_{in}$ for all $n \in N$. In the unary expansion, the bilinear terms $x_j\, z_{in}$ are linearized by applying their McCormick envelopes which consist of $4\,u$ additional constraints. By exploiting the SOS-1 property of the binary variables z_{in}, we use an equivalent but smaller linearization with only $2\,u + 1$ additional constraints:

$$x_j = \sum_{n \in N} y_{ijn}, \qquad \ell_j\, z_{in} \leq y_{ijn} \leq u_j\, z_{in} \quad \forall\, n \in N. \tag{2}$$

The above constraints (1)-(2) allow us to rewrite the bilinear term w_{ij} as a linear term where we sum over each discretized value x_{in} multiplied by corresponding additional variable y_{ijn}:

$$w_{ij} = \sum_{n \in N} x_{in}\, y_{ijn}\,. \tag{3}$$

In the second case where w_{ij} is strictly quadratic, i.e., $i = j$, the additional continuous variables y_{ijn} and their corresponding linearization (2) are not necessary. Along with constraints (1), we can rewrite w_{ii} as follows:

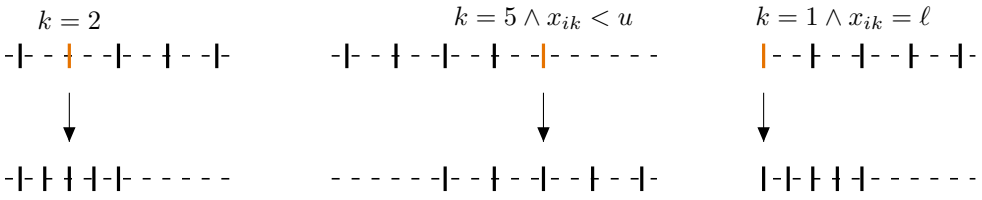
$$w_{ii} = \sum_{n \in N} x_{in}^2\, z_{in}\,. \tag{4}$$

It is known from the results in [14] that the above linearization of the bilinear terms $x_j z_{in}$ is stronger than the standard one using McCormick envelopes.

By discretizing all quadratic terms in the QCQP as described above, the continuous non-convex problem turns into a mixed-integer linear problem which we denote as discretized MILP. This discretized MILP is likely to be easier to solve than the original QCQP since the solution methods for MILPs are very advanced and modern MILP solvers are able to solve very large instances quickly. All feasible solutions to the discretized MILP are also feasible to the original QCQP. However, since we restricted the solution space of the original problem by discretization, an optimal solution to the discretized MILP may not be an optimal solution to the original QCQP.

2.2 Selection of Discretized Variables

To apply the discretization from Section 2.1 to general QCQPs, we have to decide which variables in the quadratic terms we discretize. To this end, we consider a graph $G = (V, E)$ where the set of nodes V is equal to the set of variables of the QCQP and the set of edges E



(a) Internal point. **(b)** End point within feasible region. **(c)** End point on boundary.

Figure 1 Adaption of discretization for $u = 5$.

contains an edge $\{x_i, x_j\}$ if and only if the quadratic term $x_i\, x_j$ exists in the QCQP. In this graph, a vertex cover is equivalent to a feasible discretization for the QCQP where at least one variable in each quadratic term is discretized. It is desirable to discretize as few variables as possible because the problem size of the discretized MILP increases with the number of discretized variables. However, since the minimum vertex cover problem is NP-complete, solving it to optimality might become too time consuming for larger instances. Therefore, we apply the greedy heuristic to obtain a good vertex cover in fast running time.

This heuristic creates a vertex cover by successively adding variables to the set D. After initializing the set $D = \emptyset$, we first add all strictly quadratic variables to D and afterwards remove them along with its adjacent edges from G. Then, we iteratively determine the node with the highest degree in the current graph, add it to D if its degree is positive, and remove it from G afterwards. When the graph is empty, the set D consists of a vertex cover, i.e., D represents a feasible discretization for the QCQP. On our instances, this heuristic performed better than other well-known factor 2 approximation algorithms for the vertex cover problem.

2.3 Adaption of Discretization

The quality of the optimal solution of the discretized MILP depends on the chosen discretized values $x_{i1}, \ldots, x_{iu}$ of each discretized variable $x_i \in D$. Using a big discretization size u might lead to better solutions, but the corresponding discretized MILPs get too large and computationally intractable. On the other hand, if we use a reasonable discretization size, it is unlikely to choose an initial discretization that leads to a very good solution for the original problem. Therefore, we use a computationally tractable discretization size u and iteratively adapt the discretization values $x_{i1}, \ldots, x_{iu}$ while keeping the size u fixed. This leads to an iterative algorithm which allows us to improve the discretization quality in each iteration based on the previous solution of the discretized MILP. By keeping the discretized values corresponding to the solution of the previous iteration, the adapted discretized MILPs yield solutions at least as good as the previous one. Moreover, since the discretization size is fixed, we are able to solve each discretized MILP in reasonable running time.

Let us again consider the discretization of a single quadratic term $w_{ij} = x_i\, x_j$ and let $k \in N$ be the selected index in the solution of the previous discretized MILP, i.e., we have $x_i = x_{ik}$ with $z_{ik} = 1$. To adapt the discretization, we consider three different cases depending on x_{ik} as proposed by [13]. First, we consider the case $1 < k < u$ where the previous solution x_{ik} is an internal point of the discretization. Here, we halve the length of the discretization by moving the new discretized values closer around x_{ik}. This procedure is illustrated in Figure 1a where the second discretized value x_{i2} is selected in the solution of the previous discretized MILP. Second, we consider the case $k \in \{1, u\}$ where the previous solution is an end point of the discretization which lies strictly within the feasible region and not on its boundary, i.e., $x_{ik} \in (\ell_i, u_i)$. Then, we shift the discretization towards x_{ik} without reducing

Algorithm 1 Adaption of discretization.

Input: Discretized values x_{in} for $n \in N$, Previous solution z_{in} for $n \in N$
Output: Adapted discretized values x_{in} for $n \in N$

Choose $k \in N$ such that $z_{ik} = 1$
if $1 < k < u$ **then** ▷ Internal point
  $\delta \leftarrow (x_{i2} - x_{i1}) / 2$ ▷ Step size of new discretization
  $m \leftarrow \lceil \frac{u}{2} \rceil$ ▷ New selected index
else if $k \in \{1, u\} \wedge x_{ik} \in (\ell, u)$ **then** ▷ End point within feasible region
  $\delta \leftarrow (x_{i2} - x_{i1})$
  $m \leftarrow \lceil \frac{u}{2} \rceil$
else if $k = 1 \wedge x_{ik} = \ell$ **then** ▷ End point on left boundary
  $\delta \leftarrow (x_{i2} - x_{i1}) / 2$
  $m \leftarrow 1$
else if $k = u \wedge x_{ik} = u$ **then** ▷ End point on right boundary
  $\delta \leftarrow (x_{i2} - x_{i1}) / 2$
  $m \leftarrow u$
end if
while $x_{ik} + (1 - m)\,\delta < \ell$ **do** ▷ New discretization is outside of feasible region
  $m \leftarrow m - 1$ ▷ Shift to the right into feasible region
end while
while $x_{ik} + (u - m)\,\delta > u$ **do** ▷ New discretization is outside of feasible region
  $m \leftarrow m + 1$ ▷ Shift to the left into feasible region
end while
$x_i \leftarrow x_{ik}$ ▷ Store previous selected discretized value
for $n \in N$ **do**
  $x_{in} \leftarrow x_i + (n - m)\,\delta$ ▷ Assign new discretized values
end for

its length. See Figure 1b for a visualization of this adaption where the last point of the discretization is selected. Lastly, we consider the case $k \in \{1, u\}$ where the previous solution is an end point of the discretization which lies on the boundary of the feasible region, i.e., $x_{ik} \in \{\ell_i, u_i\}$. We then halve the length of the discretization by moving the discretized values towards x_{ik} while keeping x_{ik} as an end point of the new discretization. This situation is shown in Figure 1c where x_{ik} lies on the left boundary of the feasible region. In all three cases, we ensure that the new discretization is equidistant and that it still contains the previous solution x_{ik}. If parts of the new discretization are located outside the feasible region, we shift the discretization back into the feasible region by keeping the previous solution x_{ik} in the discretization. A detailed description of the whole adaption can be found in Algorithm 1.

To adapt the whole discretized MILP, we perform the above adaption for all discretized variables $x_i \in D$. Since the solution of the previous problem is also feasible to the adapted discretized MILP, it can be used as MILP warm start for the latter. This guarantees that the adapted discretized MILP yields a solution at least as good as the previous one.

2.4 Iterative Algorithm

Now we present the adaptive discretization algorithm for the calculation of feasible solutions to QCQPs. A flowchart of this algorithm is displayed in Figure 2. We start the algorithm by determining the set of discretized variables D according to Section 2.2 and then choosing

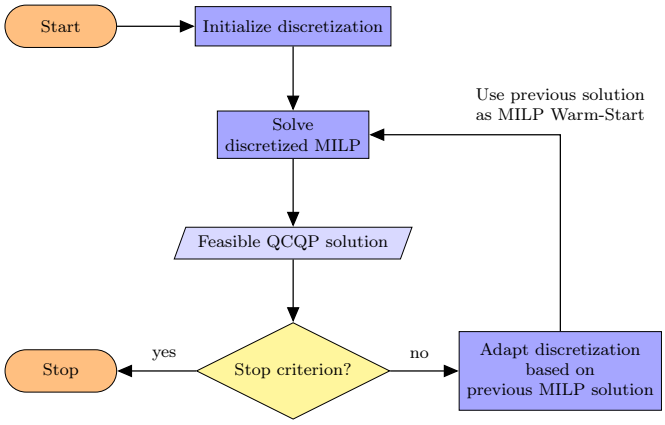


Figure 2: Adaptive discretization algorithm

Figure 2 Adaptive discretization algorithm.

Algorithm 2 Adaptive discretization algorithm.

Input: QCQP instance $\mathcal{T}$, Discretization size K
Output: Feasible solution for $\mathcal{T}$

Determine discretized variables D
for $x_i \in D$ **do** ▷ Initialize discretization
  $\delta \leftarrow (u - \ell)\,/\,(K - 1)$ ▷ Initial step size
  for $n \in N$ **do**
    $x_{in} \leftarrow \ell + (n-1)\delta$ ▷ Assign initial discretized values
  end for
end for
repeat
  Compute solution (x, y, z) for discretized MIQCP ▷ Use MIQCP start
  for $x_i \in D$ **do**
    Adapt discretization of x_i with Algorithm 1 ▷ Use previous solution z
  end for
until Stop criterion is fulfilled

3.2 Mathematical Formulation

A well studied and frequently used formulation for the pooling problem is the *pq*-formulation proposed by [TS02] which uses proportion variables for the inlet flows of pools. In this paper, we consider the *tp*-formulation introduced by [AH13] which is symmetric to the *pq*-formulation, i.e., it has proportion variables for the outlet flows of pools instead for the inlet flows. We use the *tp*-formulation instead of the *pq*-formulation because our computational experiments on the discretization introduced in Section 2.1 yield much better results for the former formulation. We remark that there are also some other formulations that are in a similar vein to *pq* and *tp* and were studied by [BKR16], but for now we do not experiment with all of them.

In the following, we present the *tp*-formulation for the pooling problem. This formulation contains three different kinds of variables. The first kind are the flow variables y_{ij} which are equal to the flow on each arc $(i, j) \in A$ with $i \in I$ and $j \in L \cup J$. Second, the path flow variables z_{ijq} represent the flow along the path from input i via pool l to output j. Finally, for each pool l and output j, the variable q_{lj} is the proportion of flow in arc (l, j) to the total outlet flow of pool l. The whole *tp*-formulation is given by (8)-(20). Its objective is to maximize a linear weight function over the flow variables:

$$\max \sum_{i,j,q} f_{ijq} z_{ijq} = \sum_{(l,j)\in L\times J} (f_{lj} - f_{ij}) y_{lj} + \sum_{i\in I, j\in J} f_{ij} y_{ij} \tag{8}$$

an initial discretization $x_{i1}, \ldots, x_{iu}$ for each discretized variable $x_i \in D$. Usually, this initial discretization covers the whole feasible region, i.e., it has the end points $x_{i1} = \ell_i$ and $x_{iu} = u_i$. Next, we solve the corresponding discretized MILP (see Section 2.1) which yields a first feasible solution to the original QCQP. Then, we check if a stop criterion of the algorithm is fulfilled. If no stop criterion is fulfilled, we enter the iteration loop and adapt the discretization of each discretized variable $x_i \in D$ based on the previous MILP solution as stated in Section 2.3. Subsequently, we solve the new discretized MILP where we pass the solution of the previous problem as MILP start. This speeds up the calculations of the current discretized MILP and ensures that the current solution is at least as good as the previous one. Once we calculated the next feasible solution to the original QCQP, we again check for stop criteria. If one of the stop criteria is now fulfilled, we terminate the algorithm instead of entering the next iteration loop and return the QCQP solution calculated in the last iteration. Algorithm 2 describes the above steps in detail.

Our computational experiments in the next section show that this adaptive discretization algorithm computes high quality solutions in reasonable running time for many instances. One drawback of this algorithm is that no feasibility is guaranteed in the first iteration. It

could occur that the discretized MILP with the initial discretization is infeasible even though the original QCQP is feasible. In this case, one could either increase the discretization size u or modify the discretized MILPs to obtain a relaxation instead of a restriction [17]. We do not address in this paper the question of selecting a suitable discretization that is feasible and leave it as a question for separate work in the future.

3 Computational Study

We present an extensive computational study where we apply the adaptive discretization algorithm to two problem classes of QCQP which we describe in the subsequent sections. In preliminary testing, we did try some arbitrary instances of QCQP from the library QPLib [11]; however, for many of these instances it was not easy to determine a initial discretization that is feasible and hence our algorithm would terminate without computing a primal bound.

The source code for our implementations and the instances we used are both available upon request.

3.1 Algorithms

To evaluate the performance of our adaptive algorithm, we use performance profiles to compare objective values and running times to the commercial global solvers `BARON` and `Gurobi`, which are well-known to be state-of-the-art for solving nonconvex QCQP. We also compared against the popular global solver `SCIP` which employs some MILP-based primal heuristics [2, 3], but it performed much worse than `BARON` and `Gurobi` and our algorithm, and so we do not include `SCIP` in the results reported in this paper. As far as we are aware, `BARON` uses a variety of local solvers to compute feasible solutions to nonlinear problems, whereas `Gurobi` employs some MILP heuristics on a disjunctive formulation for QCQP. Thus, our numerical experiments showcase the advantages of our MILP discretizations not only over local solvers but also over other MILP heuristics.

Our adaptive refinement algorithm is denoted by `AD-`u, using the discretization and adaption from Section 2.1 and Section 2.3, respectively, and where the parameter u represents the size of the discretizations used in the algorithm. We calculate solutions to the original QCQPs with the global solvers `BARON` and `Gurobi`. Beside primal solutions, the global solvers also yield dual bounds which allow us to evaluate the optimality gap of our solutions.

3.2 Implementation

GAMS 31.1.1 is used along with its Python API as the mathematical modeling system. `Gurobi` 9.1.1 is used to solve all discretized MILPs, and it is also used as a global solver for the original QCQP. We also compare against the global solver `BARON` 20.4.14. For all solvers, we use a feasibility tolerance 10^{-6} and an integrality tolerance 10^{-5}. For MILPs solved by `Gurobi`, we set the option `mipstart = 1` and for QCQPs solved by `Gurobi`, we set the option `nonconvex = 2`. All remaining solver options are the standard values. Each calculation is performed on a single core of a Linux machine with an Intel Core i9-9900 CPU with 4.7 GHz clock rate and 32 GB RAM where 14 GB RAM is reserved for this calculation.

For the adaptive discretization algorithms `AD-`u and `ADP-`u, we set a global time limit of 3600 seconds. Furthermore, we specify a time limit of 1200 seconds and a relative optimality gap of 0.01% for each discretized MILP. Besides the global time limit of 3600 seconds, the overall algorithm also stops if the relative improvement of the best solution over the last two iterations is smaller than 0.01%. For the QCQP solvers `BARON` and `Gurobi`, calculations are

stopped if the global time limit of 4 hours or a relative optimality gap of 0.01% is reached. Here, we allow a longer calculation time of 4 hours to obtain good dual bounds as well as to show the difficulties of QCQP solvers to find competitive primal solutions even with this advantage in terms of running time.

3.3 Study I: BoxQPs with Complementarity Constraints

A BoxQP has a quadratic objective and lower and upper bounds on variables as the only constraints. Due to triviality of the constraints, good primal bounds for BoxQP can generally be computed very quickly by local solvers, and so it is of no interest to apply our discretization algorithm to BoxQP directly. Instead, we consider BoxQPs with complementarity constraints

$$\begin{aligned} \text{(BoxQPcc)}: \quad \max_{x} \quad & x^\top Q\, x + c^\top x \\ \text{s.t.} \quad & x_i\, x_j = 0 \quad \forall\, (i,j) \in E, \\ & 0 \le x_i \le 1 \quad \forall\, i \in \{1,\dots,n\}, \end{aligned}$$

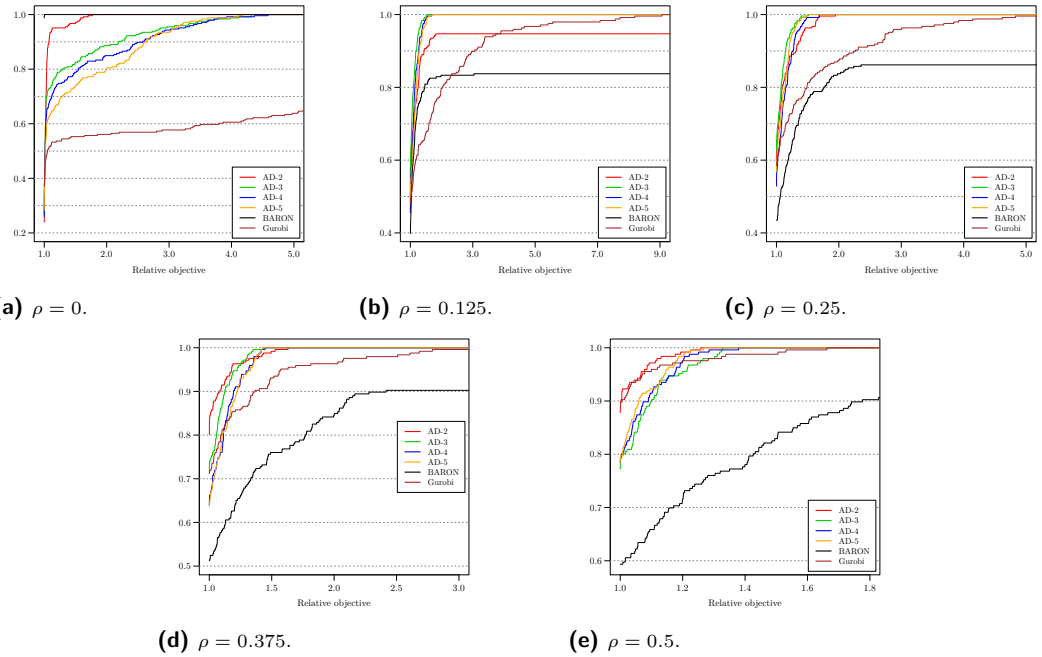
where E is some given subset of the Cartesian product of $\{1,\dots,n\}$ with itself. The complementarity constraints enforce that at least one of the variables x_i and x_j is zero.

Based on the 246 BoxQPs from [21], we generate a test set of 1230 BoxQPcc instances. Let $\rho \in \{0, 0.125, 0.25, 0.375, 0.5\}$ be a fixed probability and $\mathcal{T}$ be one of the BoxQP instances. Then, we create a new BoxQPcc instance $\mathcal{T}_\rho$ by adding for each quadratic term $x_i\, x_j$ that occurs in the objective function of $\mathcal{T}$ the constraint $x_i\, x_j = 0$ with probability ρ. This means that the instances $\mathcal{T}_0$ are equal to the original 246 BoxQP instances while the remaining instances $\mathcal{T}_\rho$ for $\rho \in \{0.125, 0.25, 0.375, 0.5\}$ have additional quadratic complementarity constraints. We only consider probabilities $\rho \in \{0, 0.125, 0.25, 0.375, 0.5\}$ in this study since higher probabilities yield instances with a very high density of complementarity constraints which force many quadratic terms to zero and thus make the instances very easy for most algorithms.

Now we compare the adaptive discretization algorithm `AD-`u for $u \in \{2, 3, 4, 5\}$ with the global solvers `BARON` and `Gurobi`. Since the gaps for most of the BoxQPcc instances are relatively large, performance profiles using relative optimality gaps are not very informative for these instances. Therefore, we only present performance profiles with relative objective values for the BoxQPcc instances.

Figure 3a shows the performance profile with relative objective values for the BoxQPcc instances with $\rho = 0$, i.e., the original BoxQP instances. We see that `BARON` calculates by far the strongest objectives with the best objective values for 99% of the instances. The second best results are achieved by `AD-2` with objectives at most 5% worse than the best for 88% of the instances and all instances at most 77% worse than the best solutions. The remaining adaptive discretization algorithms perform even weaker and only manage to solve all instances with objective values at most 5 times worse than the best. `Gurobi` achieves clearly the weakest results by terminating for 65% of the instances with objectives more than 5 times worse than the best solutions.

A performance profile with relative objective values for the BoxQPcc instances with $\rho = 0.125$ is presented in Figure 3b. Here, the results look very different as the adaptive discretization algorithms `AD-3`, `AD-4`, and `AD-5` perform clearly best and outclass all remaining algorithms. `AD-2` yields weaker objectives than the other discretization algorithms but is still much stronger than `BARON` and `Gurobi` for up to 95% of the instances. `Gurobi` only calculates better objective values than `AD-2` for percentages between 95% and 100%. While the performance of `BARON` has heavily dropped, `Gurobi` achieves stronger results as for $\rho = 0$.



(a) $\rho = 0$. **(b)** $\rho = 0.125$. **(c)** $\rho = 0.25$.

(d) $\rho = 0.375$. **(e)** $\rho = 0.5$.

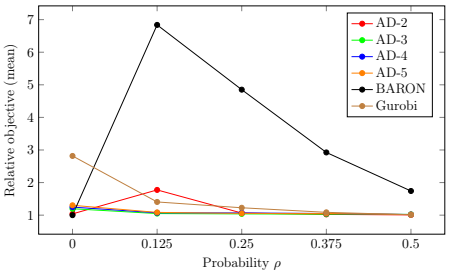
Figure 3 Relative objective values of AD-u and QCQP solvers for BoxQPcc instances.

For $\rho = 0.25$, a corresponding performance profile is depicted in Figure 3c. The adaptive discretization algorithms still yield the best objectives with their results relatively close to each other and AD-3 being the strongest and AD-2 being the weakest. However, AD-2 is now very close behind the remaining discretization algorithms and beats both QCQP solvers by far. While BARON and Gurobi are beaten by magnitudes, Gurobi now has clearly stronger objective values than BARON.

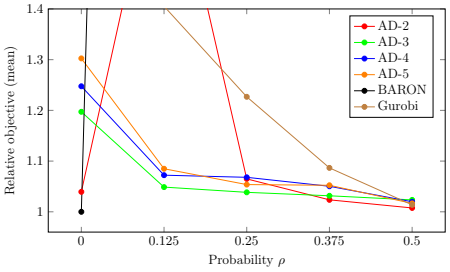
Figure 3d shows the performance profile with relative objective values for the BoxQPcc instances with $\rho = 0.375$. While the adaptive discretization algorithms are still on top, Gurobi is now relatively close behind them and is even stronger than AD-4 and AD-5 for less than 80% of the instances. AD-2 and AD-3 reach the best results where the former is even able to find the best solution for 80% of the instances. The discretization algorithms terminate for all instances with objective values at most 1.63 times worse than the best solution while Gurobi achieves this only with objective values more than 3 times worse than the best. BARON is beaten by magnitudes by all other algorithms.

The performance profile for the BoxQPcc instances with $\rho = 0.5$ is illustrated in Figure 3e. Here, only AD-2 performs stronger than Gurobi for all percentages of instances while the remaining discretization algorithms are weaker than Gurobi for less than 97% of the instances and outperform Gurobi for 97% to 100% of the instances. While Gurobi can only guarantee a relative objective value of at most 66% worse than the best, all discretization algorithms achieve achieve a value of less than 40% while AD-2 even guarantees objective values at most 26% worst than the best ones. Again, BARON is outclassed by all other algorithms.

The above five performance profiles are summarized in Figure 4a. This figure shows for each considered probability ρ the geometric mean of the relative objective values of the corresponding 246 instances calculated by each algorithm. We see that the adaptive discretization algorithms AD-3, AD-4, and AD-5 yield the most consistent results. They achieve top results for all probabilities except $\rho = 0$ where they are beaten by AD-2 and BARON. AD-2 is also very strong for most probabilities but fails for $\rho = 0.125$. While BARON



(a) All profiles.



(b) Zoomed summary.

Figure 4 Summary of relative objective values of AD-$\overline{u}$ and QCQP for BoxQPCC instances.

only achieves competitive results for $\rho = 0$, Gurobi gets stronger with increasing probabilities with competitive objective values for $\rho = 0.5$. To evaluate the performance of the individual adaptive discretization algorithms, we show the same figure zoomed to smaller relative objective values in Figure 4b. From this figure follows that AD-3 overall performs most consistent among the adaptive discretization algorithms. AD-4 and AD-5 yield slightly weaker results than AD-3 while AD-2 performs very unstable. On the one hand, AD-2 beats the remaining discretization algorithms for $\rho \in \{0, 0.375, 0.5\}$ and, on the other hand, it performs poorly for $\rho = 0.125$.

3.4 Study II: Disjointly Constrained Bilinear Programs

Another subclass of QCQP that we consider are disjoint bilinear programs

$$\text{(DisjBLP)}: \quad \max_{x,y} \quad x^\top Q\, y + c^\top x + d^\top y \qquad \text{s.t.} \quad A\, x \le a, \qquad B\, y \le b$$

with variables $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$ and inputs $Q \in \mathbb{R}^{n \times m}$, $c \in \mathbb{R}^n$, $d \in \mathbb{R}^m$, $A \in \mathbb{R}^{r \times n}$, $B \in \mathbb{R}^{s \times m}$, $a \in \mathbb{R}^r$ and $b \in \mathbb{R}^s$. These are called *disjoint* because they are bipartite due to the variables being partitioned into two sets x and y, and the feasible set is the Cartesian product of a polyhedron in x-space and a polyhedron in y-space.

We generate 60 DisjBLP test instances following the procedure in [26]. For all generated instances, we use the parameters $\delta = 2.5$, $\rho = 1.5$, and randomized them with Householder matrices defined by unit vectors with random numerators and a denominator of 1000. The size of the instances depends on the parameters κ_1 and κ_2 where the total number of variables and constraints is equal to $\kappa_1 + 3\,\kappa_2$ and $\kappa_1 + 5\,\kappa_2$. Here, we generate six classes of DisjBLP instances where each class contains 10 instances with size $\kappa_1 \in \{50, 100, 150, 200, 300, 400\}$ and $\kappa_2 = 2\,\kappa_1$. For each instance, we used the random seed $\kappa_1 + \kappa_2 + \delta + \rho + k$ where $k \in \{0, \ldots, 9\}$ is the number of the instance. After the instance generation, we applied LP based bound tightening on all variables. Moreover, we discovered that the kernel problem 2 described by [26] does not have the optimal solution stated by them in Property 6. Therefore, we added the constraint $y \ge 1$ to kernel problem 2 to fix this issue.

The adaptive algorithm AD-u for $u \in \{5, 6\}$ is compared with the QCQP solvers BARON and Gurobi on the 60 generated instances. Here, we only use greater discretization sizes since the smaller sizes $u \in \{2, 3, 4\}$ struggle to find competitive feasible solutions for many instances due to the randomized bounds on the variables. Figure 5 presents a performance profile with the relative optimality gaps of the above algorithms. This figure shows that

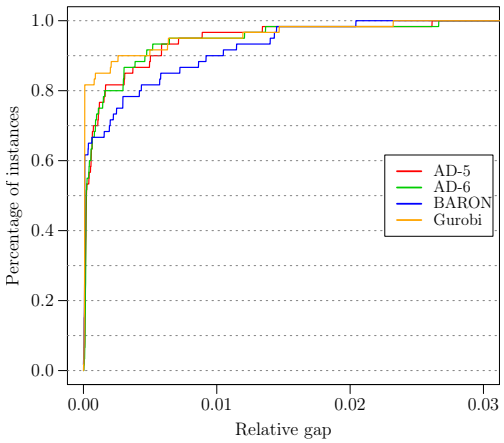


Figure 18: Relative gaps of AD-5, AD-6, and the QCQP solvers for DisjBLP instances

Figure 5 Relative gaps of `AD-5`, `AD-6`, and the QCQP solvers for DisjBLP instances.

all considered algorithms perform relatively similar on the DisjBLP instances as the scale of the relative gaps only reaches to 3%. `Gurobi` yields slightly better objective values than the rest while `BARON` yields slightly worse. The adaptive discretization algorithms `AD-5` and `AD-6` show very similar results and are even able to compete with `Gurobi` for above 90% of the instances. On the other hand, `BARON` is able to terminate for all instances with the best gap of at most 2% while the other algorithms are very close behind. Altogether, these four algorithms perform very similar on the DisjBLP instances where `Gurobi` calculates marginally better objective values than the rest.

4 Conclusion

We presented an iterative algorithm that adaptively refines a MILP restriction of a QCQP. This restriction arises from discretizing all quadratic terms, and our adaptive step modifies the discretization of the MILP after each iteration based on the previous MILP solution. During this adaption, we only change the discretization values while the discretization sizes remain the same. Since arbitrary instances of QCQP are not always amenable to a MILP discretization approach due to the difficulty of finding a good feasible discretization to begin with, for our computational testing, we chose two problem classes of QCQP. On a large test set of 1230 instances of box-constrained quadratic programs with complementarities, we showed that our adaptive discretization algorithm calculates much better objective values than the heuristics employed in global solvers `BARON` and `Gurobi`. For 60 test instances of disjointly constrained BLPs, the solutions obtained by all methods were of similar value with a slight advantage for `Gurobi`.

References

1 Mohammed Alfaki and Dag Haugland. Comparison of discrete and continuous models for the pooling problem. In Alberto Caprara and Spyros Kontogiannis, editors, *11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 112–121, 2011. `doi:10.4230/OASIcs.ATMOS.2011.112`.

2 Timo Berthold. RENS. *Mathematical Programming Computation*, 6(1):33–54, 2014. `doi:10.1007/s12532-013-0060-9`.

[BHH20] B. Beach, R. Hildebrand, and J. Huchette. *Compact mixed-integer programming relaxations in quadratic optimization*. arXiv Preprint. 2020. arXiv: 2011.08823 [math.OC].

[BGN09] A. Ben-Tal, L. E. Ghaoui, and A. Nemirovski. *Robust Optimization*. Princeton Series In Applied Mathematics. Princeton University Press, 2009.

[BH96] J. M. Bloemhof-Ruwaard and E.M. Hendrix. "Generalized bilinear programming: An application in farm management". In: *European Journal of Operational Research* 90.1 (1996), pp. 102–114.

[BKK16] N. Boland, J. Kalinowski, and [illegible]

[BL02] S. Burer and [illegible]

[BGL18] P. Bonami, O. Günlük, and J. Linderoth. "Globally solving nonconvex quadratic programming problems with box constraints via integer programming methods". In: *Mathematical Programming Computation* 10.3 (2018), pp. 333–382.

[BKL+19] P. Bonami, M. Kılınç, J. Linderoth, and A. Lodi. "Solving quadratic programming by cutting planes". In: *SIAM Journal on Optimization* 29.2 (2019), pp. 1076–1105.

[BKK14] S. Burer, S. Kim, and M. Kojima. "Faster, but weaker, relaxations for quadratically constrained quadratic programs". In: *Computational Optimization and Applications* 59.1-2 (2014), pp. 27–45.

3 Timo Berthold and Ambros M Gleixner. Undercover: a primal MINLP heuristic exploring a largest sub-MIP. *Mathematical Programming*, 144(1-2):315–346, 2014. doi:10.1007/s10107-013-0635-2.

4 Endre Boros, Peter L Hammer, and Gabriel Tavares. Local search heuristics for quadratic unconstrained binary optimization (QUBO). *Journal of Heuristics*, 13(2):99–132, 2007.

5 Samuel Burer and Anureet Saxena. The MILP road to MIQCP. In Jon Lee and Sven Leyffer, editors, *Mixed Integer Nonlinear Programming*, volume 154 of *IMA Volumes in Mathematics and its Applications*, pages 373–405. Springer, 2012.

6 Robert Burlacu, Björn Geißler, and Lars Schewe. Solving mixed-integer nonlinear programmes using adaptively refined mixed-integer linear programmes. *Optimization Methods and Software*, 35(1):37–64, 2020.

7 C. D'Ambrosio, A. Frangioni, L. Liberti, and A. Lodi. Experiments with a feasibility pump approach for nonconvex MINLPs. In P. Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 350–360. Springer, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-13193-6_30.

8 Claudia D'Ambrosio, Antonio Frangioni, Leo Liberti, and Andrea Lodi. A storm of feasibility pumps for nonconvex MINLP. *Mathematical Programming*, 136(2):375–402, 2012. doi:10.1007/s10107-012-0608-x.

9 Sanjeeb Dash, Oktay Günlük, and Robert Hildebrand. Binary extended formulations of polyhedral mixed-integer sets. *Mathematical Programming*, 170(1):207–236, 2018. doi:10.1007/s10107-018-1294-0.

10 Santanu S Dey and Akshay Gupte. Analysis of MILP techniques for the pooling problem. *Operations Research*, 63(2):412–427, 2015. doi:10.1287/opre.2015.1357.

11 Fabio Furini, Emiliano Traversi, Pietro Belotti, Antonio Frangioni, Ambros Gleixner, Nick Gould, Leo Liberti, Andrea Lodi, Ruth Misener, Hans Mittelmann, et al. QPLIB: a library of quadratic programming instances. *Mathematical Programming Computation*, 11(2):237–265, 2019.

12 Laura Galli and Adam N Letchford. A binarisation heuristic for non-convex quadratic programming with box constraints. *Operations Research Letters*, 46(5):529–533, 2018.

13 S. Goderbauer, B. Bahl, P. Voll, M.E. Luebbecke, A. Bardow, and A.M.C.A. Koster. An adaptive discretization MINLP algorithm for optimal synthesis of decentralized energy supply systems. *Computers & Chemical Engineering*, 95:38–48, 2016. doi:10.1016/j.compchemeng.2016.09.008.

14 Akshay Gupte, Shabbir Ahmed, Myun S. Cheon, and Santanu S Dey. Solving mixed integer bilinear problems using MILP formulations. *SIAM Journal on Optimization*, 23(2):721–744, 2013. doi:10.1137/110836183.

15 Akshay Gupte, Shabbir Ahmed, Santanu S. Dey, and Myun Seok Cheon. Relaxations and discretizations for the pooling problem. *Journal of Global Optimization*, 67(3):631–669, 2017. doi:10.1007/s10898-016-0434-4.

16 Scott P Kolodziej, Ignacio E Grossmann, Kevin C Furman, and Nicolas W Sawaya. A discretization-based approach for the optimization of the multiperiod blend scheduling problem. *Computers & Chemical Engineering*, 53:122–142, 2013.

17 Arie M. C. A. Koster and Sascha Kuhnke. An adaptive discretization algorithm for the design of water usage and treatment networks. *Optimization and Engineering*, 20(2):497–542, June 2019. doi:10.1007/s11081-018-9413-6.

18 Leo Liberti, Nenad Mladenović, and Giacomo Nannicini. A recipe for finding good solutions to MINLPs. *Mathematical Programming Computation*, 3(4):349–390, 2011. doi:10.1007/s12532-011-0031-y.

19 Harsha Nagarajan, Mowen Lu, Site Wang, Russell Bent, and Kaarthik Sundar. An adaptive, multivariate partitioning algorithm for global optimization of nonconvex programs. *Journal of Global Optimization*, 74(4):639–675, 2019.

20 Giacomo Nannicini and Pietro Belotti. Rounding-based heuristics for nonconvex MINLPs. *Mathematical Programming Computation*, 4(1):1–31, 2012. doi:10.1007/s12532-011-0032-x.

21 Carlos J Nohra, Arvind U Raghunathan, and Nikolaos Sahinidis. Spectral relaxations and branching strategies for global optimization of mixed-integer quadratic programs. *SIAM Journal on Optimization*, 31(1):142–171, 2021.

22 Foad Mahdavi Pajouh, Balabhaskar Balasundaram, and Oleg A Prokopyev. On characterization of maximal independent sets via quadratic optimization. *Journal of Heuristics*, 19(4):629–644, 2013. doi:10.1007/s10732-011-9171-5.

23 Jaehyun Park and Stephen Boyd. General heuristics for nonconvex quadratically constrained quadratic programming. arXiv Preprint, May 2017. arXiv:1703.07870.

24 V. Pham, C. Laird, and M. El-Halwagi. Convex hull discretization approach to the global optimization of pooling problems. *Industrial and Engineering Chemistry Research*, 48(4):1973–1979, 2009.

25 Wim van Ackooij, Welington de Oliveira, and Yongjia Song. Adaptive partition-based level decomposition methods for solving two-stage stochastic programs with fixed recourse. *INFORMS Journal on Computing*, 30(1):57–70, 2018. doi:10.1287/ijoc.2017.0765.

26 Luis N Vicente, Paul H Calamai, and Joaquim J Júdice. Generation of disjointly constrained bilinear programming test problems. *Computational Optimization and Applications*, 1(3):299–306, 1992.