18th International Symposium on Experimental Algorithms

SEA 2020, June 16–18, 2020, Catania, Italy

Edited by

Simone Faro

Domenico Cantone

LIPIcs – Vol. 160 – SEA 2020 www.dagstuhl.de/lipics

Editors

Simone Faro
University of Catania, Italy
faro@dmi.unict.it

Domenico Cantone
University of Catania, Italy
domenico.cantone@unict.it

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

ISBN 978-3-95977-148-1

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/978-3-95977-148-1.

Publication date
June, 2020

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.2020.0

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

18th International Symposium on Experimental Algorithms (SEA 2020).
Editors: Simone Faro and Domenico Cantone
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

We proudly present the collection of the papers accepted for presentation at the 18th edition of the International Symposium on Experimental Algorithms (SEA 2020), originally planned to be held in Catania (Italy) from June 16 to 18, 2020. Unfortunately, since March 2020, the situation surrounding the COVID19 outbreak has evolved very rapidly also outside of China, endangering the health and safety of everyone, which are our highest priorities. Thus, based on the current health situation in Italy and in the rest of world, the leadership of SEA 2020 has decided that the work of this year symposium will be delivered online to all registered attendees, rather than by the planned physical meeting.

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.

The symposium aims at attracting papers from both the Computer Science and the Operations Research/Mathematical Programming communities. The main theme of the symposium is the role of experimentation and of algorithm engineering techniques in the design and evaluation of algorithms and data structures. 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 to SEA 2020 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 52%.

The 18th edition of SEA was organized by the University of Catania. We thank our university and, more specifically, our Department of Mathematics and Computer Science for their support. We also thank the "Gruppo Nazionale per il Calcolo Scientifico" (GNCS/Indam) for its support in the organization of the event. 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 2020. Finally, we express our gratitude to the members of the Program Committee for their support, collaboration, and very good work. Finally, we express our gratitude to the EasyChair platform.

Catania, June 11 2020
Simone Faro and Domenico Cantone

18th International Symposium on Experimental Algorithms (SEA 2020).
Editors: Simone Faro and Domenico Cantone
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)

18th International Symposium on Experimental Algorithms (SEA 2020).
Editors: Simone Faro and Domenico Cantone

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

Organization

Program Chairs

- Domenico Cantone, University of Catania (Italy)
- Simone Faro, University of Catania (Italy)

Program Committee

- Golnaz Badkobeh, Goldsmiths University of London (UK)
- Gianfranco Bilardi, University of Padova (Italy)
- Christina Boucher, University of Florida (USA)
- Pierluigi Crescenzi, Université de Paris-IRIF (France)
- Maxime Crochemore, Kings College London (UK)
- Paola Festa, University of Naples Federico II (Italy)
- Irene Finocchi, Sapienza University of Rome (Italy)
- Travis Gagie, Dalhousie University (Canada)
- Arie Koster, RWTH Aachen University (Germany)
- Oguzhan Kulekci, Istanbul Technical University (Turkey)
- Susana Ladra, University of A Coruña (Spain)
- Thierry Lecroq, University of Rouen Normandy (France)
- Veli Mäkinen, University of Helsinki (Finland)
- Petra Mutzel, TU Dortmund (Germany)
- Gonzalo Navarro, University of Chile (Chile)
- Panos Pardalos, University of Florida (USA)
- Nadia Pisanti, University of Pisa (Italy)
- Ely Porat, Bar-Ilan University (Israel)
- Simon J. Puglisi, University of Helsinki (Finland)
- Ilya Razenshteyn, Microsoft Research (USA)
- Mauricio Resende, Amazon.com Inc. (USA)
- Marie-France Sagot, INRIA (France)
- Alessandra Sala, Bell Labs (Ireland)
- Peter Sanders, Karlsruhe Institute of Technology (Germany)
- Stefan Schmid, University of Vienna (Austria)
- Sabine Storandt, Universität Konstanz (Germany)
- Dorothea Wagner, Karlsruhe Institute of Technology (Germany)
- Renato Werneck, Amazon.com Inc. (USA)

External Reviewers

Konstantina Mellou, Adrián Gómez Brandón, Alejandro Pacheco, Alessio Conte, Andrea Marino, Antoine Limasset, Carlos Ochoa, Christian Schulz, Daniele Ferone, Daniele Ferone, Daniil Galaktionov, Demian Hespe, Diego Arroyuelo, Diego Diaz, Dustin Cobas, Fritz Bökler, Giulia Bernardini, Guillaume Hanrot, Javiel Rojas-Ledesma, Johannes Blum, Jonas Sauer, Keisuke Goto, Kunihiro Wasa, Lars Gottesbüren, Marcel Radermacher, Marcin Pilipczuk, Michael Rice, Michał Gańczorz, Mohammad Malekzadeh, Moritz Beck, Nicola Prezza, Oded Lachish, Pablo Rotondo, Pascal Ochem, Pierre Peterlongo, Rajeev Raman, Robert Gwadera, Sascha Gritzbach, Tal Wagner, Tim Zeitz, Tobias Zündorf, Valentin Buchhold, Vincent Limouzy.

18th International Symposium on Experimental Algorithms (SEA 2020).
Editors: Simone Faro and Domenico Cantone
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Algorithm Engineering for High-Dimensional Similarity Search Problems

Martin Aumüller

IT University of Copenhagen, Denmark
maau@itu.dk

Abstract

Similarity search problems in high-dimensional data arise in many areas of computer science such as data bases, image analysis, machine learning, and natural language processing. One of the most prominent problems is finding the k nearest neighbors of a data point $q \in \mathbb{R}^d$ in a large set of data points $S \subset \mathbb{R}^d$, under same distance measure such as Euclidean distance. In contrast to lower dimensional settings, we do not know of worst-case efficient data structures for such search problems in high-dimensional data, i.e., data structures that are faster than a linear scan through the data set. However, there is a rich body of (often heuristic) approaches that solve nearest neighbor search problems much faster than such a scan on many real-world data sets. As a necessity, the term *solve* means that these approaches give approximate results that are close to the true k-nearest neighbors. In this talk, we survey recent approaches to nearest neighbor search and related problems.

The talk consists of three parts: (1) What makes nearest neighbor search difficult? (2) How do current state-of-the-art algorithms work? (3) What are recent advances regarding similarity search on GPUs, in distributed settings, or in external memory?

2012 ACM Subject Classification Theory of computation → Nearest neighbor algorithms; Computing methodologies → Simulation evaluation; Theory of computation → Computational geometry

Keywords and phrases Nearest neighbor search, Benchmarking

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.1

Category Invited Talk

1 Difficulty of nearest neighbor search (NNS)

In the first part of the talk, we give an overview over the general setting of high-dimensional NNS. We look at general observations about the difficulty of high-dimensional NNS, a phenomenon usually referred to as the "concentration of distances" [5]. This for example happens if we draw data and query points at random from a d-dimensional standard normal distribution $\mathcal{N}(0,1)^d$; for large d, the distance to the furthest neighbor will be very close to the distance of the nearest neighbor. Next, we look at the benchmarking tool `http://ann-benchmarks.com` [2]. We observe that there exist many different approaches that solve NNS with almost perfect quality – in terms of the fraction of true nearest neighbors found on average – with a query time that is several orders of magnitude faster than a linear scan through the dataset. Before looking at these approaches in detail, we survey approaches to measure the difficulty of solving a NNS task discussing the work of [3, 10].

2 Current state-of-the-art approaches to NNS

In the second part of the talk, we explore the landscape of NNS implementations. The focus of the discussion will be on hashing-based approaches using locality-sensitive hashing and graph-based approaches.

Approaches using locality-sensitive hashing [12] are excellent candidates for designing theoretical sound data structures for NNS. In a nutshell, a locality-sensitive hash function makes it possible to map a data point to a number, such that the smaller the distance between the points, the more likely it is that they are mapped to the same number. The talk puts an emphasis on solving k-NNS using a recent adaptive query algorithm on an LSH data structure proposed in [4], building on the general idea of adaptive query algorithms discussed in [1]. We review the details of the query algorithm and engineering choices on the way, such as hash functions pools and sketches as described in [6], and fast distance estimation using AVX instructions.

In contrast to the theoretical guarantees that can be achieved using LSH approaches, graph-based approaches give little theoretical guarantees (a notable exception is the analysis in [15]). Nonetheless, implementations following this approach dominate performance benchmarks. The idea of a graph-based approach is to consider each data point in the data set as a node in a graph. An edge (u, v) represents that v is among the nearest neighbors of u. Given a query point, a greedy search – usually starting with a random start node – is performed to obtain a candidate set of nearest neighbors. Graph-based approaches have several complications that need to be addressed. For example, in high-dimensional space we usually find a very skewed degree distribution which leads to large hubs that make the search slow [19]. Another problem lies in the random starting node: finding a good neighborhood of nodes requires the insertion of long-range edges and diversification of node neighborhoods. The talk surveys recent approaches addressing these issues [13, 16].

As a potential starting point for future engineering work, we further discuss the recent machine learning approach [7] that considers NNS as a learning problem and proposes to use a learned index in the style of [14].

3 Similarity search problems on the GPU, in external memory, and in distributed settings

In the last part of the talk, we survey recent progress on similarity search problems in other areas. We discuss how graph-based algorithms are made to run efficiently on GPUs [9] and in external memory [20]. Finally, we discuss recent work in the context of *similarity joins*, in which we are given two datasets $R, S \subset \mathbb{R}^d$ and want to emit all pairs $(u, v) \in R \times S$ such that the similarity between u and v is above a given threshold value. This problem is of particular interest, because a recent survey [8] found that distributed implementations using MapReduce on a small cluster are usually worse than non-distributed implementations working on a single core [17]. We will discuss general difficulties of similarity joins and take a look at a recent near-optimal solutions based on LSH [11, 18].

References

1 Thomas D. Ahle, Martin Aumüller, and Rasmus Pagh. Parameter-free locality sensitive hashing for spherical range reporting. In *SODA*, pages 239–256. SIAM, 2017.

2 Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.*, 87, 2020. See https://arxiv.org/abs/1807.05614 for an open access version.

3 Martin Aumüller and Matteo Ceccarello. The role of local intrinsic dimensionality in benchmarking nearest neighbor search. In *SISAP*, volume 11807 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2019.

4 Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. PUFFINN: parameterless and universally fast finding of nearest neighbors. In *ESA*, volume 144 of

LIPIcs, pages 10:1–10:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. URL: `https://github.com/puffinn/puffinn`.

5 Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *ICDT*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235. Springer, 1999.

6 Tobias Christiani. Fast locality-sensitive hashing frameworks for approximate near neighbor search. In *SISAP*, volume 11807 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2019.

7 Yihe Dong, Piotr Indyk, Ilya Razenshteyn, and Tal Wagner. Learning space partitions for nearest neighbor search. In *International Conference on Learning Representations*, 2020.

8 Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. Set similarity joins on MapReduce: An experimental survey. *PVLDB*, 11(10):1110–1122, 2018.

9 Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik Lensch. Ggnn: Graph-based gpu nearest neighbor search. *arXiv preprint arXiv:1912.01059*, 2019.

10 Junfeng He, Sanjiv Kumar, and Shih-Fu Chang. On the difficulty of nearest neighbor search. In *ICML*. icml.cc / Omnipress, 2012.

11 Xiao Hu, Ke Yi, and Yufei Tao. Output-optimal massively parallel algorithms for similarity joins. *ACM Transactions on Database Systems*, 44(2):1–36, April 2019. `doi:10.1145/3311967`.

12 Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM. `doi:10.1145/276698.276876`.

13 M. Iwasaki and D. Miyazaki. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. *ArXiv e-prints*, October 2018. `arXiv:1810.07355`.

14 Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

15 Thijs Laarhoven. Graph-based time-space trade-offs for approximate near neighbors. In *Symposium on Computational Geometry*, volume 99 of *LIPIcs*, pages 57:1–57:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

16 Yury A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, 2020.

17 Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. An empirical evaluation of set similarity join techniques. *Proceedings of the VLDB Endowment*, 9(9):636–647, May 2016. `doi:10.14778/2947618.2947620`.

18 Samuel McCauley and Francesco Silvestri. Adaptive MapReduce similarity joins. In *BeyondMR@SIGMOD*, pages 4:1–4:4. ACM, 2018.

19 Milos Radovanovic. Hubs in nearest-neighbor graphs: Origins, applications and challenges. In *WIMS*, pages 5:1–5:4. ACM, 2018.

20 Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishnaswamy, and Harsha Simhadri. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. In *NeurIPS 2019*, November 2019. URL: `https://www.microsoft.com/en-us/research/publication/diskann-fast-accurate-billion-point-nearest-neighbor-search-on-a-single-node/`.

Algorithm Engineering for Sorting and Searching, and All That

Stefan Edelkamp
University of Koblenz, Postfach 201 602, 56016 Koblenz, Germany
edelkamp@uni-koblenz.de

Abstract

We look at several proposals to engineer the set of fundamental searching and sorting algorithms. Aspects are improving locality of disk access and cache access, the efficiency tuning by reducing the number of branch mispredictions, and reducing at leading factors hidden in the Big-Oh notation. These studies in algorithm engineering, in turn, lead to exiting new algorithm designs. On the practical side, we will establish that efficient sorting and searching algorithms are in tight collaboration, as sorting is used for finding duplicates in disk-based search, and heap structures designed for efficient graph search can be exploited in classical and adaptive sorting. We indicate the effects of engineered sorting and searching for combined task and motion planning.

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

Keywords and phrases Sorting, Searching, Algorithm Engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.2

Category Invited Talk

Acknowledgements I want to thank all co-authors of my papers.

1 Introduction

Several decades ago, in the early days of computer science Donald E. Knuth and Kurt Mehlhorn both dedicated a book of their monographs to the topic of sorting and searching. Since then, driven by the advances in computer hardware technology there have been several proposals to engineer the set of fundamental algorithms. One aspect we look at is improving locality of disk access and cache access, another one efficiency tuning by reducing the number of branch mispredictions. We also will look at leading factors hidden in Landau's Big-Oh notation to study how far the algorithms are from their respective lower bounds. We highlight one result in sorting and one in searching, as well as one possible application area.

2 Sorting

QuickXsort is a highly efficient in-place sequential sorting scheme that mixes Hoare's Quicksort algorithm with X, where X can be chosen from a wider range of other known sorting algorithms, like Heapsort, Insertionsort and Mergesort. Its major advantage is that QuickXsort can be in-place even if X is not. We provide general transfer theorems expressing the number of comparisons of QuickXsort in terms of the number of comparisons of X. More specifically, if pivots are chosen as medians of (not too fast) growing size samples, the average number of comparisons of QuickXsort and X differ only by $o(n)$-terms. For median-of-k pivot selection for some constant k, the difference is a linear term whose coefficient we compute precisely. For instance, median-of-three QuickMergesort uses at most $n \lg n - 0.8358n + O(\log n)$ comparisons. Furthermore, we examine the possibility of sorting base cases with some other algorithm using even less comparisons. By doing so the average-case number of comparisons can be reduced down to $n \lg n - 1.4112n + o(n)$ for a remaining gap of only $0.0315n$ comparisons

to the known lower bound (while using only $O(\log n)$ additional space and $O(n \log n)$ time overall). Implementations of these sorting strategies show that the algorithms challenge well-established library implementations like Musser's Introsort.

3 Searching

A priority queue – a data structure supporting, the operations minimum (top), insert (push), and extract-min (pop) – is said to operate in-place if it uses O(1) extra space in addition to the n elements stored at the beginning of an array. Prior to this work, no in-place priority queue was known to provide worst-case guarantees on the number of element comparisons that are optimal up to additive constant terms for both insert and extract-min. In particular, for the standard implementation of binary heaps, insert and extract-min operate in logarithmic time while involving at most $\lceil \lg n \rceil$ and $2 \lg n$ [could possibly be reduced to $\lg \lg n + O(1)$ and $\lg n + \log^* n + O(1)$] element comparisons, respectively. We propose a variant of a binary heap that operates in-place, executes minimum and insert in $O(1)$ worst-case time, and extract-min in $O(\lg n)$ worst-case time while involving at most $\lg n + O(1)$ element comparisons. These efficiencies surpass lower bounds known for binary heaps, thereby resolving a long-standing theoretical debate.

4 Application

Logistics operations often require a robot to pickup and deliver objects from multiple locations within certain time frames. This is a challenging task-and-motion planning problem as it intertwines logical and temporal constraints about the operations with geometric and differential constraints related to obstacle avoidance and robot dynamics. To address these challenges, we couple vehicle routing over a discrete abstraction with sampling-based motion planning. On the one hand, vehicle routing provides plans to effectively guide sampling-based motion planning as it explores the vast space of feasible motions. On the other hand, motion planning provides feasibility estimates which vehicle routing uses to refine its plans. This coupling makes it possible to extend the state-of-the-art in multi-goal motion planning by also incorporating capacities, pickups, and deliveries in addition to time windows. When not all pickups and deliveries can be completed in time, the approach seeks to minimize the violations and maximize the profit.

5 Conclusion

We passed by various priority queue designs for pimping up shortest path search and recent mixtures of sorting algorithms that show both outstanding theoretical and practical performances. On the theoretical side we introduced an in-place heap, for which minimum and insert take $O(1)$ worst-case time, and extract-min takes $O(\lg n)$ worst-case time and involves at most $\lg n + O(1)$ element comparisons. We established an algorithm which in a sequence of n min-deletes and m decrease-keys requires $2m + 1.5n \lg n + o(n)$ comparisons. In sorting, the average-case number of comparisons can be reduced down to $n \lg n - 1.4112n + o(n)$ for a remaining gap of only $0.0315n$ comparisons to the known lower bound, while using only $O(\lg n)$ additional space and $O(n \lg n)$ time overall.

On the practical side, we established that efficient sorting and searching algorithms are in tight collaboration, as sorting is used for finding duplicates in disk-based search, and new heap structures designed for efficient graph search can be exploited in classical and adaptive sorting. We indicated the effects engineered sorting and searching for combined task and motion planning in robotics.

References

1 Stefan Edelkamp. External-memory state space search. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *LNCS*, pages 185–225. Springer, 2016. doi:10.1007/978-3-319-49487-6_6.

2 Stefan Edelkamp. Improving the cache-efficiency of shortest path search. In Gabriele Kern-Isberner, Johannes Fürnkranz, and Matthias Thimm, editors, *KI 2017*, volume 10505 of *LNCS*, pages 99–113. Springer, 2017. doi:10.1007/978-3-319-67190-1_8.

3 Stefan Edelkamp and Tristan Cazenave. Improved diversity in nested rollout policy adaptation. In Gerhard Friedrich, Malte Helmert, and Franz Wotawa, editors, *KI 2016*, volume 9904 of *LNCS*, pages 43–55. Springer, 2016. doi:10.1007/978-3-319-46073-4_4.

4 Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen. Two constant-factor-optimal realizations of adaptive heapsort. In Costas S. Iliopoulos and William F. Smyth, editors, *IWOCA 2011*, volume 7056 of *LNCS*, pages 195–208. Springer, 2011. doi:10.1007/978-3-642-25011-8_16.

5 Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen. The weak-heap data structure: Variants and applications. *J. Discrete Algorithms*, 16:187–205, 2012. doi:10.1016/j.jda.2012.04.010.

6 Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen. Weak heaps engineered. *J. Discrete Algorithms*, 23:83–97, 2013. doi:10.1016/j.jda.2013.07.002.

7 Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen. An in-place priority queue with $o(1)$ time for push and $\lg n + O(1)$ comparisons for pop. In Lev D. Beklemishev and Daniil V. Musatov, editors, *CSR 2015*, volume 9139 of *LNCS*, pages 204–218. Springer, 2015. doi:10.1007/978-3-319-20297-6_14.

8 Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen. Heap construction - 50 years later. *Comput. J.*, 60(5):657–674, 2017. doi:10.1093/comjnl/bxw085.

9 Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen. Optimizing binary heaps. *Theory Comput. Syst.*, 61(2):606–636, 2017. doi:10.1007/s00224-017-9760-2.

10 Stefan Edelkamp, Max Gath, Tristan Cazenave, and Fabien Teytaud. Algorithm and knowledge engineering for the TSPTW problem. In *CISched 2013*, pages 44–51. IEEE, 2013. doi:10.1109/SCIS.2013.6613251.

11 Stefan Edelkamp, Peter Kissmann, and Martha Rohte. Symbolic and explicit search hybrid through perfect hash functions - A case study in connect four. In Steve A. Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml, editors, *ICAPS 2014*. AAAI, 2014. URL: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS14/paper/view/7921.

12 Stefan Edelkamp, Morteza Lahijanian, Daniele Magazzeni, and Erion Plaku. Integrating temporal reasoning and sampling-based motion planning for multigoal problems with dynamics and time windows. *Rob. Autom. Lett.*, 3(4):3473–3480, 2018. doi:10.1109/LRA.2018.2853642.

13 Stefan Edelkamp, Erion Plaku, and Yassin Warsame. Monte-carlo search for prize-collecting robot motion planning with time windows, capacities, pickups, and deliveries. In Christoph Benzmüller and Heiner Stuckenschmidt, editors, *KI 2019*, volume 11793 of *LNCS*, pages 154–167. Springer, 2019. doi:10.1007/978-3-030-30179-8_13.

14 Stefan Edelkamp and Armin Weiß. Blockquicksort: Avoiding branch mispredictions in quicksort. *ACM Journal of Experimental Algorithmics*, 24(1):1.4:1–1.4:22, 2019. doi:10.1145/3274660.

15 Stefan Edelkamp and Armin Weiß. Worst-case efficient sorting with quickmergesort. In Stephen G. Kobourov and Henning Meyerhenke, editors, *ALENEX 2019*, pages 1–14, 2019. doi:10.1137/1.9781611975499.1.

16 Stefan Edelkamp, Armin Weiß, and Sebastian Wild. Quickxsort: A fast sorting scheme in theory and practice. *Algorithmica*, 82(3):509–588, 2020. doi:10.1007/s00453-019-00634-0.

17 Erion Plaku, Sara Rashidian, and Stefan Edelkamp. Multi-group motion planning in virtual environments. *J. of Visualization and Comp. Animation*, 29(6), 2018. doi:10.1002/cav.1688.

18 Álvaro Torralba, Vidal Alcázar, Peter Kissmann, and Stefan Edelkamp. Efficient symbolic search for cost-optimal planning. *Artif. Intell.*, 242:52–79, 2017. doi:10.1016/j.artint.2016.10.001.

Indexing Compressed Text: A Tale of Time and Space

Nicola Prezza
LUISS Guido Carli, Rome, Italy
https://nicolaprezza.github.io/
nprezza@luiss.it

Abstract

Text indexing is a classical algorithmic problem that has been studied for over four decades. The earliest optimal-time solution to the problem, the suffix tree [11], dates back to 1973 and requires up to two orders of magnitude more space than the text to be stored. In the year 2000, two breakthrough works [6, 3] showed that this space overhead is not necessary: both the index and the text can be stored in a space proportional to the text's entropy. These contributions had an enormous impact in bioinformatics: nowadays, the two most widely-used DNA aligners employ compressed indexes [9, 8]. In recent years, it became apparent that entropy had reached its limits: modern datasets (for example, collections of thousands of human genomes) are extremely large but very repetitive and, by its very definition, entropy cannot compress repetitive texts [7]. To overcome this problem, a new generation of indexes based on dictionary compressors (for example, LZ77 and run-length BWT) emerged [7, 5, 1], together with generalizations of the indexing problem to labeled graphs [2, 10, 4]. This talk is a short and friendly survey of the landmarks of this fascinating path that took us from suffix trees to the most modern compressed indexes on labeled graphs.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Sorting and searching; Theory of computation → Pattern matching

Keywords and phrases Compressed Text Indexing

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.3

Category Invited Talk

References

1. F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 180–192, 2012.
2. Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), November 2009. doi:10.1145/1613676.1613680.
3. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, 2000.*, pages 390–398. IEEE, 2000.
4. Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. Algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60th Birthday of Professor Raffaele Giancarlo). doi:10.1016/j.tcs.2017.06.016.
5. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1), January 2020. doi:10.1145/3375890.
6. Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, page 397–406, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/335305.335351.

7 S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

8 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.

9 Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *bioinformatics*, 25(14):1754–1760, 2009.

10 Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 11(2):375–388, March 2014. doi:10.1109/TCBB.2013.2297101.

11 Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

High-Quality Hierarchical Process Mapping

Marcelo Fonseca Faraj
Faculty of Computer Science, University of Vienna, Austria
marcelo.fonseca-faraj@univie.ac.at

Alexander van der Grinten
Humboldt-Universität zu Berlin, Germany
avdgrinten@hu-berlin.de

Henning Meyerhenke
Humboldt-Universität zu Berlin, Germany
meyerhenke@hu-berlin.de

Jesper Larsson Träff
Faculty of Informatics, TU Wien, Vienna, Austria
traff@par.tuwien.ac.at

Christian Schulz[1]
Faculty of Computer Science, University of Vienna, Austria
christian.schulz@univie.ac.at

Abstract

Partitioning graphs into blocks of roughly equal size such that few edges run between blocks is a frequently needed operation when processing graphs on a parallel computer. When a topology of a distributed system is known, an important task is then to map the blocks of the partition onto the processors such that the overall communication cost is reduced. We present novel multilevel algorithms that integrate graph partitioning and process mapping. Important ingredients of our algorithm include fast label propagation, more localized local search, initial partitioning, as well as a compressed data structure to compute processor distances without storing a distance matrix. Moreover, our algorithms are able to exploit a given hierarchical structure of the distributed system under consideration. Experiments indicate that our algorithms speed up the overall mapping process and, due to the integrated multilevel approach, also find much better solutions in practice. For example, one configuration of our algorithm yields similar solution quality as the previous state-of-the-art in terms of mapping quality for large numbers of partitions while being a factor 9.3 faster. Compared to the currently fastest iterated multilevel mapping algorithm Scotch, we obtain 16% better solutions while investing slightly more running time.

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

Keywords and phrases Process Mapping, Graph Partitioning, Algorithm Engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.4

Related Version `http://arxiv.org/abs/2001.07134`

Funding Austrian Science Fund (FWF, project P 31763-N31); DFG grant FINCA (ME-3619/3-2, SPP 1736 Algorithms for Big Data); German Federal Ministry of Education and Research (BMBF, project WAVE, grant 01|H15004B).

[1] Corresponding author.

1 Introduction

The performance of applications that run on high-performance computing systems depends on many factors such as the capability and topology of the underlying communication system, the required communication between processes in the given applications, and the software and algorithms used to realize the communication. For example, communication is typically faster if communicating processes are located on the same physical processor node compared to cases where processes reside on different nodes. This becomes even more pronounced for large supercomputer systems where processing elements are hierarchically organized (e.g., islands, racks, nodes, processors, cores) with corresponding communication links of similar quality, and where differences in the process placement can have a huge impact on the communication performance (latency, bandwidth, congestion). Often the communication pattern between application processes is or can be known. Additionally, a hardware topology description that reflects the capacity of the communication links is typically available. Hence, it is natural to attempt to find a good mapping of the application processes onto the hardware processors such that pairs of processes that frequently communicate large amounts of data are located closely. Another typical application of process mapping can be found in data allocation [33] problems where expensive optimization algorithms are used to assign jobs to data centers in order to minimize expected wait times. Finding such best or just good mappings is the objective of some usually hard optimization problems.

Previous work can be grouped into two categories. One line of research intertwines process mapping with multilevel graph partitioning (see for example [37, 20]). To this end, the objective of the partitioning algorithm – most commonly the number of cut edges – is typically replaced by an objective function that considers the processor distances. Throughout these algorithms, the distances are directly taken into consideration. The second category decouples partitioning and mapping (see for example [29, 3, 12, 18]). First, a graph partitioning algorithm is used to partition a large graph into k blocks, while minimizing some measure of communication, such as edge-cut, and at the same time balancing the load (size of the blocks). Afterwards, a coarser model of computation and communication is created in which the number of nodes matches the number of processing elements (PEs) in the given processor network. This model is then mapped to a processor network of k PEs with given pair-wise distances.

Recently, process mapping algorithms have made two assumptions that are typically valid for modern supercomputers and the applications that run on those: communication patterns are sparse and there is a hierarchical communication topology where links on the same level in the hierarchy exhibit the same communication speed. Using these assumptions, better non-integrated mapping algorithms have been obtained [29]. Here, the model of computation and communication is first partitioned using a standard graph partitioning algorithm, and then a smaller model that has the same number of nodes as the underlying network of processors is mapped. On the other hand, there has been a large body of work on the multilevel (hyper-)graph partitioning problem, which led to enhanced partitioning quality or faster local search [24, 25, 17, 27, 26]. The *multilevel* approach [4] is probably the most prominently used algorithm. Here, the input is recursively *contracted* to obtain a smaller instance which should reflect the same basic structure as the input. After applying an *initial partitioning* algorithm to the smallest instance, contraction is undone and, at each level, *local search* methods are used to improve the partitioning induced by the coarser level.

Our *main contribution* in this paper is the integration of process mapping into a multilevel scheme with high-quality local search techniques and recently developed non-integrated mapping algorithms. Additionally, we introduce faster techniques that avoid to store

distance matrices. The rest of this paper is organized as follows. In Section 2, we introduce basic concepts and describe relevant related work in more detail. We present our main contributions in Section 3. We present a summary of extensive experiments to evaluate algorithm performance in Section 4. The experiments indicate that our new integrated algorithm improves mapping quality over other state-of-the-art integrated and non-integrated mapping algorithms. For example, one configuration of our algorithm yields similar solution quality as the previous state-of-the-art in terms of mapping quality for large values of k while being a factor 9.3 faster. Compared to the currently fastest iterated multilevel mapping algorithm Scotch, we obtain 16% better solutions while investing slightly more running time. Most importantly, hierarchical multisection algorithms that take the system hierarchy into account for model creation improve the results of the overall process mapping significantly.

2 Preliminaries

The communication requirements between the components of a set of processes in (some section of) an application can be represented by a weighted communication graph. The underlying hardware topology can likewise be represented by a weighted graph, particularly a complete graph since any two physical processors can communicate with each other facilitated by the routing system. This complete graph can be represented by a topology cost matrix reflecting the costs of routing along shortest or cheapest paths between physical processors. Furthermore, it does not need to be explicitly expressed if the topology is organized as a regular hierarchy of components with fixed communication cost per message inside each level. We tackle the problem of embedding a communication graph onto a topology graph under optimization criteria that we explain below. Unless otherwise mentioned, a processing element (PE) represents a core of a machine.

2.1 Basic Concepts

Let $G = (V = \{0, \dots, n-1\}, E)$ be an *undirected graph* with edge weights $\omega : E \to \mathbb{R}_{>0}$, vertex weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We generalize c and ω functions to sets, such that $c(V') = \sum_{v \in V'} c(v)$ and $\omega(E') = \sum_{e \in E'} \omega(e)$. Let $\Gamma(v) = \{u : \{v, u\} \in E\}$ denote the neighbors of a vertex v. Let $I(v)$ denote the set of edges incident to v. A graph $S = (V', E')$ is said to be a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. When $E' = E \cap (V' \times V')$, S is an *induced* subgraph.

The *graph partitioning problem* (GPP) consists of assigning each node of G to exactly one of k distinct blocks respecting a balancing constraint in order to minimize the edge-cut. More precisely, GPP partitions V into k blocks $V_1, \dots, V_k$ (i.e., $V_1 \cup \dots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$), which is called a *k-partition* of G. The *balancing constraint* demands that the sum of node weights in each block does not exceed a threshold associated with some allowed *imbalance* ϵ. More specifically, $\forall i \in \{1, \dots, k\} : c(V_i) \leq L_{\max} := \lceil (1+\epsilon)\frac{c(V)}{k} \rceil$. Let a block V_i be called *λ-underloaded* if $c(V_i) + \lambda \leq L_{\max}$ and *overloaded* if $c(V_i) > L_{\max}$. The *edge-cut* of a k-partition consists of the total weight of the edges crossing blocks, i.e., $\sum_{i<j} \omega(E_{ij})$, where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. An abstract view of the partitioned graph is a *quotient graph* $\mathcal{Q}$, in which nodes represent blocks and edges are induced by the connectivity between blocks. More precisely, there is an edge in the quotient graph if there is an edge that runs between the blocks in the original, partitioned graph. We call *neighboring blocks* a pair of blocks connected to each other by an edge in the quotient graph. If a node $v \in V_i$ has a neighbor $w \in V_j, i \neq j$, then it is called a *boundary* node. Let $R(v)$ be the set of all blocks containing at least one element from $\{v\} \cup \Gamma(v)$.

Assume that we have n processes and a topology containing k PEs. Let $\mathcal{C} \in \mathbb{R}^{n \times n}$ denote the communication matrix and let $\mathcal{D} \in \mathbb{R}^{k \times k}$ denote the (implicit) topology matrix or distance matrix. In particular, $\mathcal{C}_{i,j}$ represents the required amount of communication between processes i and j, while $\mathcal{D}_{x,y}$ represents the cost of each communication between PEs x and y. Hence, if processes i and j are respectively assigned to PEs x and y, or vice-versa, the communication cost between i and j will be $\mathcal{C}_{i,j}\mathcal{D}_{x,y}$. Throughout this work, we assume that $\mathcal{C}$ and $\mathcal{D}$ are symmetric – otherwise one can create equivalent problems with symmetric inputs [3].

In this work, we deal with topologies organized as homogeneous hierarchies, even though our algorithms could be extended to heterogeneous hierarchies in a straightforward way. Let $\mathcal{S} = a_1 : a_2 : ... : a_\ell$ be a sequence describing the hierarchy of a supercomputer. The sequence should be interpreted as each processor having a_1 cores, each node a_2 processors, each rack a_3 nodes, and so forth, such that the total number of PEs is $k = \Pi_{i=1}^{\ell} a_i$. Let $D = d_1 : d_2 : \ldots : d_\ell$ be a sequence describing the communication cost inside each hierarchy level, meaning that two cores in the same processor communicate with cost d_1, two cores in the same node but in different processors communicate with cost d_2, two cores in the same rack but in different nodes communicate with cost d_3, and so forth.

Throughout the paper, we assume that the input communication matrix is already given as a graph $G_{\mathcal{C}}$, i.e., no conversion of the matrix into a graph is necessary. More precisely, the graph representation is defined as $G_{\mathcal{C}} := (\{1, \ldots, n\}, E[\mathcal{C}])$ where $E[\mathcal{C}] := \{(u, v) \mid \mathcal{C}_{u,v} \neq 0\}$. In other words, $E[\mathcal{C}]$ is the edge set of the processes that need to communicate with each other. Note that the set contains forward and backward edges, and that the weight of each edge in the graph equals the corresponding entry in the communication matrix $\mathcal{C}$.

Our *main focus* in this work is the *general process mapping problem* (GPMP). It consists of assigning each node of a given communication graph to a specific PE in a communication topology while respecting a balancing constraint (the same as in the graph partitioning problem above) in order to minimize the total communication costs. Within the scope of this work, the number of nodes (processes) n in the communication graph is much larger than the number of PEs k in the topology graph which matches most real-world situations. Let the mapping function that maps a node onto its block be $\Pi : \{1, \ldots, n\} \mapsto \{1, \ldots, k\}$. Hence, the objective function of GPMP is to minimize $J(\mathcal{C}, \mathcal{D}, \Pi) := \sum_{i,j} \mathcal{C}_{i,j}\mathcal{D}_{\Pi(i),\Pi(j)}$. Many authors deal with the specific case in which $n = k$, resulting in the *one-to-one process mapping problem* (OPMP), where each process i is assigned to a unique PE $\Pi(i)$. Within the context of OPMP, searching for the inverse permutation instead, i.e., assigning PE x to node $\Pi^{-1}(x)$, results in the same problem since Π is a bijection.

GPP and OPMP are both NP-hard problems [8, 22]. Since GPP and OPMP are special cases of GPMP, the latter is also NP-hard. Two of the most common methods to solve GPMP are the two-phase approach and the integrated approach. In the *two-phase* approach, GPMP is solved in two consecutive steps: (i) a heuristic for GPP is applied in the communication graph, obtaining a balanced k-partition; (ii) a heuristic for OPMP is used to map the blocks of the k-partition onto the topology of PEs. On the other hand, the *integrated* approach consists of tackling GPMP directly, i.e., not decomposing the input problem into k independent sub-problems first.

2.2 Multilevel Approach

In this section, we characterize the multilevel approach within the scope of GPMP, although the same basic structure is extensible to many other problems, such as GPP. Before describing the multilevel scheme, we need to define the terms contraction and uncontraction. *Contracting*

an edge $e = \{u, v\}$ consists of replacing the nodes u and v by a new node x connected to the former neighbors of u and v and setting $c(x) = c(u) + c(v)$. If replacing edges of the form $\{u, w\}$, $\{v, w\}$ would generate two parallel edges $\{x, w\}$, a single edge with $\omega(\{x, w\}) = \omega(\{u, w\}) + \omega(\{v, w\})$ is inserted. *Uncontracting* undoes contraction.

A *multilevel approach* to solve GPMP consists of three main phases. In the *contraction* (coarsening) phase, successive approximations of an original input graph are created. The contractions quickly reduce the size of the graph and stop as soon as it becomes sufficiently small to be partitioned and mapped by an expensive algorithm. In the construction phase, an initial mapping is produced for the coarsest approximation of the input graph. Due to the way we define contraction, every mapping of the coarsest level implies a corresponding mapping of the input graph with equal objective function and balance. In the *local improvement* (or uncoarsening) phase, we uncontract previously contracted nodes to go back through each level, from the coarsest approximation to the original graph. After each uncoarsening, local improvement algorithms move nodes between blocks in order to improve the objective function or balance.

2.3 Related Work

There has been an immense amount of research on GPP – we refer to [2, 4, 28] for extensive material. The most successful general-purpose methods to solve GPP for huge real-world graphs are based on the multilevel approach. The most commonly used formulation of the multilevel scheme was proposed in [13]. Among the most successful multilevel software packages to solve GPP, we mention Jostle [36], Metis [14], Scotch [19], and KaHIP [23].

Systems like KaHIP [23] and Metis [14] typically compute a k-partition on the coarsest level through a recursive bisection strategy or a direct k-way partitioning scheme. In recursive bisection, the graph is recursively divided into two blocks until the number of blocks is reached, i.e., a bisection algorithm is used to split the graph into two blocks. More precisely, each bisection step itself uses a multilevel algorithm. To obtain a bipartition in the coarsest level, KaHIP uses the *greedy graph growing* algorithm. In KaHIP, if k is not even, the graph gets split into two blocks, V_1 and V_2, such that $c(V_1) \leq \lfloor \frac{k}{2} \rfloor L_{\max}$, $c(V_2) \leq \lceil \frac{k}{2} \rceil L_{\max}$. Block V_1 will be recursively partitioned in $\lfloor \frac{k}{2} \rfloor$ blocks and block V_2 will be recursively partitioned in $\lceil \frac{k}{2} \rceil$ blocks.

In addition to GPP, Jostle and Scotch can also solve GPMP. Jostle integrates local search into a multilevel scheme to partition the model of computation and communication. In this scheme, it solves the problem on the coarsest level and afterwards performs refinements based on the user-supplied network communication model. Scotch performs dual recursive bipartitioning to compute a mapping. More precisely, it starts the recursion considering all given processes and PEs. At each recursion level, it bipartitions the communication graph and also the distance graph with a graph bipartitioning algorithm. The first (resp., second) block of the communication graph is then assigned to the first (resp., second) block of the distance graph.

There is likewise a large literature on OPMP, often in the context of scientific applications using the *Message Passing Interface* (MPI). Hatazaki [11] was among the first authors to propose graph partitioning to solve the MPI process mapping of a virtual unweighted topology onto a hardware topology organized in modules and sub-modules. In [31], a similar approach was used to implement one of the first non-trivial mappings designed for the NEC SX-series of parallel vector computers. In [15] and later [16], the mapping problem was simplified to ignore the whole network topology except that inside each node. These works also investigated multiple placement policies to enhance overall system performance. In [34],

the authors proposed a gradient–based heuristic for OPMP that involves solving assignment problems, and gave experimental evidence for better solution quality and speed compared to other heuristics.

Müller-Merbach [18] proposed a greedy construction method to obtain an initial permutation for OPMP. The method roughly works as follows: Initially compute the total communication volume for each process and also the sum of distances from each core to all the others. Afterwards, the algorithm proceeds in rounds. In each round, the process with the largest communication volume is assigned to the core with the smallest total distance. Glantz et al. [9] noted that the algorithm does not link the choices for the vertices and cores and hence propose a modification of this algorithm called *GreedyAllC* (the best algorithm in [9]). GreedyAllC links the mapping choices by scaling the distance with the amount of communication to be done.

A method to improve an already given solution for OPMP was proposed in [12]. The method repeatedly tries to perform swaps in the assignment in a pair-exchange neighborhood $N(\Pi)$ that contains all permutations that can be reached by swapping two elements in Π. Here, swapping two elements means that $\Pi^{-1}(i)$ will be assigned to processor j and $\Pi^{-1}(j)$ will be assigned to processor i after the swap is done. The algorithm then looks at the neighborhood in a cyclic manner. A swap is performed if it reduces the objective. To reduce the runtime, Brandfass et al. [3] introduced a couple of modifications to speed up the algorithm, such as only considering pairs for swapping that can reduce the objective or partitioning the search space into s consecutive blocks and only performing swaps inside those blocks.

In [29], GPMP was tackled with a two-phase approach. First, the graph is partitioned using KaHIP (which uses recursive bisection). Their best OPMP algorithm, *hierarchy top down*, recursively partitions the communication graph into blocks defined by the given hierarchy. A local search similar to that from [3] with faster data structures was also used to improve the initially computed mapping. The authors experimentally showed that N_C^{10}, which restricts swapping to processes that have a distance smaller than 10 in the communication graph, is an adequate choice to obtain good solutions with a moderate running time.

The OPMP algorithm proposed in [10] requires the hardware topology to be a partial cube, i. e. an isometric subgraph of a hypercube. This requirement allows to label (i) the PEs as well as (ii) the nodes of the application graph G with meaningful bit-strings along convex cuts. These bit-strings facilitate (i) the fast computation of distances between PEs and (ii) an effective hierarchical local search method to improve the mapping induced by the labels. Subsequently, in [35], the graph partitioning step was modified to already use *hierarchical multisection* itself. This yields better communication graphs for the second OPMP mapping step.

3 High-Quality Multilevel General Process Mapping

We engineered all the components of a multilevel algorithm to solve GPMP in an *integrated* way, as illustrated in Figure 1. In this section, we present our algorithmic contributions and discuss each of their components. This includes coarsening-uncoarsening schemes, methods to obtain initial solutions, local refinement methods, and additional tools to explore trade-offs in memory usage and performance.

3.1 Coarsening

We use a matching-based coarsening scheme. The *matching–based* coarsening is the most common choice in multilevel partitioning algorithms due to its simplicity, speed, and generality. It has two consecutive steps: An edge rating function and a matching algorithm. Based on local information, the *edge rating function* scores each edge to estimate the benefit of contracting it. We employ the same edge rating function $\exp^*(e) = \omega(e)/(|\Gamma(u)| * |\Gamma(v)|)$ as used in [24]. Then, the *matching algorithm* obtains a maximal match to maximize the sum of the ratings of the contracted edges. As in [24], we computed matchings with the *Global Paths Algorithm* [24], which is a $\frac{1}{2}$-approximate algorithm.

3.2 Initial Solution Algorithms

We compute the initial mapping using a two-phase approach. To solve GPP, we compare two multilevel recursive bisection algorithms: (i) *standard bisection* setup, in which we perform a recursive bisection to obtain k blocks; (ii) *multisection* setup, in which we perform recursive bisections throughout the hierarchical structure of PEs. To construct a solution for OPMP, we apply two different construction methods: (i) *identity*, which assigns each block to the PE with the same ID to favor locality; (ii) *hierarchy top down*, which partitions the set of blocks throughout the hierarchical structure of PEs. To refine the OPMP solution, we perform an N_C^{10} swap neighborhood local search. Hence, the resulting map Π of nodes to PEs becomes our initial GPMP solution.

Our *standard bisection* setup for initial partition corresponds to the initial partition step in KaHIP. Moreover, it is a canonical choice to produce initial solutions in multilevel schemes tackling GPP. On the other hand, the *multisection* setup draws inspiration from the scheme used in [35]. It is an attempt to specialize the initial partition for the particular case tackled in this paper: a regularly hierarchical distribution of PEs in which the communication cost between two processes (nodes) highly depends on the hierarchy level shared by their corresponding PEs (blocks). Particularly, we apply a recursive partitioning scheme that splits all the nodes in a_ℓ blocks, then splits the nodes in each block in $a_{\ell-1}$ sub-blocks, then splits the nodes in each sub-blocks in $a_{\ell-2}$ sub-sub-blocks, and so forth. Observing that the communication costs decrease as the communicating processes share lower hierarchy levels, the multisection approach implies a hierarchy of sub-problems that directly reflects the problem cost hierarchy.

In both setups of the partitioning step, we recursively assign consecutive IDs to blocks throughout the process in order to maintain locality. Moreover, the PEs belonging to each hierarchy module are labeled with consecutive IDs, which also promotes locality. Then, the

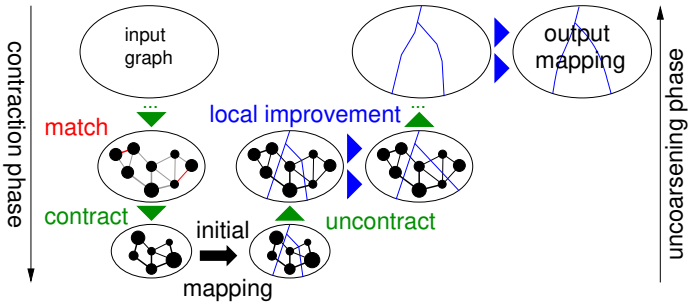


Figure 1 Multilevel scheme used to solve GPMP (Figure from [24]).

identity method is a fast way to construct a solution for OPMP taking advantage of this locality: it assigns each block V_i to the PE with the ID i. Note, the *standard bisection* setup conveniently combines with the identity mapping approach when k is a power of 2 since the recursive bisections will be automatically performed throughout the hierarchical topology. For an analogous reason, the *multisection* setup is a good algorithm to create a coarse model to be mapped by the *identity* mapping approach independently of k. The *hierarchy top down* [29] is a more general approach to construct solutions for OPMP when the PEs are hierarchically organized. Its mechanism is similar to the idea of multisection throughout the hierarchy.

3.3 Uncoarsening

After obtaining an initial solution for GPMP at the coarsest level, we apply a sequence of four local refinement methods to move nodes between blocks (which are already associated to unique PEs). Then, we undo each of the contractions performed previously, from the coarsest graph until the original input graph. After each uncoarsening step, we repeat our four local refinement methods. The refinements run in a specific order based on their characteristics. First, a *quotient graph refinement* exhaustively tries to improve solution quality and eliminate imbalance by moving nodes between each pair of blocks connected by an edge in the quotient graph. Second, a *k-way Fiduccia-Mattheyses (FM) algorithm [7, 32] refinement* greedily goes through the boundary nodes trying to relocate them with a more global perspective in order to improve the mapping. Third, a *label propagation refinement* randomly visits all nodes and moves each one to the most appropriate block while not increasing the objective. Finally, a *multi-try FM refinement* is exhaustively applied in rounds with random starting points throughout the graph in order to escape local optima as many times as possible. Before explaining the local search algorithms, we introduce the notion of *gain* for GPMP.

Gain. All our refinement methods are based on the concept of *gain*. We define $\Psi_b(v)$ as the *partial* contribution of a node v to the objective function $J(\mathcal{C}, \mathcal{D}, \Pi)$ in case v is assigned to the PE b. More precisely, $\Psi_b(v)$ represents the total cost of the communications involving v if $\Pi(v) = b$ and the neighbors of v remain assigned to their current PEs. The *gain* $g_b(v)$ represents the value that will be subtracted from $J(\mathcal{C}, \mathcal{D}, \Pi)$ if a node v is moved from its current PE $\Pi(v)$ to PE b. More precisely, $\Psi_b(v) := \sum_{\{v,u\} \in I(v)} \mathcal{C}_{v,u} \mathcal{D}_{b,\Pi(u)}$ and $g_b(v) := \Psi_{\Pi(v)}(v) - \Psi_b(v)$. Note that $g_{\Pi(v)}(v) \equiv 0$. Observe that a positive (resp., negative) gain indicates improvement (resp., worsening) of the solution. Computing the gains of v to all blocks in $R(v)$ costs $O\big(|R(v)||I(v)|\big) = O\big(|I(v)|^2\big)$. For comparison purposes, the computation of the same corresponding gains in the context of GPP and edge-cut objective function costs $O\big(|I(v)|\big)$.

Quotient Graph Refinement. We implemented an adapted version of the *quotient graph refinement* [24] to incorporate our definition of gains. Within this refinement, we visit each pair of neighboring blocks in the quotient graph $\mathcal{Q}$ underlying the current k-partition. Then we apply an FM algorithm [7] to move nodes between the two currently visited blocks, keeping two respective gain–based priority queues of eligible nodes. Each queue is randomly initialized with the boundary in its corresponding block. After a node is moved (which can only happen once during an execution of the local search), its unmoved neighbors become eligible.

K-Way FM Refinement. Our k-way FM refinement was adapted from the implementation in [24]. Unlike the quotient graph refinement, the k-way FM does not restrict the movement of a node to a certain pair of blocks, but performs global-aware movement choices. Our implementation of k-way FM uses only one gain–based priority queue P, which is initialized with the *complete* partition boundary in a random order. Then, the local search repeatedly looks for the highest-gain node v and moves it to the best $c(v)$-underloaded neighboring block. When a node is moved, we insert in P all its neighbors that were not in P and have not been moved yet. The k-way local search stops if P is empty (i.e., each node was moved once) or when a stopping criterion based on a random-walk model described in [24] applies. To escape from local optima, this refinement allows some movements with negative gain or to blocks that are not $c(v)$-underloaded. Afterwards local search is rolled back to the lowest cut fulfilling the balance criterion that occurred.

Label Propagation Refinement. We propose a local search inspired by *label propagation* [21]. The algorithm works in rounds. In each round, the algorithm visits all nodes in a random order, starting with the labels being the current assignment of nodes to blocks. When a node v is visited, it is moved to the $c(v)$-underloaded neighboring block with highest positive gain. We consider only $c(v)$-underloaded blocks since this ensures that the target block is not overloaded when the node is moved there. Ties are broken randomly and a 0-gain neighboring block can be occasionally chosen with 50% probability if there is no neighboring $c(v)$-underloaded block with positive gain. We perform at most ℓ rounds of the algorithm, where ℓ is a tuning parameter.

Multi-Try FM Refinement. We also adapted our gain concept to a localized variant of the k-way local search algorithm similar to that proposed in [24] under the name of *multi-try* FM. Instead of being initialized with all boundary nodes, as in k-way FM, multi-try FM is repeatedly initialized with a single boundary node. This introduces a higher diversification to the search since it is not restricted to movements in boundary nodes with global largest gain. As a result, this local search can escape local optima more easily than *k-way* FM.

3.4 Additional Techniques

Implicit Distance Matrix. When the topology matrix $\mathcal{D}$ is stored in memory, access time to obtain the distance between a pair of PEs is $O(1)$, but this requires $O(k^2)$ space. From now on, we refer to the algorithm explicitly keeping $\mathcal{D}$ in memory as *matrix–based* approach. We implement three alternative approaches to save memory by exploiting the fact that our topology matrix is a hierarchy and the IDs of PEs in each of the hierarchy modules are sequential. For simplification reasons, we call these approaches: (i) *division–based*; (ii) *stored division–based*; and (iii) *binary notation–based*.

In the *division–based* approach, we perform $O(\ell)$ successive integer divisions and comparisons in the ID of two PEs when we need to find out their distance. Here, ℓ is the number of levels in the system hierarchy. As a preprocessing step to be executed only once, we create a vector $h = \left(k \Big/ \prod_{t=1}^{\ell} a_t,\ k \Big/ \prod_{t=2}^{\ell} a_t,\ \ldots,\ k \Big/ a_\ell\right)$. To find the distance between PEs b and b' with $b \neq b'$, we loop through the hierarchy layers from $i = \ell$ to $i = 1$. In each iteration, we perform the integer division of b and b' by h_i. Whenever the division results differ, then we break the loop and return $\mathcal{D}_{b,b'} = d_i$. This approach does not require any additional memory other than a vector with $O(\ell)$ integers and has time complexity $O(\ell)$.

The *stored division–based* approach works in a similar way as the *division–based* one. The only difference is that we avoid repetitive integer divisions of IDs by elements of h by storing the results of all possible divisions in a preprocessing step executed only once. Although we still need $O(\ell)$ running time to perform comparisons in order to obtain the distance between a pair of PEs, the constant factors involved are much lower. This improvement in running time comes at the cost of additional $O(k\ell)$ memory.

The *binary notation–based* approach is a more compact way of decomposing the IDs of PEs. Instead of storing ℓ numbers for each PE, we keep in memory a single binary number per PE. This binary number r consists of ℓ sections r_i, each containing $s = \lceil \log_2 \left(\max_{1 \leq t \leq \ell}(a_t) \right) \rceil$ bits. To describe the construction of r for a PE b, let a variable t be initialized as $t = b$. Then, we loop through the hierarchy layers, from $i = 1$ to $i = \ell$. In each iteration i, r_i receives the remainder of the division of t by a_i and, then, t is updated to store the integer quotient of t by a_i. Afterwards, it is possible to precisely locate b at the hierarchy by sweeping the sections of r from r_ℓ to r_1. In particular, r_ℓ specifies its data center, $r_{\ell-1}$ specifies its server among those belonging to its data center, and so forth. Obtaining the distance between distinct PEs b and b' is equivalent to finding which section r_i contains the leftmost nonzero bit in the result of the bit-wise operation XOR(b,b'). The running-time complexity of finding the section of the leftmost nonzero bit is $O(\log(\ell))$. Furthermore, current processors often implement a *count leading zeros* (CLZ) operation in hardware which allows the identification of the leftmost nonzero bit in $O(1)$ time, under the assumption that the size $\log r = O(\log k)$ of the binary numbers is smaller than the size of a machine word.

Delta-Gain Updates. Our local searches frequently need to compute *gains* involved in the movement of nodes. A *base approach* to check these gains consists of computing them from scratch whenever they are needed, which can yield many gain recomputations. For this reason, we implement a technique to save running time called *delta-gain updates* [26].

In *delta-gain updates*, we store a vector R of length $|R(v)| = O(|\Gamma(v)|)$ for each node v. In this vector, we keep the gains $g_b(v)$ for all PEs b containing neighbors of v. Additionally, we store an n-sized vector h to keep flags that indicate whether a node has up-to-date gains in memory. Asymptotically speaking, these vectors represent $O(n + m)$ extra memory. Each flag is initialized with an inactive seed and is considered active if its value equals a counter that is increased after each uncoarsening steps. When we need to check a gain of some node v, we look at h_v to verify if the gains of v are up-to-date. If they are not, we compute all gains $g_b(v)$ from scratch, which costs $O\big(|I(v)|^2\big)$, and activate h_v. Otherwise, we just access the required gain from memory in $O(1)$ time.

If a node v moves from its current PE to another one, we have to update all delta gains of v and $u \in \Gamma(v)$ with h_u being active. Assume that h_v and h_u are active and v moves from PE 1 to PE 2 during some local refinement. After this movement, we should change the delta gains of u and v in memory. For v, it suffices to subtract $g_2(v)$ from all other gains of v and then set $g_2(v)$ to 0. For u, it is slightly trickier, but we do not need to recalculate all its gains from scratch since their only source of change is the edge e that connects u and v. Hence, we respectively subtract and add to $g_b(u)$ the corresponding contribution of e before and after the movement of v. We end up doing the update in time $O\big(|I(v)| + |I(v)| * \overline{|R(u)|}\big)$, where $\overline{|R(u)|}$ is the average of $|R(u)|, \forall \{v, u\} \in I(v)$.

4 Experimental Evaluation

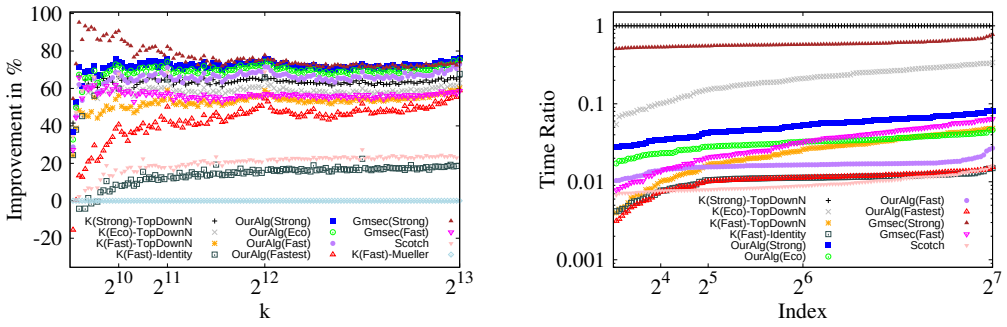
Methodology. We performed our implementations using the KaHIP framework (using C++) and compiled them using gcc 8.3 with full optimization turned on (-O3 flag). All of our experiments were run on a single core of a machine with four sixteen-core Intel Xeon Haswell-EX E7-8867 processors running at 2.5 GHz, 1 TB of main memory, and 32768 KB of L2-Cache. The machine runs Debian GNU/Linux 10 and Linux kernel version 4.19.67-2.

For experiments based on the two-phase approach for tackling GPMP, we solve GPP using KaHIP [24]. We use its configurations *fast*, *eco* and *strong* which are described in [24] – we respectively refer to them as *K(Fast)*, *K(Eco)* and *K(Strong)*. KaHIP also contains the *top down* approach to solve OPMP. We also run Scotch [19] configured to only use recursive bipartitioning methods using the quality setting. We contacted Christopher Walshaw, who informed us that Jostle [36] is not available anymore. Lastly, we compare against global multisection [35] in which the graph partitioning step is already using *hierarchical multisection* itself. We use the configurations *Gmsec(Strong)*, *Gmsec(Eco)* and *Gmsec(Fast)* which differ in the configuration used for partitioning in a global multisection way.

To keep the evaluation simple, we use the following hierarchy configurations for all the experiments: $D = 1 : 10 : 100$, $\mathcal{S} = 4 : 16 : r$, with $r \in \{1, 2, 3, \ldots, 128\}$. Hence, $k = 64 \cdot r$. Depending on the focus of the experiment, we measure running time and/or $J(\mathcal{C}, \mathcal{D}, \Pi)$. We perform ten repetitions of each algorithm using different random seeds for initialization and calculate the arithmetic average of the computed objective functions and running time. When further averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the *final score*. Some of our plots are performance profiles. These plots relate the running times of all algorithms to the slowest algorithm on a per-instance basis. For each algorithm, these ratios are sorted in increasing order. The plots show $\left(\frac{\sigma_A}{\sigma_{\text{slowest}}}\right)$ on the y-axis. A point close to zero indicates that the algorithm was considerably faster than the slowest algorithm.

Instances. Our instances come from various sources. We use the largest six graphs from Chris Walshaw's benchmark archive [30]. Graphs derived from sparse matrices have been taken from the SuiteSparse Matrix Collection [5]. We also use graphs from the 10th DIMACS Implementation Challenge [1] website. Basic properties of the graphs under consideration can be found in Table 1 of the full version of the paper [6].

Algorithm Configuration. We performed a wide range of experiments to tune our algorithm (on the tuning graphs from (Table 1, [6]). Due to space constraints, we refer the reader to the full version of the paper [6] for details. After the tuning step, we have four configurations of our algorithm: All of our algorithms use the binary notation algorithm to compute distances between PEs as this is favorable over storing the distance matrix in terms of speed and memory (see [6]). (i) *fast* uses multisection to compute initial solutions without additional OPMP local search, label propagation with delta-gain updates; (ii) *eco* computes initial solutions as the fast configuration and uses OPMP local search, quotient graph refinement, k-way FM, label propagation; and (iii) *strong* applies multisection to compute initial solutions with additional OPMP local search, and uses all available local search methods. To improve speed even more, we include a configuration called *fastest* which is the same as the fast configuration but does not use local search during uncoarsening.



(a) Improvements in objective function over K(Fast)-Müller-Merbach. Higher is better.

(b) Performance profile for running time. Lower is better.

Figure 2 Comparisons against state-of-the-art approaches for GPMP.

Comparison with State of the Art. In this section, we compare our algorithms against the best alternative algorithms in the literature. We report experiments on all graphs listed in (Table 1, [6]) – excluding the graph used to tune our algorithm. We select the most successful algorithms from [29] and also Scotch for our comparison: (i) *Top down* with N_C^d local search (TopDownN), which represent the state-of-the-art for OPMP when k is not a power of 2; (ii) *identity* mapping, which (when coupled with the KaHIP multilevel partitioning algorithm) represents the state-of-the-art for GPMP via two-phase approach when k is a power of 2; (iii) the algorithm of *Müller-Merbach* [18] (Müller-Merbach), whose results are also used as a reference algorithm to calculate solution improvements in [29]; and (iv) *Scotch* [19]. We run the two-phase approches TopDownN, Identity, and Müller-Merbach coupled with K(Fast), K(Eco) and K(Strong) as a partitioning algorithm. We also compare against global multisection [35], in which the graph partitioning step is already using *hierarchical multisection* itself. Additionally, we use the algorithms Gmsec(Strong), Gmsec(Eco) and Gmsec(Fast). Recall that these algorithms are non-integrated: they use different quality configurations of KaHIP to partition the graph, compute the coarser communication model, and then apply TopDownN to solve OPMP. Scotch is among the algorithms with best running times in our experiments. Hence, we add an algorithm (ScotchTC) which reports the best solution out of multiple runs of Scotch with different random seeds when given the same amount of time to compute a solution as our *strong* configuration has used. Figure 2 gives an overview over our results.

Overall, Scotch has the lowest average running time, directly followed by our algorithm *fastest*, K(Fast)-Identity, and our algorithm *fast* (respectively 9%, 10%, and 73% slower than Scotch on average). Next, the average running time of K(Fast)-TopDownN and Gmsec(Fast) are respectivelly a factor 2.3 and 3.1 higher than Scotch. For our algorithms *eco* and *strong*, this factor is respectively 3.3 and 5.4. By definition ScotchTC is also a factor 5.4 higher than Scotch. Next, Gmsec(Eco) and K(Eco)-TopDownN have much higher running times (9.3 and 20.4 times slower than Scotch, respectively). Finally, Gmsec(Strong) and K(Strong)-TopDownN are the slowest ones (62 and 107 times slower than Scotch, respectively).

We now highlight the comparison of various configurations/algorithms. Gmsec(Strong) is the algorithm with best overall mapping quality. It is 2.5% on average better compared to our *strong* configuration, however our *strong* configuration is more than an order of magnitude faster on average – a factor 11.5 faster. Better quality of Gmsec(Strong) stems from the fact

that global multisection itself already takes the system hierarchy into account and hence yields good models to be mapped. Moreover, the graph partitioning approach itself which is used to compute a communication graph also uses more (time-consuming) sophisticated local search algorithms. This includes methods such as flow-based methods which particularly work well for small values of k as well as global search methods such as V-cycles. In particular for $k > 2^{11}$, our *strong* has the same average quality as Gmsec(Strong) but is 9.3 times faster. Our algorithm computes similar solutions in much less time since the multilevel algorithm directly optimizes the correct objective. For $k > 2^{11}$, our *eco* is 2.4% better and 2.8 times faster than Gmsec(Eco), and our *fast* is 9% better and 2.6 times faster than Gmsec(Fast). Overall, our *strong* configuration improves solution quality over K(Strong)-TopDownN by 5.1% while being a factor 20 on average faster. Our *eco* configuration has roughly 3.6% better quality than K(Strong)-TopDown but is a factor 32 faster on average. Our *fast* configuration still yields 1.3% better solutions than K(Strong)-TopDown on average, and is a factor 62 faster. Here, improvements stem from the fact that the new algorithms are integrated and not two-phase as well as the fact that these algorithms do not perform multisections. Lastly, our *fastest* algorithm is on average 9% slower than Scotch but also improves solution quality over Scotch by 16%. Our *strong* algorithm is 40% better than Scotch and consumes a factor 5.4 more running time.

5 Conclusion

We tackled the general process mapping problem, which is to assign a larger set of processes to a smaller set of processing elements such that total communication cost between cores is minimzed. Our algorithms integrate graph partitioning and process mapping. Important ingredients of our algorithm include fast label propagation, more localized local search, initial partitioning, as well as a compressed data structure to compute processor distances without storing a distance matrix.

Experimental results indicate that our algorithms are the new state-of-the-art for general process mapping. In particular, our algorithms generate much better or similar overall solutions in comparison to any of the competitors while being an order of magnitude faster than the previous best algorithm in terms of quality. Our improvements are mostly due to the integrated multilevel approach combined with high-quality local search algorithms and initial mapping algorithms that split the initial network along the specified system hierarchy.

Important future work includes parallelization as well as the integration of global search schemes and different types of coarsening to improve solution quality further. Moreover, we plan to investigate the impact on the real performance of applications such as sparse matrix vector multiplications. Lastly, we plan to release the proposed algorithms in the VieM (`http://viem.taa.univie.ac.at/`) and KaHIP (`http://algo2.iti.kit.edu/kahip/`) frameworks.

References

1 D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014.

2 C. Bichot and P. Siarry, editors. *Graph Partitioning.* Wiley, 2011.

3 B. Brandfass, T. Alrutz, and T. Gerhold. Rank reordering for MPI communication optimization. *Computers & Fluids*, 80:372–380, 2013.

4 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.

5 T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011. doi:10.1145/2049662.2049663.

6 M. Fonseca Faraj, A. van der Grinten, H. Meyerhenke, J. L. Träff, and C. Schulz. High-quality hierarchical process mapping. *CoRR*, abs/2001.07134, 2020. arXiv:2001.07134.

7 C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proc. of the 19th Conference on Design Automation*, pages 175–181, 1982.

8 M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-Complete Problems. In *Proc. of the 6th ACM Symposium on Theory of Computing*, (STOC), pages 47–63. ACM, 1974.

9 R. Glantz, H. Meyerhenke, and A. Noe. Algorithms for mapping parallel processes onto grid and torus architectures. In *23rd Euromicro Intl. Conference on Parallel, Distributed, and Network-Based Processing*, pages 236–243, 2015.

10 R. Glantz, M. Predari, and H. Meyerhenke. Topology-induced enhancement of mappings. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018*, pages 9:1–9:10. ACM, 2018. doi:10.1145/3225058.3225117.

11 T. Hatazaki. Rank reordering strategy for MPI topology creation functions. In *5th European PVM/MPI User's Group Meeting*, volume 1497 of *LNCS*, pages 188–195, 1998.

12 C. H. Heider. A computationally simplified pair-exchange algorithm for the quadratic assignment problem. Technical report, DTIC Document, 1972.

13 B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proc. of the ACM/IEEE Conference on Supercomputing'95*. ACM, 1995.

14 G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

15 G. Mercier and J. Clet-Ortega. Towards an efficient process placement policy for MPI applications in multicore environments. In *16th European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, volume 5759 of *LNCS*, pages 104–115. Springer, 2009.

16 G. Mercier and E. Jeannot. Improving MPI applications performance on multicore clusters with rank reordering. In *18th European MPI Users' Group Meeting*, volume 6960 of *LNCS*, pages 39–49. Springer, 2011.

17 H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning Complex Networks via Size-constrained Clustering. In *13th Int. Symp. on Exp. Algorithms*, volume 8504 of *LNCS*. Springer, 2014.

18 H. Müller-Merbach. *Optimale reihenfolgen*, volume 15 of *Ökonometrie und Unternehmensforschung.* Springer-Verlag, 1970.

19 F. Pellegrini. Scotch Home Page. http://www.labri.fr/pelegrin/scotch.

20 François Pellegrini and Jean Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer, 1996. doi:10.1007/3-540-61142-8_588.

21 U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

22 S. Sahni and T. F. Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, 1976. doi:10.1145/321958.321975.

23 P. Sanders and C. Schulz. KaHIP – Karlsruhe High Qualtity Partitioning Homepage. http://algo2.iti.kit.edu/documents/kahip/index.html.

24 P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proc. of the 19th European Symp. on Algorithms*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.

25 P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *12th Intl. Sym. on Experimental Algorithms (SEA)*, LNCS. Springer, 2013.

26 S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. k-way hypergraph partitioning via n-level recursive bisection. In *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX*, pages 53–67, 2016. doi:10.1137/1.9781611974317.5.

27 C. Schulz. *High Quality Graph Partitititioning*. PhD thesis, Karlsruhe Institute of Technology, 2013.

28 C. Schulz and D. Strash. Graph partitioning: Formulations and applications to big data. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019. doi:10.1007/978-3-319-63962-8_312-2.

29 C. Schulz and J. L. Träff. Better process mapping and sparse quadratic assignment. In *16th International Symposium on Experimental Algorithms*, volume 75 of *LIPIcs*, pages 4:1–4:15, 2017. doi:10.4230/LIPIcs.SEA.2017.4.

30 A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *Global Optimization*, 29(2):225–241, 2004.

31 J. L. Träff. Implementing the MPI process topology mechanism. In *ACM/IEEE Supercomputing*, pages 40:1–40:14, 2002.

32 Jesper Larsson Träff. Direct graph k-partitioning with a Kernighan-Lin like heuristic. *Operations Research Letters*, 34(6):621–629, 2006.

33 R. Vamosi, M. Lassnig, and E. Schikuta. Data allocation based on evolutionary data popularity clustering. In Yong Shi, Haohuan Fu, Yingjie Tian, Valeria V. Krzhizhanovskaya, Michael Harold Lees, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Computational Science - ICCS 2018 - 18th International Conference, Wuxi, China, June 11-13, 2018, Proceedings, Part I*, volume 10860 of *Lecture Notes in Computer Science*, pages 153–166. Springer, 2018. doi:10.1007/978-3-319-93698-7_12.

34 J. T. Vogelstein, J. M. Conroy, V. Lyzinski, L. J. Podrazik, S. G. Kratzer, E. T. Harley, D. E. Fishkind, R. J. Vogelstein, and C. E. Priebe. Fast approximate quadratic programming for graph matching. PLOS One, April 2015.

35 K. von Kirchbach, C. Schulz, and J. L. Träff. Better process mapping and sparse quadratic assignment. *CoRR*, 2019. URL: http://arxiv.org/abs/1702.04164v2.

36 C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

37 C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Comp. Syst.*, 17(5):601–623, 2001. doi:10.1016/S0167-739X(00)00107-2.

Probing a Set of Trajectories to Maximize Captured Information

Sándor P. Fekete
Department of Computer Science,
TU Braunschweig, Germany
s.fekete@tu-bs.de

Alexander Hill
Department of Computer Science,
TU Braunschweig, Germany
a.hill@tu-bs.de

Dominik Krupke
Department of Computer Science,
TU Braunschweig, Germany
d.krupke@tu-bs.de

Tyler Mayer
Decision Management Systems,
Charles River Analytics Inc., Boston, MA, USA
tmayer@cra.com

Joseph S. B. Mitchell
Department of Applied Mathematics and
Statistics, Stony Brook University, NY, USA
joseph.mitchell@stonybrook.edu

Ojas Parekh
Sandia National Laboratories,
Albuquerque, NM, USA
odparek@sandia.gov

Cynthia A. Phillips
Sandia National Laboratories,
Albuquerque, NM, USA
caphill@sandia.gov

Abstract

We study a trajectory analysis problem we call the Trajectory Capture Problem (TCP), in which, for a given input set $\mathcal{T}$ of trajectories in the plane, and an integer $k \geq 2$, we seek to compute a set of k points ("portals") to maximize the total weight of all subtrajectories of $\mathcal{T}$ between pairs of portals. This problem naturally arises in trajectory analysis and summarization.

We show that the TCP is NP-hard (even in very special cases) and give some first approximation results. Our main focus is on attacking the TCP with practical algorithm-engineering approaches, including integer linear programming (to solve instances to provable optimality) and local search methods. We study the integrality gap arising from such approaches. We analyze our methods on different classes of data, including benchmark instances that we generate. Our goal is to understand the best performing heuristics, based on both solution time and solution quality. We demonstrate that we are able to compute provably optimal solutions for real-world instances.

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

Keywords and phrases Algorithm engineering, optimization, complexity, approximation, trajectories

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.5

Related Version A full version of the paper is available at `https://arxiv.org/abs/2004.03486` [6].

Supplementary Material The code and data of the experimental evaluation is available at `https://github.com/ahillbs/trajectory_capturing`.

Acknowledgements Work by Tyler Mayer was mostly carried out while at Stony Brook University. Joe Mitchell and Tyler Mayer were partially supported by the National Science Foundation (CCF-1526406) and a grant from the US-Israel Binational Science Foundation (BSF project 2016116). Joe Mitchell was also partially supported by the DARPA Lagrange program. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

1 Introduction

In recent years, the progress in technical capabilities has resulted in massive amounts of trajectory data for cars, trucks, trains, aircraft, ships, people, and animals being collected at increasing rates. This presents major challenges for storing and evaluating this ever-growing data, as well as for extracting useful information; this motivates the search for data structures and algorithms that capture some of the most important and useful aspects of such trajectories. At the same time, the availability of large volumes of data makes it possible to consider useful aspects that were previously unavailable due to the lack of data or algorithmic evaluation methods, such as collecting useful information along the traveled trajectories.

One such means of analyzing a set $\mathcal{T}$ of trajectories is to determine "popular pairs" (p_1, p_2) of locations (or location/time pairs) for which there is a significant "value" of the trajectories $\mathcal{T}$ going between those points. This value can arise from aggregated data between checkpoints, such as the total passenger-distance or the accumulated total pollution along the way; it also comes up through the use of tomographic methods (i.e., determining physical phenomena by measuring aggregated effects along the path between two sensors), which are highly important in the context of many other application areas, such as in astrophysics [10]. A limiting factor is usually the need to pick a set of locations of finite cardinality, i.e, placing a limited number of toll booths, cameras for average speed measurement, various other types of sensors, or abstract collections of focus points for sampling trajectories. For an example, consider the scenario shown in Figure 1, which corresponds to more than 500,000 data points that arise from the trajectories of over 250 taxi cabs in San Francisco. Our goal is to identify a small subset of locations that allow us to capture as much of the movement information, in terms of weighted distance between checkpoints, as possible.

Figure 1 A set of taxi trajectories after preprocessing in San Francisco Bay Area. (Satellite images courtesy of Planet Labs Inc.)

In this paper, we study the Trajectory Capturing Problem (TCP), a generalization of "popular pair" computation: Given a set $\mathcal{T}$ of trajectories and an integer $k \geq 2$, determine a set of k *portals* (points) to maximize the sum of the weights of the inter-portal subtrajectories in $\mathcal{T}$; such subtrajectories are said to be "captured" by the set of portals. After establishing that the TCP is NP-hard, and giving some first approximation bounds, we focus on algorithm engineering methods for attacking the problem practically.

1.1 Our Results

We provide results both for algorithmic theory and algorithm engineering aspects of the Trajectory Capturing Problem.

- We prove NP-hardness for the TCP, even for instances with trajectories consisting of individual axis-parallel segments.
- We establish two approximation algorithms. One has approximation factor K if the input trajectory set decomposes into K subsets where within each subset no two segments cross, though they can overlap, (K noncrossing subsets). The other has an approximation factor Δ, the maximum number of input trajectories hit by any single point.
- We develop an Integer Linear Program (IP) to solve TCP instances to provable optimality.
- We show that, in general, the IP formulation has unbounded integrality gap[1]. For inputs decomposing into 2 noncrossing subsets (e.g., arising from axis-parallel segments), we show that the integrality gap is at most $\frac{k}{\lceil k/2 \rceil}$ (Theorem 7).
- We develop methods for generating challenging benchmark instances for experimentation. For geometric instances, based on segment arrangements, this requires care to address geometric robustness and accuracy.
- We compute provably optimal solutions for general instances up to thousands of candidate capture points.
- We give provably optimal solutions for even larger instances, with up to 7000 possible capture points, for instances based on axis-parallel segments, where we find the integrality gap to be quite small.
- Using the IP solutions as a reference, we perform a thorough computational study using heuristic algorithms (a greedy algorithm, iterated local search, simulated annealing, and an evolutionary algorithm), with various settings, to understand how heuristics perform on various instances, in terms of time and solution quality.
- We demonstrate our methods on real-world instances, including a provably optimal solution for taxi-cab data on 250 trajectories, with more than 500,000 individual geographic data points.

Given the broad range of potential applications, scenarios and assumptions, we do not claim (or even aim) to provide a final set of methods for the general problem. Instead, we focus on demonstrating that a number of modeling and optimization approaches can provide a promising range of insights and tools for future work from various directions.

1.2 Related Work

Related to our problem is the well-studied Geometric Hitting Set Problem (GHS), in which one seeks a smallest cardinality set of points to hit a given set of lines, segments, or trajectories in the plane; as a single point suffices to capture all of a trajectory it lies on, achieving large objective values for the GHS is easier than for the TCP. The GHS is known to be NP-hard, hard to approximate (below a threshold), and some natural geometric cases have constant-factor approximation algorithms; see, e.g., [5], and the references therein.

There is a vast literature on problems of analyzing, clustering, mining, and summarizing a set of trajectories. For an extensive survey of trajectory data mining methods, see Zheng [22] and Zheng and Zhou [23]. Notions of "flocks" and "meetings" have been formalized and

[1] The integrality gap is $\max_{\mathcal{I}} \frac{\mathrm{LP}(\mathcal{I})}{\mathrm{IP}(\mathcal{I})}$, where $\mathcal{I}$ is a TCP instance, $\mathrm{IP}(\mathcal{I})$ is the optimal IP solution value and $\mathrm{LP}(\mathcal{I})$ is the optimal solution value to the linear programming (LP) relaxation.

studied algorithmically [2, 7, 11, 21]. Gudmundsson, van Kreveld, and Speckmann [8] define *leadership*, *convergence*, and *encounter* and provide exact and approximate algorithms to compute each. Andersson, Gudmundsson, Laube, and Wolle [1] show that several *Leader-Problem* (LP) variants (*LP-Report-All, LP-max-Length, LP-Max-Size*) are all polynomially solvable and provide exact algorithms. Buchin et al. [3] present a framework to fully categorize trajectory grouping structures (grouping, merging, splitting, and ending of groups). Other approaches to trajectory summarization naturally include cluster analysis, of which there is a large body of related work. Li, Han, and Yang [14] consider rectilinear trajectories and show how to cluster with bounding rectangles of a given size. Several approaches (e.g., [9, 12, 13, 17]) consider density-based methods for clustering sub-trajectories. Lee, Han, Li, and Gonzalez [12] take it one step further by considering a two-level clustering hierarchy that first accounts for regional density and then considers lower-level movement patterns. Li, Ding, Han, and Kays [15] consider a problem (related to [8]) in which they seek to identify all *swarms* or groups of entities moving within an arbitrary shaped cluster for a certain, possibly disconnected, duration of time. Also, Uddin, Ravishankar, and Tsotras [20] consider finding what they call *regions of interest* in a trajectory database.

In motivating tomographic applications, the number of checkpoints is an important constraint in the use of discrete tomography, e.g., in astrophysics (Korth et al. [10]).

2 Preliminaries

We are given a set of trajectories $\mathcal{T}$, each specified by a sequence of points, e.g., in the Euclidean plane. We seek a set $P = \{p_1, \ldots, p_k\}$ of k *portals*, i.e., selected points that lie on some of the trajectories. While our practical study focuses on instances in which the trajectories $\mathcal{T}$ are purely spatial, e.g., given as polygonal chains or line segments in the plane, our methods apply equally well to more general portals and to trajectories that include a temporal component and live in space-time. More generally, we are given a graph $\mathcal{G}$, with length-weighted edges, and a set of paths within $\mathcal{G}$. We wish to determine a subset of k of the nodes of $\mathcal{G}$ that maximizes the sum of the (weighted) lengths of the subpaths (of the input paths) that link consecutive portals along the input paths.

We seek to compute a P that maximizes the total *captured weight* of subtrajectories between pairs of portals. For a trajectory $\tau \in \mathcal{T}$, if there are two or more portals of P that lie along τ, say $\{p_{i_1}, \ldots, p_{i_q}\}$ (for $q \geq 2$), then the subtrajectory, $\tau_{p_{i_1}, p_{i_q}}$, between p_{i_1} and p_{i_q} is *captured* by P, and we get credit for its weight $f(\tau_{p_{i_1}, p_{i_q}})$. (For many of our instances, $f(\tau_{p_{i_1}, p_{i_q}})$ corresponds to the Euclidean distances, denoted by $|\tau_{p_{i_1}, p_{i_q}}|$, but our methods generalize to other types of weights.) Let $f_P(\tau)$ denote the captured weight of trajectory τ by the portal set P. The Trajectory Capture Problem (TCP) is then to compute, for given $\mathcal{T}$ and k, a set of k portals $P = \{p_1, \ldots, p_k\}$ to maximize $\sum_{\tau \in \mathcal{T}} f_P(\tau)$.

3 Analytical Results

The TCP is NP-hard and hard to approximate for general graphs even when all trajectories have weight 1. Given a graph, let each edge be a trajectory. Then an optimal solution to TCP gives the densest k-node subgraph. Manurangsi [16] showed that, assuming the exponential time hypothesis, there cannot be an $n^{\frac{1}{\log \log n^c}}$-approximation algorithm for the Densest k-Subgraph problem for any constant $c > 0$.

In the following, we give more specific results for a range of geometric TCP versions.

3.1 One-Dimensional TCP

In the one-dimensional setting, the underlying graph $\mathcal{G}$ is a path, and the input trajectories $\mathcal{T} = \{(a_1, b_1), \ldots, (a_n, b_n)\}$ are a set of subpaths of $\mathcal{G}$, specified by pairs of integers, a_i, b_i. A solution to the TCP then consists of k points, $P = \{p_1, \ldots, p_k\}$, w.l.o.g. indexed in sorted order, $p_1 < p_2 < \cdots < p_k$.

▶ **Theorem 1.** *The one-dimensional TCP can be solved exactly in polynomial time.*

Proof. For $i = 1, 2, \ldots, k-1$, let $V_i(x)$ be the maximum possible length of $\mathcal{T}$ captured by points $(p_i, \ldots, p_k)$, with $p_i = x \in \{a_1, \ldots, a_n, b_1, \ldots, b_n\}$; let $V_k(x) = 0$, for any x. Then, the value functions V_i satisfy the following dynamic programming recursion, for $i = 1, 2, \ldots, k-1$, and each $x \in \{a_1, \ldots, a_n, b_1, \ldots, b_n\}$:

$$V_i(x) = \max_{x' \in \{a_1, \ldots, a_n, b_1, \ldots, b_n\}, x' > x} \{V_{i+1}(x') + \sum_{j:(x,x') \subseteq (a_j, b_j)} (x' - x)\}.$$

The summation counts the length $(x' - x)$ once for each input interval that contains the interval (x, x'). We can compute the $O(nk)$ values $V_i(x)$ in time $O(n^2 k)$ by incrementally updating the summation as we consider values of x' in increasing order. ◀

3.2 Two-Dimensional TCP

3.2.1 Complexity

▶ **Theorem 2.** *The TCP is NP-hard, even for an input $\mathcal{T}$ of n line segments in the plane.*

The proof of Theorem 2 (see full version [6]) uses a construction involving segments of *many* orientations whose pairwise intersections may only be single points. The following shows that the TCP is already NP-hard for segments of *two* orientations, provided that two intersecting segments may be collinear, and three different segments can intersect in a single point.

▶ **Theorem 3.** *The TCP is NP-hard, even for an input $\mathcal{T}$ of n line segments of at most two orientations in the plane, with any two segments intersecting in at most a single point (there are no overlapping pairs of segments) but possible collinearity.*

3.2.2 Approximation Algorithms

Consider first the case in which the set $\mathcal{T}$ of input trajectories is the (disjoint) union of K subsets of trajectories, $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \cdots \cup \mathcal{T}_K$, with each subset $\mathcal{T}_i$ having the following *path property*: For any connected component of the intersection graph of $\mathcal{T}_i$, the trajectories in that component are all subpaths of some path in the union of $\mathcal{T}_i$. This condition holds, for example, if $\mathcal{T}$ is a set of line segments of K distinct orientations.

▶ **Theorem 4.** *The TCP for an input set $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \cdots \cup \mathcal{T}_K$, with each subset $\mathcal{T}_i$ having the path property, has a polynomial-time K-approximation algorithm.*

Proof. Within each class $\mathcal{T}_i$, the path property implies that the trajectories behave like intervals on a line, so our one-dimensional dynamic programming solution applies. Selecting the best solution that uses all k points for one of the $\mathcal{T}_i$ gives a polynomial-time algorithm with approximation ratio K. ◀

▶ **Theorem 5.** *The TCP for an input set $\mathcal{T}$ of arbitrarily overlapping/crossing trajectory paths in the plane having bounded depth Δ (i.e., no point of $\Re^2$ lies in more than Δ input trajectories) has a polynomial-time Δ-approximation algorithm, for any even number, k, of portals. (If k is odd, the approximation factor is at most $\Delta(1+\frac{1}{k-1})$.)*

Proof. Consider an optimal set, H^*, of k hit points, capturing total trajectory length L^*. For each hit point $h \in H^*$, which lies on $\delta \leq \Delta$ trajectories of $\mathcal{T}$, we replace h with δ copies ("clones") of the point h, with one clone associated with each of the δ trajectories that h hits. In total there are at most $k\Delta$ clones. Consider any trajectory $\tau \in \mathcal{T}$, and consider the clones (copies of hit points) that lie along τ. If there are at least 2 clones on τ, then the portion of τ that lies between the extreme clones on τ is captured; the length of this portion is at most the length of τ. The $k\Delta$ clones capture portions of at most $\lfloor k\Delta/2 \rfloor$ trajectories, resulting in total captured length $L^* \leq \ell_1 + \ell_2 + \cdots + \ell_{\lfloor k\Delta/2 \rfloor}$, where ℓ_i denotes the length of the ith longest trajectory of $\mathcal{T}$ ($\ell_1 \geq \ell_2 \geq \ell_3 \geq \cdots$).

Now consider the simple greedy algorithm that places hit points at the 2 endpoints of the $\lfloor k/2 \rfloor$ longest trajectories of $\mathcal{T}$, using at most k total hit points. This algorithm captures length $L = \ell_1 + \ell_2 + \cdots + \ell_{\lfloor k/2 \rfloor}$. The approximation ratio is at most

$$\frac{L^*}{L} \leq \frac{\lfloor k\Delta/2 \rfloor}{\lfloor k/2 \rfloor}.$$

Thus, $\frac{L^*}{L} \leq \Delta$ for even k. For odd k, the denominator is exactly $(k-1)/2$, while the numerator is either $k\Delta/2$ (if Δ is even) or $(k\Delta-1)/2$ (if Δ is odd); thus, $\frac{L^*}{L} \leq \frac{k\Delta/2}{(k-1)/2} = \Delta(1+\frac{1}{k-1})$. ◀

4 Algorithm Engineering

As the TCP can be considered an optimization problem on a weighted graph, we can use approaches such as Integer Linear Programming and local search heuristics. Given the geometric origins of the TCP, we consider geometric aspects; in addition, dealing with geometric data involved a number of other aspects of algorithm engineering, such as accuracy and correctness when handling locations, coordinates, and intersections.

4.1 Integer Linear Programming

4.1.1 An IP Formulation

As a problem of combinatorial optimization, the TCP can be modeled as an Integer Linear Program (IP), for which solutions can be computed with the help of powerful IP solvers. The following IP models the TCP.

$$\max \textstyle\sum_{\tau\in\mathcal{T}, e\in E(\tau)} f(e) x_{\tau,e}$$

$$\textstyle\sum_{v\in V} y_v \leq k \qquad \text{(Constraint 1)}$$

$$\forall \tau = (v_0, \ldots, v_l) \in \mathcal{T}:$$

$$\forall i \in 0, \ldots, l-1: \quad \begin{cases} x_{\tau,v_iv_{i+1}} \leq y_i & \text{if } i = 0, \\ x_{\tau,v_iv_{i+1}} \leq y_i + x_{\tau,v_{i-1}v_i} & \text{else} \end{cases} \qquad \text{(Constraint 2)}$$

$$\forall i \in 1, \ldots, l: \quad \begin{cases} x_{\tau,v_{i-1}v_i} \leq y_i & \text{if } i = l, \\ x_{\tau,v_{i-1}v_i} \leq y_i + x_{\tau,v_iv_{i+1}} & \text{else} \end{cases} \qquad \text{(Constraint 3)}$$

$$\forall v \in V, \tau \in e \in \tau: \quad x_{\tau,e}, y_v \in \{0,1\}$$

We have two types of Boolean variables: y_v, for $v \in V$, which indicates if node v is one of the k selected portals, and $x_{\tau,e}$, for edge $e \in E$ on trajectory τ, which indicates if the portion e of trajectory τ is captured by selected portals. For an edge e, there are distinct variables, $x_{\tau,e}, x_{\tau',e}$, for trajectories $\tau \neq \tau'$, because e can be captured in τ but not in τ'.

Our objective function maximizes the weighted sum of captured trajectory edges, where $E(\tau)$ denotes the edges of τ in $\mathcal{G}$, and $f(e)$ is the weight (i.e. length) of edge e. (Optionally, we could have trajectory-dependent weights on edges.) Constraint 1 limits the number ($\leq k$) of selected portals. Constraints 2 and 3 enforce that, in order for an edge to be captured as part of trajectory τ, there must be a selected portal in each direction; either there is a selected portal at the next node, or the following trajectory edge is also captured. In the latter case, because τ has no cycle (it is a simple path), there must be a selected portal on τ at some point in that direction if any portion of τ is to be captured.

For an example, see Section A.5 in the full version [6].

4.1.2 Fractional Solutions

Relaxing the integrality constraints of the IP may result in fractional solutions. We show (in the full version [6]) that the gap (ratio) between the best fractional and the integral (optimal) solution objective functions can be arbitrarily large, for any fixed k.

▶ **Theorem 6.** *The integrality gap for the TCP IP can be arbitrarily large for any k.*

For instances arising from non-overlapping (i.e., no parallel segments may share more than one point) axis-parallel segments, we can bound the integrality gap, because the particularly bad "clusters" of the general case cannot occur.

▶ **Theorem 7.** *For trajectories $\mathcal{T}$ arising from non-overlapping axis-parallel line segments, the integrality gap is at most $\frac{k}{\lfloor k/2 \rfloor}$, for $k \geq 2$.*

Proof. We can easily get an integral solution by simply capturing the $\lfloor \frac{k}{2} \rfloor$ longest trajectories (segments) by selecting their at most k endpoints as portals.

We create a new LP instance, called LP_2, by including two copies v_1, v_2 of each portal variable v, so that one copy lies only on horizontal segments, while the other lies only on vertical segments. We constrain $y_{v_1} = y_{v_2} = y_v$ and allow a budget of $2k$ portals for LP_2. Any feasible values for the y_i in the original LP solution are still feasible in LP_2 (setting both copies), so the optimal solution of LP_2 is an upper bound for the original LP. Because segments do not overlap, every portal now lies only on a single segment. Thus the optimal solution for LP_2 covers the $\frac{2k}{2} = k$ longest segments. This shows the integrality gap is at most $\frac{k}{\lfloor k/2 \rfloor}$. ◀

The bound of Theorem 7 is tight for $k = 2$: Consider four segments that are edges of a unit square; then, $k = 2$ portals can capture at most length 1, while a fractional value of $1/2$ at each of the four corners yields objective value $4/2 = 2$ for the LP. For $k \geq 4$, it becomes increasingly difficult to build instances with a high integrality gap.

4.2 Heuristics

Integer Linear Programming solvers can provide provably optimal or near-optimal solutions for relatively large instances. However, eventually runtime and memory requirements become a limiting factor for large enough instances, so it becomes important to develop effective heuristics. We considered a spectrum of heuristics: *Greedy*, which constructs solutions from

scratch by locally optimal choices; *Iterated Local Search*, which iteratively improves a current solution by finding a better one in its local neighborhood; *Simulated Annealing*, which uses a "temperature" function that governs the probability of temporarily accepting a worse solution during a local search; and *Genetic Algorithms*, which maintain a selection of solutions that are locally modified and combined to achieve gradually better solutions.

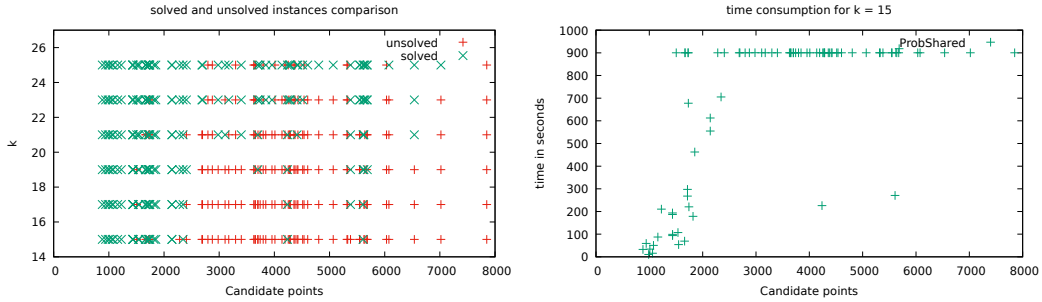
Greedy begins by selecting the two portals at the ends of the longest trajectory, and then incrementally, greedily selects portals that in each step increase the total captured length as much as possible. *Greedy* can be fooled and give poor solutions; it can, though, serve to give a reasonable starting solution for our other metaheuristics.

Iterated Local Search (ILS) is a basic metaheuristic that, given an initial solution, iteratively replaces the current solution with the best solution found by applying a single local modification, until no further improvement can be achieved. The set of solutions that can be obtained by a single local modification from a specific solution is called its *neighborhood*. For a local modification operator based on changing a single portal, the neighborhood consists of all solutions that differ in exactly one portal. The smaller the neighborhood, the faster the best solution within it can be found; however, a smaller neighborhood also reduces the search space and correspondingly can reduce the quality of the obtained solutions. We considered *global neighborhoods*, based on moving a random portal to an alternative random candidate node, and *local neighborhoods*, based on moving a single portal to positions adjacent to other (unmoved) portals. ILS is initialized with any reasonable solution; after some experimentation with alternatives (e.g., random selection), we settled on using *Greedy* as the starting solution for ILS.

Simulated Annealing is similar to ILS, but instead of searching for the best solution in the neighborhood, it selects a random solution in the neighborhood and moves to it if (1) it is an improvement, or (2) if it is not an improvement, but it passes a random test. The probability of moving to a worse solution is determined by a "temperature function" and decreases with time. Initially, it can easily escape local optima; when the search satisfies a termination criterion, it returns the best solution it found.

We considered three different termination criteria: the total number of iterations, the number of iterations without an improvement, and the total runtime. For temperature regulation, we used a geometric reduction by a constant multiplicative amount; for diversification we "reheated" the temperature to the start temperature when we did not change the solution for a certain number of iterations. For translating the temperature function to a probability function, we used the Bolzmann function: $\text{bolzmann}(s', s'', T) := \exp^{(-\frac{s'-s''}{T})}$, where s' and s'' are the captured weight of two neighboring solutions. In addition, we used parallelization for pursuing multiple searches from different starting points.

Evolutionary algorithms (EAs) are motivated by the way adaption to environmental conditions happens in nature. They maintain a "population" of current solutions. At each step, the EA produces new solutions through mutations (i.e., local changes) and recombination (combining pieces of solutions in the current population to create new ones). Then, the EA keeps the best solutions (previous or new) to maintain a stable population size. We create the initial population by a version of Greedy that starts with a random segment instead of the longest one. For mutations, we used ILS or SA. The probability of selection for recombination is $\frac{f(s)-f(s_{\min})}{\sum_{s' \in S} f(s')}$. We used uniform random crossover.



(a) Solved and unsolved instances. **(b)** Time used for instances with $k = 15$.

Figure 2 Test results for solved and unsolved instances using the IP. The number of seed points varies from 35 up to 55 points. All tests were performed with a time limit of 900 seconds.

4.3 Generating Benchmark Instances

Our IP and heuristic methods apply to general sets of trajectories $\mathcal{T}$, given by spatiotemporal or combinatorial data. We focus on geometric instances, most of which are based on line-segment trajectories. Instances based on random segments tend to be very easy to solve because most vertices have degree 2. So we generated instances based on a set of seed points and selected segments linking them, resulting in arrangement graphs with multi-trajectory intersections and more complicated covering graphs. Alternatively, we tested adding new intersection points to the set of seed points when incrementally constructing the arrangement. For all methods, we used exact intersection point computations from the Computational Geometry Algorithms Library (CGAL) to overcome problems of floating point precision for large instances. We generated seed points randomly, using a variety of spatial distributions, including uniform distributions, point sets from the TSP benchmark library [19], and point sets with density distributions based on light maps, corresponding to population densities (see [4]).

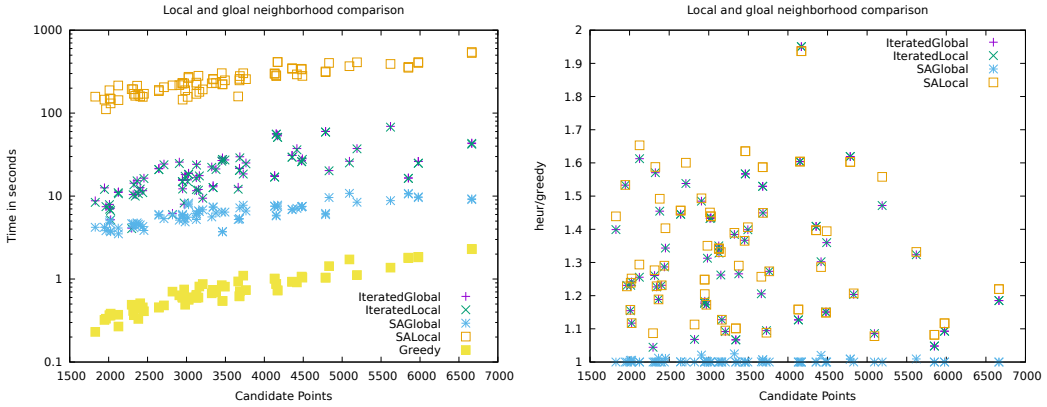
4.4 Experimental Evaluation

All experiments were performed on a single Intel(R) Core(TM) i7-4770 (4×3.4 GHz) with 32 GB and CPLEX (V12.7.1 with default settings), with a time limit of 900 s. The code and data is available at `https://github.com/ahillbs/trajectory_capturing`.

4.4.1 Integer and Linear Programming

We first consider the sizes of instances that our IP can solve to optimality within a 900-second time limit, and we consider which factors contribute to the difficulty of the instance.

We varied the number of seed points between 35 and 55 and varied the (uniform) probability of a segment connecting two seeds between 10% and 20%. In Figure 2a, we see that instances with up to about 2500 candidate points (i.e., nodes at intersection points in the arrangement, where portals can be placed) can be solved for $15 \leq k \leq 23$ to provable optimality within the time limit. Instances with more than 2500 candidate points are most often not solved for $k < 23$. For instances with $k \geq 23$, the problem seems to be easier to solve. In Figure 2b we see that for $k = 15$ and between 1500 and 2000 candidate points, instances start to become very difficult to solve. However, for $k \geq 23$, instances are still solvable for more than 2500 candidate points.



(a) Comparison of required time.

(b) Comparison of achieved objective values and greedy solutions.

Figure 3 Comparison of solutions for local and global neighborhood with Iterated Local Search and Simulated Annealing for $k = 25$.

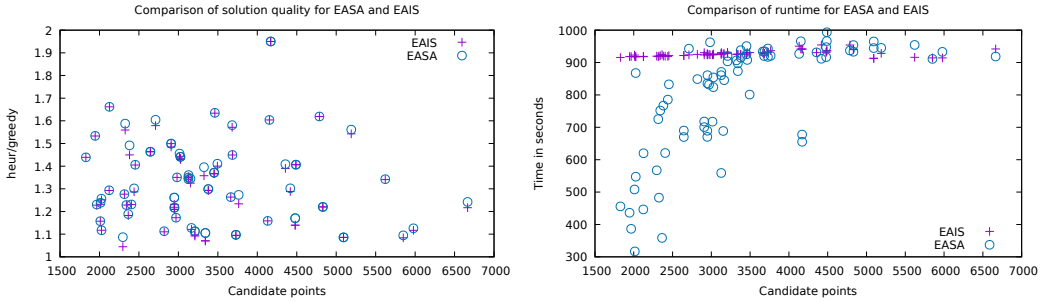
4.4.2 Heuristic Methods

Neighborhoods for Local Methods. For modifying a given solution, we considered global neighborhoods, in which a portal is moved to an arbitrary other position, and local neighborhoods, in which a portal is only moved to positions that connect to another portal. Using global neighborhoods, all solutions are theoretically quickly reachable but they are significantly larger than local neighborhoods and, thus, a meta-heuristic may not work in a focused enough way. Details of this comparison can be found in Section A.4 in the full version [6]; in particular Figure 3 shows our experimental evaluation. Iterated Local Search yields the same solution quality for both neighborhoods; with global neighborhood, only the runtime increases. Simulated Annealing with global neighborhoods barely improves the initial greedy solution, while it gives the best solutions with local neighborhoods.

As a result, we used local neighborhoods for all meta-heuristics.

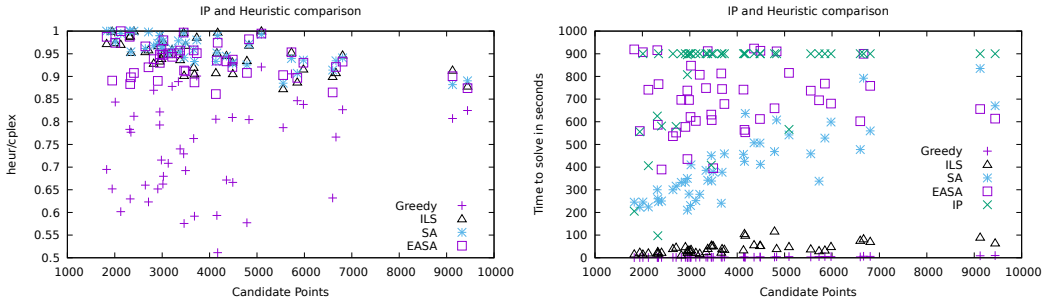
Mutation Strategy for Evolutionary Algorithms. For evolutionary algorithms, choosing the right kinds of mutations is of crucial importance, as these allow reaching solutions that are not achievable only via recombination. Practical usefulness requires focused mutations that have a high probability of being useful, instead of purely random changes. That is why we considered Iterated Local Search and Simulated Annealing as mutation operations. As Simulated Annealing has a longer runtime, we used a faster terminating version (with potentially worse solutions) when using it for mutation. In the following we refer to the version with Simulated Annealing as EASA and with Iterated Local Search as EAIS. We have a start with 100 solutions and keep an ongoing population of 50 solutions. The evolutionary algorithm stops after 15 minutes (but can take slightly longer to finish the last round) or if it has not found an improvement for multiple rounds.

Figure 4 shows the experimental comparison of both mutation variants. One can see that EASA performs slightly better and is significantly faster for smaller instances. This implies that EASA often quickly finds a good solution but is usually not able to improve it further and terminates early. EAIS, on the other hand, is able to improve its quality until the time limit but still remains slightly worse. For the further experiments, we settled on EASA.



(a) Solution quality for Evolutionary Algorithms. **(b)** Runtimes for Evolutionary Algorithms.

Figure 4 Comparison of solution quality and runtime by Evolutionary Algorithm with Iterated Local Search and with Simulated Annealing.



(a) Solution quality for heuristics and IP solutions. **(b)** Runtimes of heuristics and IP solutions.

Figure 5 Comparisons of solution quality and runtime for all tested algorithms.

4.4.3 Comparison of Heuristics with IP as Baseline

We compared the heuristics in terms of solution quality and runtime against the IP solver, which produces not only solutions, but also guaranteed bounds.

Figure 5 shows the obtained results. The Evolutionary Algorithm produces, on average, the worst solutions of all metaheuristics, while still requiring more time than the others. However, it is the only metaheuristic tested that reliably computes good solutions for instances consisting of several point clusters, for which solutions consist of several connected components. These instances did not occur with our generation method but only in separate, manually created instances which are not part of this experiment.

The greedy approach never falls below $\frac{1}{2}$OPT for these instances, while being the fastest.

Iterated Local Search (ILS) appears to produce quite good solutions, which are not worse than 10% below the optimal solution, in a very short time frame. While it only finds a local optimum, it seems that the objective function quality of these are quite close to the optimal values. As a consequence, Iterated Local Search can produce good solutions for instances with up to 5000 candidate points and $k = 25$.

The best heuristic algorithm, in terms of solution quality and runtime, appears to be Simulated Annealing. The combination of fast diversification at high temperature and random swaps for improving solutions at low temperature seems to work quite well; in addition, the mechanism of reheating to restart diversification, in combination with multi-threading to cover a larger search space, are characteristics that are not present in the Evolutionary Algorithm. For $k = 25$, Simulated Annealing produces excellent solutions for instances with up to 5000 candidate points; this instance size can be easily increased for smaller k.

In summary, we can produce excellent heuristic solutions for instances with up to 5000 candidate points and $k = 25$. If fast solutions are desired, Iterated Local Search is the method of choice. The best tradeofff between runtime and solution quality is offered by Simulated Annealing. Finally, for cluster-based instances, we recommend Evolutionary Algorithms.

4.4.4 Linear Programming and Integrality Gap

As described in Section 3, if the TCP input trajectories come from K subsets of noncrossing trajectories, we have a K-approximation algorithm based on dynamic programming. In particular, if the input trajectories consist of axis-parallel line segments, $K = 2$, so there is a 2-approximation. This may coincide with better practical solvability of these kinds of instances. We have verified this for some instances for which all segments are axis-parallel and (for collinear segments) non-overlapping. (See Figure 11 in the full paper [6] for such an instance with 1100 segments and roughly 8200-8500 candidate points. We have also solved instances with 2000 segments and 19,000 candidate points.) Furthermore, for instances with up to 20,000 points, the integrality gap was never larger than 20% for $k = 5$, 7% for $k = 10$, 5% for $k = 15$ and less than 2.5% for larger k. See Figures 13–17 in the full version [6].

4.4.5 Application to Taxi Trajectory Data

We have applied our TCP model to solve real-world data sets to optimality. In Figure 6 we show the results of computing $k = 5$ optimal portals for a set of trajectories based on taxi cab routes in the San Francisco Bay Area. The data is based on 375 vehicles, sampled every 5 minutes, 288 times per day, for one week [18]; see the trajectories in Figure 1.

Our experiments included runs on 30 instances, with k ranging from 5 to 11, on sets of 10 to 120 trajectories of varying lengths (comprised of 1300 to 3700 edges, and 600 to 1800 vertices). The trajectories are snapped to a regular grid graph. Solution times of the IP were up to 200 seconds of computation, with most instances taking less than 10 seconds.

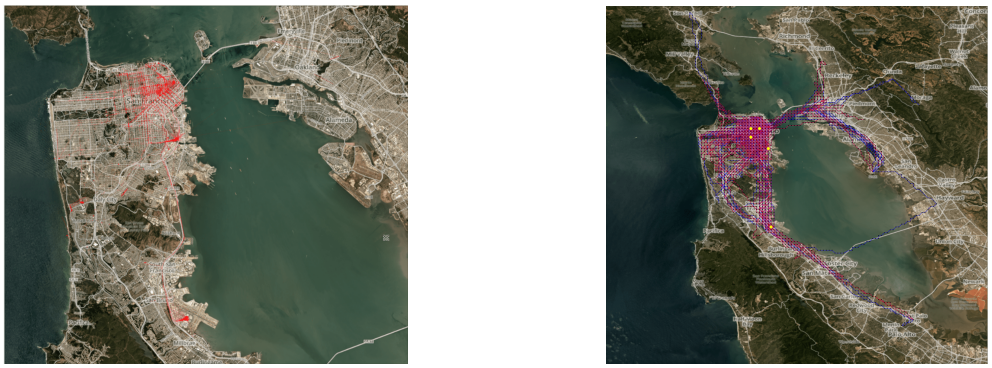


Figure 6 (Left) Candidate points before processing. (Right) Solution to real-world TCP instance: An optimal set of $k = 5$ portals are highlighted. Red trajectory portions (some of which may loop back) are captured, blue ones are not captured. Satellite images are courtesy of Planet Labs Inc.

5 Conclusion

We have shown that the TCP is NP-hard, even for axis-aligned line segment trajectories in the plane, and given approximation algorithms for two cases. Can we improve the approximation factor of K for a set of trajectories that is the union of K subsets, each of which is noncrossing? Can we improve the approximation factor of Δ for a set of trajectories of depth at most Δ?

A focus of our work is the exploration, via algorithm engineering, of practical methods for solving the TCP. Our methods are based on integer programming and on simple heuristic search methods. It will be interesting to develop more specified methods for other, specific classes of instances, such as further geometric instances arising from other types of real-world geographic data.

References

1. Mattias Andersson, Joachim Gudmundsson, Patrick Laube, and Thomas Wolle. Reporting leaders and followers among trajectories of moving point objects. *GeoInformatica*, 12(4):497–528, 2008.
2. Marc Benkert, Joachim Gudmundsson, Florian Hübner, and Thomas Wolle. Reporting flock patterns. *Computational Geometry*, 41(3):111–125, 2008.
3. Kevin Buchin, Maike Buchin, Marc van Kreveld, Bettina Speckmann, and Frank Staals. Trajectory grouping structure. In *Algorithms and data structures*, pages 219–230. Springer, 2013.
4. Sándor P. Fekete, Andreas Haas, Michael Hemmer, Michael Hoffmann, Irina Kostitsyna, Dominik Krupke, Florian Maurer, Joseph S. B. Mitchell, Arne Schmidt, Christiane Schmidt, and Julian Troegel. Computing nonsimple polygons of minimum perimeter. *Journal of Computational Geometry*, 8:340–365, 2017.
5. Sándor P Fekete, Kan Huang, Joseph SB Mitchell, Ojas Parekh, and Cynthia A Phillips. Geometric hitting set for segments of few orientations. *Theory of Computing Systems*, 62(2):268–303, 2018.
6. Sándor P. Fekete, Alexander Hill, Dominik Krupke, Tyler Mayer, Joseph S. B. Mitchell, Ojas Parekh, and Cynthia A. Phillips. Probing a set of trajectories to maximize captured information, 2020.
7. Joachim Gudmundsson and Marc van Kreveld. Computing longest duration flocks in trajectory data. In *Proc. of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems*, pages 35–42. ACM, 2006.
8. Joachim Gudmundsson, Marc van Kreveld, and Bettina Speckmann. Efficient detection of patterns in 2D trajectories of moving points. *Geoinformatica*, 11(2):195–215, 2007.
9. Marios Hadjieleftheriou, George Kollios, Dimitrios Gunopulos, and Vassilis J Tsotras. On-line discovery of dense areas in spatio-temporal databases. In *Advances in Spatial and Temporal Databases*, pages 306–324. Springer, 2003.
10. Haje Korth, Michelle F. Thomsen, Karl-Heinz Glassmeier, and W. Scott Phillips. Particle tomography of the inner magnetosphere. *Journal of Geophysical Research: Space Physics*, 107(A9):SMP–5, 2002.
11. Patrick Laube, Matt Duckham, and Thomas Wolle. Decentralized movement pattern detection amongst mobile geosensor nodes. In *Geographic Information Science*, pages 199–216. Springer, 2008.
12. Jae-Gil Lee, Jiawei Han, Xiaolei Li, and Hector Gonzalez. TraClass: trajectory classification using hierarchical region-based and trajectory-based clustering. *Proceedings of the VLDB Endowment*, 1(1):1081–1094, 2008.
13. Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. Trajectory clustering: a partition-and-group framework. In *Proc. of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 593–604. ACM, 2007.
14. Yifan Li, Jiawei Han, and Jiong Yang. Clustering moving objects. In *Proc. Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 617–622. ACM, 2004.
15. Zhenhui Li, Bolin Ding, Jiawei Han, and Roland Kays. Swarm: Mining relaxed temporal moving object clusters. *Proceedings of the VLDB Endowment*, 3(1-2):723–734, 2010.

16 Pasin Manurangsi. Almost-polynomial ratio eth-hardness of approximating densest k-subgraph. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 954–961, 2017.

17 Mirco Nanni and Dino Pedreschi. Time-focused clustering of trajectories of moving objects. *Journal of Intelligent Information Systems*, 27(3):267–289, 2006.

18 Michal Piorkowski, Natasa Sarafijanovic-Djukic, and Matthias Grossglauser. CRAWDAD dataset epfl/mobility (v. 2009-02-24), doi:10.15783/c7j010, February 2009.

19 Gerhard Reinelt. TSPLIB–a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

20 Md Reaz Uddin, Chinya Ravishankar, and Vassilis J Tsotras. Finding regions of interest from trajectory data. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 1, pages 39–48. IEEE, 2011.

21 Marcos R Vieira, Petko Bakalov, and Vassilis J Tsotras. On-line discovery of flock patterns in spatio-temporal data. In *Proc. of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 286–295. ACM, 2009.

22 Yu Zheng. Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3):29, 2015.

23 Yu Zheng and Xiaofang Zhou. *Computing with spatial trajectories*. Springer Science & Business Media, 2011.

Storing Set Families More Compactly with Top ZDDs

Kotaro Matsuda
Graduate School of Information Science and Technology, The University of Tokyo, Japan
kotaro_matsuda@mist.i.u-tokyo.ac.jp

Shuhei Denzumi
Graduate School of Information Science and Technology, The University of Tokyo, Japan
denzumi@mist.i.u-tokyo.ac.jp

Kunihiko Sadakane
Graduate School of Information Science and Technology, The University of Tokyo, Japan
sada@mist.i.u-tokyo.ac.jp

Abstract

Zero-suppressed Binary Decision Diagrams (ZDDs) are data structures for representing set families in a compressed form. With ZDDs, many valuable operations on set families can be done in time polynomial in ZDD size. In some cases, however, the size of ZDDs for representing large set families becomes too huge to store them in the main memory.

This paper proposes top ZDD, a novel representation of ZDDs which uses less space than existing ones. The top ZDD is an extension of top tree, which compresses trees, to compress directed acyclic graphs by sharing identical subgraphs. We prove that navigational operations on ZDDs can be done in time poly-logarithmic in ZDD size, and show that there exist set families for which the size of the top ZDD is exponentially smaller than that of the ZDD. We also show experimentally that our top ZDDs have smaller size than ZDDs for real data.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases top tree, Zero-suppressed Decision Diagram, space-efficient data structure

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.6

Related Version A full version of the paper is available at [9], `https://arxiv.org/abs/2004.04586`.

1 Introduction

Zero-suppressed Binary Decision Diagrams (ZDDs) [10] are data structures which are derived from Binary Decision Diagrams (BDDs) [3] and which represent a family of sets (combinatorial sets) in a compressed form by Directed Acyclic Graphs (DAGs). ZDDs are data structures specialized for processing set families and compress sparse set families well. ZDDs support binary operations between two set families in time polynomial to the ZDD size. Due to these advantages, ZDDs are used for combinatorial optimization and enumeration.

Though ZDDs can store set families compactly, their size may grow for some set families, and we need further compression. DenseZDDs [5] are data structures for storing ZDDs in a compressed form and supporting operations on the compressed representation. DenseZDDs represent a ZDD by a spanning tree of the DAG representing it, and an array of pointers between nodes on the spanning tree. Therefore its size is always linear to the original size and to compress more, we need another representation.

Our basic idea for compression is as follows. In a ZDD, the identical sub-structures are shared and replaced by pointers. However identical sub-structures cannot be shared if they appear at different heights in ZDD. As a result, even if the DAG of a ZDD contains repetitive structures in height direction, they cannot be shared.

For not DAGs but trees, there exists a data structure called top DAG compression [2], which can capture repetitive structures in height direction. We extend it for DAGs and apply to compress ZDDs which support the operations on compressed ZDDs.

1.1 Our contribution

We propose top ZDDs, which partition the edges of a ZDD into a spanning tree and other edges called complement edges, and store each of them in a compressed form. For the spanning tree, we use the top DAG compression, which represents a tree by a DAG with fewer number of nodes. For the complement edges, we store them in some nodes of the top DAG by sharing identical edges. We show that basic operations on ZDDs can be supported in $O(\log^2 n)$ time where n is the number of nodes of the ZDD. For further compression we use succinct data structures for trees [12] and for bitvectors [14, 8].

We show experimental results on the size and query time of our top ZDDs and existing data structures. The results show that the top ZDDs use less space for most of input data.

2 Preliminaries

Here we explain notations and basic data structures.

Let $C = \{1, \ldots, c\}$ be the universal set. Any set in this paper is a subset of C. The empty set is denoted by $\emptyset$. For a set $S = \{a_1, \ldots, a_s\} \subseteq C$ $(s \geq 1)$, its size is denoted by $|S| = s$. The size of the empty set is $|\emptyset| = 0$. A subset of the power set of C is called a set family. If a set family $\mathcal{F}$ satisfies either $S \in \mathcal{F} \Rightarrow \forall k \in S, S\backslash\{k\} \in \mathcal{F}$ or $S \in \mathcal{F} \Rightarrow \forall k \in C, S \cup \{k\} \in \mathcal{F}$, $\mathcal{F}$ is said to be monotone. If the former is satisfied, $\mathcal{F}$ is monotone decreasing and the latter monotone increasing.

2.1 Zero-suppressed Binary Decision Diagrams

Zero-suppressed Binary Decision Diagrams (ZDDs) [10] are data structures for manipulating finite set families. A ZDD is a directed acyclic graph (DAG) $G = (V, E)$ with a root node satisfying the following properties. A ZDD has two types of nodes; branching nodes and terminal nodes. There are two types of terminal nodes $\bot$ and $\top$. These terminal nodes have no outgoing edges. Each branching node v has an integer label $\ell(v) \in \{1, \ldots, c\}$, and also has two outgoing edges 0-edge and 1-edge. The node pointed to by the 0-edge (1-edge) of v is denoted by $v_0 = \mathit{zero}(v)$ $(v_1 = \mathit{one}(v))$. If for any branching node v it holds $\ell(v) < \ell(v_0)$ and $\ell(v) < \ell(v_1)$, the ZDD is said to be ordered. In this paper, we consider only ordered ZDDs. For convenience, we assume $\ell(v) = c + 1$ for terminal nodes v. We divide the nodes of the ZDD into layers $L_1, \ldots, L_{c+1}$ $(i = 1, \ldots, c + 1)$ according to the labels of the nodes. Note that if $i \geq j$ there are no edges from layer L_i to layer L_j. The number of nodes in ZDD G is denoted by $|G|$ and called the size of the ZDD. On the other hand, the data size of a ZDD stands for the number of bits used in the data structure representing the ZDD.

The set family represented by a ZDD is defined as follows.

▶ **Definition 1** (The set family represented by a ZDD). *Let v be a node of a ZDD and $v_0 = \mathit{zero}(v)$, $v_1 = \mathit{one}(v)$. Then the set family $\mathcal{F}_v$ represented by v is defined as follows.*

1. *If v is a terminal node: if $v = \top$, $\mathcal{F}_v = \{\emptyset\}$, if $v = \bot$, $\mathcal{F}_v = \emptyset$.*
2. *If v is a branching node: $\mathcal{F}_v = \{S \cup \{\ell(v)\} \mid S \in \mathcal{F}_{v_1}\} \cup \mathcal{F}_{v_0}$.*

For the root node r of ZDD G, $\mathcal{F}_r$ corresponds to the set family represented by the ZDD G. This set family is also denoted by $\mathcal{F}_G$.

All the paths from the root to the terminal $\top$ on ZDD G have one-to-one correspondence to all the sets $S = \{a_1, \ldots, a_s\}$ in the set family represented by G. Consider a traversal of nodes from the root towards terminals so that for each branching node v on the path, if $\ell(v) \notin S$ we go to $v_0 = \mathit{zero}(v)$ from v, and if $\ell(v) \in S$ we go to $v_1 = \mathit{one}(v)$ from v. By repeating this process, if $S \in \mathcal{F}_G$ we arrive at $\top$, and if $S \notin \mathcal{F}_G$ we arrive at $\bot$ or the branching node corresponding to $a_i \in S$ does not exist.

2.2 Succinct data structures

Succinct data structures are data structures whose size match the information theoretic lower bound. A data structure is succinct if any element of a finite set U with cardinality L is encoded in $\log_2(L) + o(\log_2(L))$ bits. In this paper we use the following data structures.

2.2.1 Bitvectors

Bitvectors are the most basic succinct data structures. A length-n sequence $B \in \{0,1\}^n$ of 0's and 1's is called a bitvector. On this bitvector we consider the following operations:

- $\mathit{access}(B, i)$ $(1 \leq i \leq n)$: returns $B[i] \in \{0,1\}$, the i-th entry of B.
- $\mathit{rank}_c(B, i)$ $(1 \leq i \leq n, c = 0, 1)$: returns the number of c in the first i bits of B.
- $\mathit{select}_c(B, j)$ $(1 \leq j \leq n, c = 0, 1)$: returns the position of the j-th occurrence of c in B.

The following result is known.

▶ **Theorem 2** ([14]). *For a bitvector of length n, using a $n + \mathrm{O}(n \log\log n / \log n)$-bit data structure constructed in $\mathrm{O}(n)$ time, $\mathit{access}(B, i), \mathit{rank}_c(B, i), \mathit{select}_c(B, j)$ are computed in constant time on the word-RAM with word length $\Omega(\log n)$.*

Consider a bitvector of length n with m ones. For a sparse bitvector, namely, the one with $m = \mathrm{o}(n/\log n)$, we can obtain a more space-efficient data structure.

▶ **Theorem 3** ([8]). *For a bitvector B of length $n = 2^w$ with m ones, $\mathit{select}_1(B, i)$ is computed in constant time on the word-RAM with word length $\Omega(\log n)$ using a $m(2 + w - \lfloor \log_2 n \rfloor) + \mathrm{O}(m \log\log m / \log m)$-bit data structure.*

Note that on this data structure, $\mathit{rank}_0, \mathit{rank}_1, \mathit{select}_0$ takes $\mathrm{O}(\log m)$ time.

2.2.2 Trees

Consider a rooted ordered tree with n nodes. An information-theoretic lower bound of such trees is $2n - \Theta(\log n)$ bits. We want to support the following operations: (1) $\mathit{parent}(x)$: returns the parent of node x, (2) $\mathit{firstchild}(x), \mathit{lastchild}(x)$: returns the first/last child of node x, (3) $\mathit{nextsibling}(x), \mathit{prevsibling}(x)$: returns the next/previous sibling of node x (4) $\mathit{isleaf}(x)$: returns if node x is a leaf or not, (5) $\mathit{preorder\_rank}(x)$: returns the preorder of node x, (6) $\mathit{preorder\_select}(i)$: returns the node with preorder i, (7) $\mathit{leaf\_rank}(x)$: returns the number of leaves whose preorders are smaller than that of node x, (8) $\mathit{leaf\_select}(i)$: returns the i-th leaf in preorder, (9) $\mathit{depth}(x)$: returns the depth of node x, that is, the distance from the root to x, (10) $\mathit{subtreesize}(x)$: returns the number of nodes in the subtree rooted at node x, (11) $\mathit{lca}(x, y)$: returns the lowest common ancestor (LCA) between nodes x and y.

▶ **Theorem 4** ([12]). *On the word-RAM with word length $\Omega(\log n)$, the above operations are done in constant time using a $2n + \mathrm{o}(n)$-bits data structure.*

We call this the BP representation in this paper.

2.3 DenseZDD

A DenseZDD [5] is a static representation of a ZDD with attricuted edges [11] by using some succinct data structures. In comparison to the ordinary ZDD, a DenseZDD provides a much faster membership operation and less memory usage for most of cases. When we construct a DenseZDD from a given ZDD, dummy nodes are inserted so that $\ell(v_0) = \ell(v) + 1$ holds for each internal node v for fast traversal. The spanning tree consisting of all reversed 0-edges is represented by straight forward BP. The DenseZDD is a combination of this BP and other succinct data structures that represent remaining information of the given ZDD.

3 Top Tree and Top DAG

We explain top DAG compression [2] to compress labeled rooted trees.

Top DAG compression is a compression scheme for labeled rooted trees by converting the input tree into top tree [1] and then compress it by DAG compression [4, 6, 7]. DAG compression is a scheme to represent a labeled rooted tree by a smaller DAG obtained by merging identical subtrees of the tree. Top DAG compression can compress repeated sub-structures (not only subtrees). For example, a path of length n with identical labels can be represented by a top DAG with $O(\log n)$ nodes. Also, for a tree with n nodes, accessing a node label, computing the subtree size, and tree navigational operations such as first child and parent are done in $O(\log n)$ time. Here we explain the top tree and its greedy construction algorithm. We also explain operations on top DAGs.

The top tree [1] for a labeled rooted tree T is a binary tree $\mathcal{T}$ representing the merging process of clusters of T defined as follows. We assume that all edges in the tree are directed from the root towards leaves, and an edge (u, v) denotes the edge from node u to node v. Clusters are subsets of T with the following properties.

- A cluster is a subset F of the nodes of the original tree T such that nodes in F are connected in T.
- F forms a tree and we regard the node in F closest to the root of T as the root of the tree. We call the root of F as the *top boundary node*.
- F contains at most one node having directed edges to outside of F. If there is such a node, it is called the *bottom boundary node*.

A boundary node is either a top boundary node or a bottom boundary node.

By merging two adjacent clusters, we obtain a new cluster, where merge means to take the union of node sets of two clusters and make it as the node set of the new cluster. There are five types of merges, as shown in Figure 1.

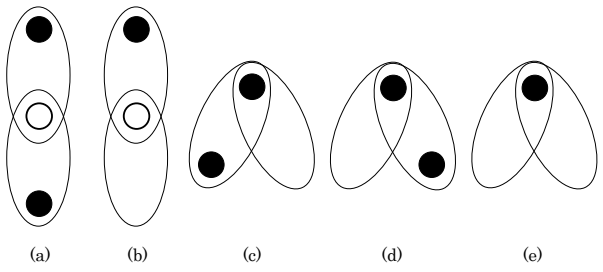


Figure 1 Types of merging of clusters. Ellipses are clusters before merge, black circles are boundary nodes of new clusters, and white circles are not boundary nodes in new clusters.

These five merges are divided into two.

1. (a)(b) Vertical merge: two clusters can be merged vertically if the top boundary node of one cluster coincides with the bottom boundary node of the other cluster, and there are no edges from the common boundary node to nodes outside the two clusters.
2. (c)(d)(e) Horizontal merge: two clusters can be merged horizontally if the top boundary nodes of the two clusters are the same and at least one cluster does not have the bottom boundary node.

The top tree of the tree T is a binary tree $\mathcal{T}$ satisfying the following conditions.

- Each leaf of the top tree corresponds to a cluster with the endpoints of an edge of T.
- Each internal vertex of the top tree corresponds to the cluster made by merging the clusters of its two children. This merge is one of the five types in Figure 1.
- The cluster of the root of the top tree is T itself.

We call the DAG obtained by DAG compression of the top tree $\mathcal{T}$ as top DAG $\mathcal{T}D$, and the operation to compute the top DAG $\mathcal{T}D$ from tree T is called top DAG compression [2].

We define labels of vertices in the top tree to apply DAG compression as follows. For a leaf of the top tree, we define its label as the pair of labels of both endpoints of the corresponding edge in T. For an internal vertex of the top tree, its label must have the information about the cluster merge. It is enough to consider three types of merges, not five as in Figure 1. For vertical merges, it is not necessary to store the information that the merged cluster has the bottom boundary node or not. For horizontal merges, it is enough to store if the left cluster has a bottom boundary node or not. From this observation, we define labels of internal vertices as follows.

- For vertices corresponding to vertical merge: we set their labels as V.
- For vertices corresponding to horizontal merge: we set their labels as $\mathrm{H_L}/\mathrm{H_R}$ if the left/right child cluster has the bottom boundary node, respectively. If both children do not have bottom boundary nodes, the label can be arbitrary.

Top trees created by a greedy algorithm satisfy the following.

▶ **Theorem 5** ([2]). *Let n be the number of nodes of a tree T. Then the height of* top tree $\mathcal{T}$ *created by a greedy algorithm is* $\mathrm{O}(\log n)$.

Consider to support operations on a tree T which is represented by top DAG. From now on, a node x in T stands for the preorder of x in T. By storing additional information to each vertex of the top DAG, many operations can be supported [2]. For example, $Access(x)$ returns the label of x and $Decompress(x)$ returns the subtree $T(x)$ rooted at x. For a tree with n nodes, many operations in [2] are done in $\mathrm{O}(\log n)$ time, and $Decompress(\cdot)$ is done in $\mathrm{O}(\log n + |T(x)|)$ time. A pseudo code of $Access(\cdot)$ is in the full version [9].

4 top ZDD

We explain our top ZDD, which is a representation of ZDD by top DAG compression. Though it is easy to apply our compression scheme for general rooted DAGs, we consider only compression of ZDDs.

A ZDD $G = (V, E)$ is a directed acyclic graph in which nodes have labels $\ell(\cdot)$ (terminal nodes have $\bot$ and $\top$) and edges have labels 0 or 1. We can regard it as a graph with only edges being labeled. For each edge (u, v) of ZDD G, we define its label as a pair (edge label 0/1, $\ell(u) - \ell(v)$) if v is a branching node, or a pair (edge label 0/1, $\bot/\top$) if v is a terminal

node. In practice, we can use $c+1$ instead of $\bot$ and $c+2$ instead of $\top$ for the second element, where $c+1 = \ell(\bot) = \ell(\top)$. Below we assume ZDDs have labels for only edges, and 0-edge comes before 1-edge for each node.

Next we consider top trees for edge-labeled trees. The difference from node-labeled trees is only how to store the information for single edge clusters. In top trees, we stored labels for both endpoints of edges. We change this for storing only edge labels.

The top ZDD is constructed from a ZDD $G = (V, E)$ as follows.

1. We perform a depth-first traversal from the root of G and obtain a spanning tree T of all branching nodes. During the process, we do not distinguish 0-edges and 1-edges, and terminal nodes are not included in the tree. Nodes of the tree are identified with their preorders in T. If we say node u, it means the node in T with preorder u. We call edges of G not included in T as *complement edges*.
2. We convert the spanning tree T to a top tree $\mathcal{T}$ by the greedy algorithm.
3. For each complement edge (u, v), we store its information in a node of $\mathcal{T}$ as follows. If v is a terminal, let a be the vertex of the top tree corresponding to the cluster of single edge between u and its parent in T. Note that a is uniquely determined. Then we store a triple $((u, v)$, edge label 0/1, $\bot/\top)$ in a. If v is a branching node, we store the information of the complement edge to a vertex of T corresponding to a cluster containing both u and v. The information to store is a triple $((u, v)$, edge label 0/1, $\ell(u) - \ell(v))$. We decide a vertex to store it as follows. Let a, b be the vertices of the top tree corresponding to the clusters of single edges towards u, v in T, respectively. Then we store the triple in the lowest common ancestor $\mathrm{lca}(a, b)$ in T. Here the information (u, v) represents local preorders inside the cluster corresponding to $\mathrm{lca}(a, b)$. Note that $\mathrm{lca}(a, b)$ may not be the minimal cluster including both u and v.
4. We create a top DAG $\mathcal{T}D$ by DAG compression by sharing identical clusters. To determine identicalness of two clusters, we compare them together with the information of complement edges in the clusters stored in step 3. Complement edges which do not appear in multiple clusters are moved to the root of T.

Figure 2 shows an example of a top ZDD. The left is the original ZDD and the right is the corresponding top ZDD. Red and green edges show edges in the spanning tree and complement edges, respectively. In this figure we show for each vertex of the top DAG, the corresponding cluster, but they are not stored explicitly.

To achieve small space, it is important to use what data structure for representing each information. For example, we explained that each vertex of the top DAG stores the cluster size etc., this is redundant and the space can be reduced. Next we explain our space-efficient data structure which is enough to support efficient queries in detail.

4.1 Details of the data structure

We need the following information to recover the original ZDD from a top ZDD.

- Information about the structure of top DAG $\mathcal{T}D$.
- Information about each vertex of $\mathcal{T}D$. There are three types of vertices: vertices corresponding to a leaf of the top tree, vertices representing vertical merge, and vertices representing horizontal merge. For each type we store different information.
- Information about complement edges. The root or other vertices of $\mathcal{T}D$ store them.

We show space-efficient data structures for storing these information. In theory, we use the succinct bitvector with constant time rank/select support [14]. In practice, we use the SparseArray [13] to compress a bitvector if the ratio of ones in the bitvector is less than $\frac{1}{4}$,

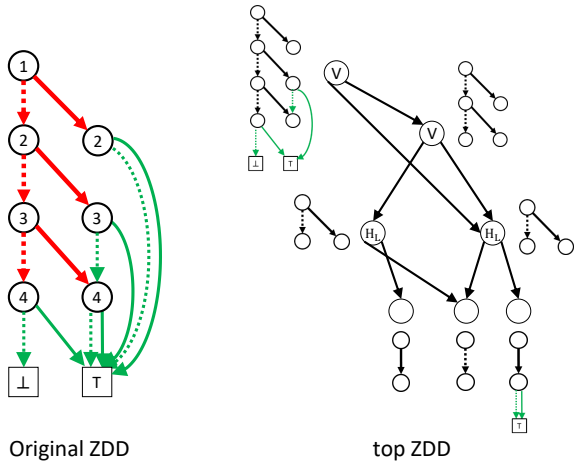


Figure 2 An example of a top ZDD. 0-edges and 1-edges are depicted by dotted and solid lines, respectively. Red edges are spanning tree edges and green edges are complement edges. For each vertex of the top DAG, the corresponding cluster and the information stored in the vertex are shown.

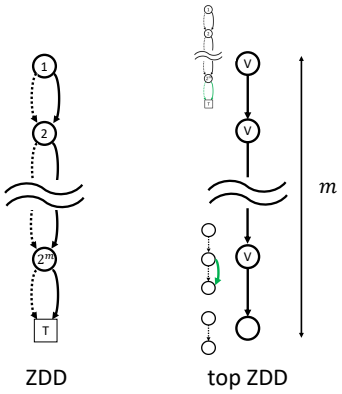


Figure 3 A top ZDD with $O(\log n)$ vertices, where $n = 2^m$.

and use the SparseArray for the bitvector whose 0/1 are flipped if the ratio of zeros is less than $\frac{1}{4}$. To store an array of non-negative integers, we use $\lfloor \log_2 m \rfloor$ bits for each entry where m is the maximum value in the array. Let n denote the number of internal nodes of a ZDD. We use $n+1, n+2$ to represent terminals $\bot, \top$, respectively.

4.1.1 The data structure for the structure of top DAG

We store top DAG $\mathcal{TD}$ after converting it to a tree. We make tree T' by adding dummy vertices to $\mathcal{TD}$. For each vertex x of $\mathcal{TD}$ whose in-degree is two or more, we do the following.

1. Let $a_1, \cdots, a_t$ be the vertices of $\mathcal{TD}$ from which there are edges towards x. Note that there may exist identical vertices among them corresponding to different edges. We create $t-1$ dummy vertices $d_1, \cdots, d_{t-1}$.
2. For each $1 \leq i \leq t-1$, remove edge (a_i, x_i) and add edge (a_i, d_i).
3. For each dummy vertex d_i, we store information about a pointer to x. In our implementation, we store the preorder of x in T' from which the dummy vertices are removed.

Then we can convert the top DAG to the tree T' and the pointers from the dummy vertices.

Next we explain how to store T' and the information about the dummy vertices. The structure of T' is represented by the BP sequence [12]. There are two types of leaves in T': those which exist in the original top DAG, and those for the dummy vertices. To distinguish them, we use a bitvector. Let m be the number of leaves in T'. We create a bitvector B_{dummy} of length m whose i-th bit corresponds to the i-th leaf of T' in preorder. We set $B_{dummy}[i] = 1$ if the i-th leaf is a dummy vertex, and we set $B_{dummy}[i] = 0$ otherwise.

We add additional information to dummy vertices to support efficient queries. We define an array *clsize* of length D where D is the number of dummy vertices. For the i-th dummy vertex in preorder, let s_i be the vertex pointed to by the dummy vertex. We define $clsize[k] = \sum_{i=1}^{k}$(the number of vertices in the cluster represented bys_i). That is, $clsize[k]$ stores the cumulative sum of cluster sizes up to k. This array is used to compute the cluster size for each vertex efficiently.

4.1.2 Information on vertices

We explain how to store information on vertices of T' except for dummy vertices.

Each vertex corresponding to a leaf in the original top tree is a cluster for a single edge in the spanning tree, and it is a non-dummy leaf in T'. We sort these vertices in preorder in T', and store information on edges towards them in the following two arrays. One is array *label_span* to store differences of levels between endpoints of edges. Let u and v be the starting and the ending points of the edge corresponding to the i-th leaf, respectively. Then we set $\mathit{label\_span}[i] = \ell(v) - \ell(u)$. The other is array *type_span* to store if an edge is 0-edge or 1-edge. We set $\mathit{type\_span}[i] = 0$ if the edge corresponding to the i-th vertex is a 0-edge, and $\mathit{type\_span}[i] = 1$ otherwise.

Each vertex of T' corresponding to vertical merge or horizontal merge is an internal vertex. We sort internal vertices of T' in their preorder. Then we make a bitvector B_H so that $B_H[i] = 0$ if the i-th vertex stands for vertical merge, and $B_H[i] = 1$ if it stands for horizontal merge. For vertices corresponding to horizontal merge, we do not store additional information. For vertices corresponding to vertical merge, we use arrays *preorder_diff* and *label_diff* to store the differences of preorders and levels between the top and the bottom boundary nodes of the merged cluster. Let x_i be the i-th vertex in preorder corresponding to vertical merge, cl_i be the cluster corresponding to x_i, t_i be the top boundary node of cl_i, and b_i be the bottom boundary node of cl_i. Note that t_i and b_i are nodes of the ZDD. Then we set $\mathit{preorder\_diff}[i] =$ (the local preorder of b_i inside cluster cl_i) and $\mathit{label\_diff}[i] = \ell(b_i) - \ell(t_i)$.

4.1.3 Information on complement edges

Complement edges are divided into two: those stored in the root of the top DAG and those stored in other vertices. We represent them in a different way.

First we explain the data structure for storing complement edges in the root of the top DAG. Let E_{root} be the set of all complement edges stored in the root. We sort edges of E_{root} in the preorder of their starting point. Orders between edges with the same starting point are arbitrary. For complement edges stored in the root, we store the preorders of their starting point using a bitvector B_{src_root}, the preorders of their ending point using an array *dst_root*, and edge labels 0/1 using an array *type_root*. The cluster corresponding to the root of top DAG is the spanning tree of the ZDD. For each node v of the spanning tree, we represent the number of complement edges in E_{root} whose starting point is v, using a unary code. We concatenate and store them in preorder in the bitvector B_{src_root}. For edges in E_{root} sorted in preorder of the starting points, we store the preorder of the ending point of the i-th edge in $\mathit{dst\_root}[i]$, and set $\mathit{type\_root}[i] = 0$ if the i-th edge is a 0-edge, and set $\mathit{type\_root}[i] = 1$ otherwise.

Next we explain the data structure for storing complement edges in vertices other than the root. Let E_{in} be the set of those edges. We sort the edges as follows.

1. We divide the edges of E_{in} into groups based on the clusters having the edges. These groups are sorted in preorder of vertices for the clusters.
2. Inside each cluster $cl(x)$, we sort the edges of E_{in} in preorder of starting points of the edges. For edges with the same starting point, their order is arbitrary.

We store the sorted edges of E_{in} using a bitvector B_{edge} and three arrays *src_in*, *dst_in*, and *type_in*. The bitvector B_{edge} stores the numbers of complement edges in vertices of T' by unary codes. The arrays *src_in*, *dst_in*, and *type_in* are defined as: $\mathit{src\_in}[i] =$ (the local preorder of the starting point of the i-th edge inside the cluster), $\mathit{dst\_in}[i] =$ (the local preorder of the ending point of the i-th edge inside the cluster), $\mathit{type\_in}[i] = 0$ if the i-th edge is a 0-edge, and $\mathit{type\_in}[i] = 1$ otherwise.

A top ZDD is composed of the following data structures:

- *bp*: a BP sequence representing the structure of T'
- B_{dummy}: a bitvector showing i-th leaf is a dummy vertex or not
- *clsize*: an array storing cumulative sum of cluster sizes of the first to the i-th dummy leaves
- *label_span*: an array storing differences of labels of ending points of i-th non-dummy leaf
- *type_span*: an array showing the edge corresponding to the i-th non-dummy leaf is 0-edge or not
- B_H: a bitvector showing i-th internal vertex is a vertical merge or not
- *preorder_diff*: an array storing differences of preorders between the top and the bottom boundary nodes of the vertex corresponding to i-th vertical merge
- *label_diff*: an array storing differences of labels between the top and the bottom boundary nodes of the vertex corresponding to i-th vertical merge
- B_{src_root}: a bitvector storing in unary codes the number of complement edges from each vertex
- *dst_root*: an array storing preorders of ending points of the i-th complement edge stored in root
- *type_root*: an array showing the i-th complement edge stored in the root is a 0-edge or not
- B_{edge}: a bitvector storing in unary codes the number of complement edges from each vertex stored in the root
- *src_in*: an array storing local preorders of starting points of i-th complement edge stored in non-root
- *dst_in*: an array storing local preorders of ending points of i-th complement edge stored in non-root
- *type_in*: an array showing the i-th complement edge stored in non-root is 0-edge or not

4.2 The size of top ZDDs

The size of top ZDDs heavily depends on not only the number of vertices in the spanning tree after top DAG compression, but also the number of complement edges for which we store some information. Therefore the size of top ZDDs becomes small if the number of nodes is reduced by top DAG compression and many common complement edges are shared.

In the best case, top ZDDs are exponentially smaller than ZDDs.

▶ **Theorem 6.** *There exists a ZDD with n nodes to which the corresponding top ZDD has $O(\log n)$ vertices.*

A detailed proof is in the full version [9]. A ZDD for a power set with $n = 2^m$ elements satisfies the claim. Figure 3 shows such a ZDD.

4.3 Operations on top ZDDs

We give algorithms for supporting operations on the original ZDD using the top ZDD. We consider the following three basic operations. We identify a node x of the ZDD with the vertex in the spanning tree T used to create the top ZDD whose preorder is x.

- $\ell(x)$: returns the label of a branching node x.
- $zero(x)$: returns the preorder of x_0, or returns $\bot$ or $\top$ if the node is a terminal.
- $one(x)$: returns the preorder of x_1, or returns $\bot$ or $\top$ if the node is a terminal.

We show $\ell(x)$ is done in $O(\log n)$ time and other operations are done in $O(\log^2 n)$ time where n is the number of nodes of the ZDD. Below we denote the vertex of T' stored in the top ZDD with preorder x by "vertex x of T'".

First we explain how to compute $\ell(x)$ in $O(\log n)$ time. We can compute $\ell(x)$ recursively using a similar algorithm to those on the top DAG. A difference is that we assumed that each vertex of the top DAG stores the cluster size, while in the top ZDD it is not stored to reduce the space requirement. Therefore, we have to compute it using the components of the top ZDD.

To work the recursive computation, we need to compute the cluster size $size(x')$ represented by vertex x' of T' efficiently. We can compute $size(x')$ by the number of non-dummy leaves in the subtree of T' rooted at x', and the sizes of the clusters corresponding to dummy leaves in the subtree rooted at x'. If we merge two clusters of size a and b, the resulting cluster has size $a+b-1$. Therefore if we merge k clusters whose total size is S, the resulting cluster after $k-1$ merges has size $S-k+1$. These values can be computed from the BP sequence bp of T', the array *clsize*, and the bitvector B_{dummy}. By using bp, we can compute the interval $[l, r]$ of leaf ranks in the subtree rooted at x'. Then, using B_{dummy}, we can find the number c of non-dummy leaves and the interval $[l', r']$ of non-dummy leaf ranks, in the subtree of x'. Because *clsize* is the array for storing cumulative sums of cluster sizes for dummy leaves, the summation of sizes of clusters corresponding to l'-th to r'-th dummy leaves is obtained from $clsize[r'] - clsize[l'-1]$. Because the size of a cluster for a non-dummy leaf is always 2, the summation of cluster sizes for non-dymmy leaves is also obtained. This can be done in constant time. A pseudo code for computing $size(x')$ is in the full version [9].

Using the function $size(x')$, we can compute a recursive function similar to the algorithm of $Access(\cdot)$. Instead of $D(\cdot)$ in Algorithm 1 [9], we use *preorder_diff*. When we arrive at a dummy leaf, we use a value in *dst_dummy* to move to the corresponding internal vertex of T' and restart the recursive computation. Then for the vertex of the original ZDD whose preorder in T is x, we can obtain the leaf of T' corresponding to the cluster of a single edge containing x.

To compute $\ell(x)$, we traverse the path from the root to the leaf corresponding to the cluster containing x. First we set $s = 1$. During the traversal, if the current vertex is for vertical merge and the next vertex is its right child, that is, the next cluster is in the bottom, we add the *label_diff* value of the top cluster to s. The index of *label_diff* is computed from B_H and bp. When we reach the leaf p', if x is its top boundary node, it holds $\ell(x) = s$, otherwise, let $k = leaf\_rank(p')$, then we obtain $\ell(x) = s + label\_span[k - rank_1(B_{dummy}, k)]$. Because each operation is $O(1)$ time and the height of the top DAG is $O(\log n)$, $\ell(x)$ is $O(\log n)$ time.

Next we show how to compute $y = zero(x)$. We can compute $one(x)$ in a similar way. We do a recursive computation as operations on top DAG, A difference is how to process complement edges. There are two cases: if the 0-edge from x is in the spanning tree or not. If the 0-edge from x is in the spanning tree, the edge is stored in a cluster with a single edge (x, y). The top boundary node of such a cluster is x. Therefore we search clusters whose top boundary node is x. If the 0-edge from x is not in the spanning tree, it is a complement edge and it is stored in some vertex on the path from a cluster C with a single edge whose bottom boundary node is x to the root. Therefore we search for C.

First we recursively find a non-dummy leaf of T' whose top boundary node is x. During this process, if there is a vertex whose top boundary is x and its cluster contains more than one edge and corresponds to horizontal merge, we move to the left child, because the 0-edge from x must exist in the left cluster. If we find a non-dummy leaf of T' which corresponds to a cluster with a single edge and its top boundary node is x, its bottom boundary node is

$y = zero(x)$. We climb up the tree until the root to compute the global preorder of y. If there does not exist such a leaf, the 0-edge from x is not in the spanning tree. We find a cluster with a single edge whose bottom boundary node is x. From the definition of the top ZDD, the 0-edge from x is stored in some vertices visited during the traversal. Because complement edges stored in a cluster are sorted in local preorders inside the cluster of starting points, we can check if there exists a 0-edge whose starting point is x in $O(\log n)$ time. If it exists, we obtain the local preorder of y inside the cluster. By going back to the root, we obtain the global preorder of y. Note that complement edges for all clusters are stored in one array, and therefore we need to obtain the interval of indices of the array corresponding to a cluster. This can be done using B_{edge}. In the worst case, we perform a binary search in each cluster on the search path. Therefore the time complexity of $zero(x)$ is $O(\log^2 n)$.

5 Experimental Comparison

We compare our top ZDD with existing data structures. We implemented top ZDD with C++ and measured the required space for storing the data structure. For comparison, we used the following three data structures.

- top ZDD (proposed): we measured the space for storing the all components of a top ZDD.
- DenseZDD [5]: data structures for representing a ZDD using succinct data structures. Two data structures are proposed; one support constant time queries and the other has $O(\log n)$ time complexity. We used the latter that uses less space.
- a standard ZDD: ZDDs that are implemented naively. We store for each node its label and two pointers corresponding to a 0-edge and a 1-edge. The space is $2n\lfloor \log n \rfloor + n\lfloor \log c \rfloor$ bits where n is the number of nodes of a ZDD and c is the size of the universal set.

We constructed ZDDs for various data with different settings. The results are shown in Table 1. The unit of size is bytes.

We found that for all data sets, the top ZDD uses less space than the naive representation of the standard ZDD. We also confirmed that the data 1, 2, and 3 can be compressed very well by top ZDDs. For any settings on the data 4, the top ZDD uses less space than the DenseZDD, and for some cases the memory usage of the top ZDD is almost $\frac{1}{2}$–$\frac{2}{3}$ of that of the DenseZDD. For the data 5 and 6, There are a few case that the DenseZDD uses less space than the top ZDD.

The results above are for monotone set families, that is, any subset of a set a the family also exists in the family. The data 7 and 8 are non-monotone set families. For the families of paths on $n \times n$ grid graphs, the top ZDD uses less space than the DenseZDD, and for $n = 9$, the top ZDD uses about $\frac{1}{3}$ the memory of DenseZDD. On the other hand, for the data 8, the top ZDD uses about 10 % more space than the DenseZDD. From these experiments we confirmed that the top ZDD uses less space than the DenseZDD for many set families.

The experiments to compare construction time and edge traversal time of top ZDDs and DenseZDDs are conducted in the full version [9]. The results show that DenseZDDs are several times faster on construction and several dozens faster on traversal than top ZDDs. Note that the traversal time is $\Theta(\log n)$ on top ZDDs, but is linear to a query size on DenseZDDs.

6 Concluding Remarks

We have proposed top ZDD to compress a ZDD by regarding it as a DAG. We compress a spanning tree of a ZDD by the top DAG compression, and compress other edges by sharing them as much as possible. We proved that the size of a top ZDD can be logarithmic of that

Table 1 Results of experiments.

Data	Setting	top ZDD	DenseZDD	ZDD
1. The power set of $\{1,\ldots,A\}$	$A = 1000$	**2,297**	4,185	3,750
	$A = 50000$	**2,507**	178,764	300,000
2. For $C = \{1,\ldots,A\}$, $\{S \subseteq C \mid \max_{a\in S} a - \min_{a\in S} a \leq B\}$	$A = 500, B = 250$	**2,471**	227,798	321,594
	$A = 1000, B = 500$	**2,551**	321,594	1,440,375
3. For $C = \{1,\ldots,A\}$, the family of sets $\{S \subseteq C \mid \lvert S\rvert \leq B\}$	$A = 100, B = 50$	**3,863**	9,544	9,882
	$A = 400, B = 200$	**13,654**	146,550	206,025
	$A = 1000, B = 500$	**43,191**	966,519	1,440,375
4. Knapsack set families with random weights. A is the number of elements, W is the maximum weight of an element, C is the capacity. (Setting is given as (A, W, C).)	$(100, 1000, 10000)$	**1,659,722**	1,730,401	2,444,405
	$(200, 100, 5000)$	**1,032,636**	1,516,840	2,181,688
	$(1000, 100, 1000)$	**2,080,965**	2,929,191	4,491,025
	$(5000, 100, 200)$	**1,135,653**	1,740,841	2,884,279
	$(1000, 10, 1000)$	**1,383,119**	2,618,970	3,990,350
	$(1000, 100, 1000)$	**565,740**	656,728	1,056,907
5. The family of edge sets which are matching of a given graph	8×8 grid	**12,246**	16,150	18,014
	complete graph K_{12}	23,078	**16,304**	25,340
	Interoute	**30,844**	39,831	50,144
6. Set families of frequent item sets with a minimum frequency	*mushroom* (0.1%)	104,774	**91,757**	123,576
	retail (0.025%)	**59,894**	65,219	62,766
	T40I10D100K (0.5%)	**177,517**	188,400	248,656
7. Families of edge sets in $n \times n$ grid graph which are paths from the bottom left to the top right	$n = 6$	**17,194**	28,593	37,441
	$n = 7$	**49,770**	107,529	143,037
	$n = 8$	**157,103**	401,251	569,908
	$n = 9$	**503,265**	1,465,984	2,141,955
8. Families of solutions of the n-queen problem	$n = 11$	40,792	**35,101**	45,950
	$n = 12$	183,443	**167,259**	229,165
	$n = 13$	866,749	**799,524**	1,126,295

of the ZDD. We also showed that navigational operations on a top ZDD are done in time polylogarithmic to the original ZDD size. Experimental results show that the top ZDD is always smaller than the ZDD, and uses less space than the DenseZDD for most of the data.

Future work will be as follows. First, in the current construction algorithm, we create a spanning tree of ZDD by a depth-first search, but this may not produce the smallest top ZDD. For example, if we choose all 0-edges, we obtain a spanning tree whose root is the terminal $\top$, and this might be better. Next, in this paper we considered only traversal operations and did not give advanced operations such as choosing the best solution among all feasible solutions based on an objective function. Lastly, we considered only compressing ZDDs, but our compression algorithm can be used for compressing any DAG. We will find applications of our compression scheme.

References

1 Stephen Alstrup, Jacom Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005. doi:10.1145/1103963.1103966.

2 Philip Bille, Inge Li Gørtz, Gad M.Landau, and Oren Weimann. Tree compression with top trees. *Information and Computation*, 243:166–177, 2015. doi:10.1016/j.ic.2014.12.012.

3 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986. doi:10.1109/TC.1986.1676819.

4 Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 141–152. Morgan Kaufmann, 2003. doi:10.1016/B978-012722442-8/50021-5.

5 Shuhei Denzumi, Jun Kawahara, Koji Tsuda, Hiroki Arimura, Shin ichi Minato, and Kunihiko Sadakane. Densezdd: A compact and fast index for families of sets. In *Proceedings of the 13th International Symposium on Experimental Algorithms*, pages 187—198. Springer Verlag, 2014. doi:10.1007/978-3-319-07959-2_16.

6 Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of ACM*, 27(4):758–771, 1980. doi:10.1145/322217.322228.

7 Markus Frick, Martin Grohe, and Christoph Koch. Query evaluation on compressed trees. In *Proceedings of 18th Annual IEEE Symposium of Logic in Computer Science*, LICS 2003, pages 188–197. IEEE Computer Society, 2003. doi:10.1109/LICS.2003.1210058.

8 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

9 Kotaro Matsuda, Shuhei Denzumi, and Kunihiko Sadakane. Storing set families more compactly with top zdds, 2020. arXiv:2004.04586.

10 Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference*, pages 272–277. ACM, 1993. doi:10.1145/157485.164890.

11 Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 52–57, 1990. doi:10.1145/123186.123225.

12 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3), 2014. doi:10.1145/2601073.

13 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments*, 2007. doi:10.1137/1.9781611972870.6.

14 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43–es, 2007. doi:10.1145/1290672.1290680.

Fast and Simple Compact Hashing via Bucketing

Dominik Köppl
Department of Informatics, Kyushu University, Japan Society for Promotion of Science (JSPS), Fukuoka, Japan
https://dkppl.de/
dominik.koeppl@inf.kyushu-u.ac.jp

Simon J. Puglisi
Helsinki Institute for Information Technology, Espoo, Finland
Department of Computer Science, University of Helsinki, Finland
puglisi@cs.helsinki.fi

Rajeev Raman
School of Informatics, University of Leicester, United Kingdom
r.raman@leicester.ac.uk

Abstract

Compact hash tables store a set S of n key-value pairs, where the keys are from the universe $U = \{0, \ldots, u-1\}$, and the values are v-bit integers, in close to $\mathcal{B}(u,n) + nv$ bits of space, where $\mathcal{B}(u,n) = \log_2 \binom{u}{n}$ is the information-theoretic lower bound for representing the set of keys in S, and support operations insert, delete and lookup on S.

Compact hash tables have received significant attention in recent years, and approaches dating back to Cleary [IEEE T. Comput, 1984], as well as more recent ones have been implemented and used in a number of applications. However, the wins on space usage of these approaches are outweighed by their slowness relative to conventional hash tables. In this paper, we demonstrate that compact hash tables based upon a simple idea of bucketing practically outperform existing compact hash table implementations in terms of memory usage and construction time, and existing fast hash table implementations in terms of memory usage (and sometimes also in terms of construction time).

A related notion is that of a compact *Hash ID map*, which stores a set $\hat{S}$ of n keys from U, and implicitly associates each key in $\hat{S}$ with a unique value (its ID), chosen by the data structure itself, which is an integer of magnitude $O(n)$, and supports inserts and lookups on $\hat{S}$, while using close to $\mathcal{B}(u,n)$ bits. One of our approaches is suitable for use as a compact Hash ID map.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases compact hashing, hash table, separate chaining

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.7

Related Version http://arxiv.org/abs/1905.00163

Supplementary Material Implementations are available at https://github.com/koeppl/separate_chaining. Our benchmarks for the operations insert, lookup, and delete are available at https://github.com/koeppl/hashbench.

Funding *Dominik Köppl*: JSPS KAKENHI Grant Number JP18F18120.
Simon J. Puglisi: Academy of Finland Grant 319454.

Acknowledgements We thank Marvin Löbel for the implementations of the Bonsai tables with the additional support of a sparse table layout.

1 Introduction

In this paper, we consider practical *compact* representations of dynamic *dictionaries*. A dictionary is arguably the single most important abstract data type, formulated as follows. We are given a dynamic set S of key-value pairs $\langle x, y\rangle$, where the key x comes from a *universe* $U = [u]$[1] and the value y is from $[2^v]$. Furthermore, all keys in S are distinct. A dictionary supports the following operations:

lookup(x, S)**:** Given $x \in U$, if there is a pair $\langle x, y\rangle$ in S, return y, or a null value otherwise.
insert$(\langle x, y\rangle, S)$**:** Add the pair $\langle x, y\rangle$ to S if S does not have x as a key.
delete(x, S)**:** Delete the pair (if any) of the form $\langle x, y\rangle$ from S.

Dictionaries can be implemented using a number of data structures such as hash tables and balanced trees, and many standard libraries use these approaches. Our interest, however, is in highly space-efficient approaches to the dictionary problem. In the worst case, a dictionary cannot use less space than the information-theoretic lower bound needed to store S. If $|S| = n$, the lower bound for storing the keys in S is $\mathcal{B}(u, n) = \lg \binom{u}{n} = n \lg u - n \lg n + O(n)$ bits (in what follows we abbreviate $\mathcal{B}(u, n)$ by $\mathcal{B}$). The lower bound on the space for the key-value pairs in S is thus $\mathcal{B} + nv$ bits. Following standard terminology, we refer to dictionaries that use $O(\mathcal{B} + nv)$ bits as *compact* and those that use $\mathcal{B} + nv + o(\mathcal{B} + nv)$ bits as *succinct*. In recent work [3, 15], a number of applications have been highlighted for compact dictionaries including compact representations of graphs, tries and arrays containing variable-length entries. In all these applications, the $\Omega(n(\lg u + v))$-bit space usage of a traditional dictionary (even those with low wasted space such as [10, 12]) is prohibitively large.

Building on earlier work on succinct *static* dictionaries [4, 14], succinct dynamic dictionaries were proposed by Raman and Rao [18] and Arbitman et al. [1]. In the transdichotomous model with word size $w = \lg u$ bits, the above solutions use $(\mathcal{B} + nv)(1 + o(1))$ bits of space, answer lookup queries in $O(1)$ worst-case time, and perform updates in $O(1)$ expected amortized or worst-case time[2]. A slightly different data structure using $O(\mathcal{B} + nv)$ bits of space was discussed in [8]. Finally, Blandford and Blelloch [3] generalized the notion of $\mathcal{B}$ when keys are variable-length bit-strings of length at most w, and gave a dictionary with a space usage of $O(\mathcal{B} + nv)$ bits. However, these data structures are complex, and although Arbitman et al. [1] discussed ideas to make their data structure more practical, we are not aware of any implementations along these lines.

We are concerned with practical approaches to compact dynamic dictionaries. A practical solution by Blandford and Blelloch [2] uses $O(\mathcal{B} + nv)$ bits of space, but takes $O(\lg n)$ time to perform (a much wider range of) operations. The only other practical compact dictionary we are aware of is Cleary's *compact hash table (CHT)* [6]. For any constant $\epsilon > 0$, Cleary's CHT uses $(1+\epsilon)n(\lg(u/n)+v)+O(n) = (1+\epsilon)(\mathcal{B}+nv)+O(n)$ bits and supports lookup in $O(1/\epsilon^2)$ expected time, and updates in $O(1/\epsilon^3)$ expected amortized time. Poyias et al. [16] proposed a variant of the CHT, called the *displacement* CHT or dCHT, which supports lookup in $O(1/\epsilon)$ expected time. However, it uses $\Omega(n)$ bits more space than the CHT (a simplified dCHT [16] in fact takes $\Theta(n \lg^{(5)} n)$ bits[3] more space than the CHT), and particularly as ϵ approaches 0, the practical performance of these approaches deteriorates significantly.

Related to a dynamic dictionary is the notion of a *hash ID map*, which stores a set $\hat{S}$ of n keys (without user-provided values) from a universe U, and associates each key in $\hat{S}$ with a unique integer, chosen by the data structure, from a range $[\rho]$. If $x \in \hat{S}$, lookup(x) returns

[1] For non-negative integers i, $[i] = \{0, 1, \ldots, i-1\}$.
[2] These two results differ regarding the model of dynamic memory allocation used.
[3] Throughout lg denotes the logarithm to base two with $\lg^{(1)} n = \lg n$ and $\lg^{(i)} = \lg(\lg^{(i-1)} n)$ for $i > 1$.

the integer associated with x. A requirement is that the integer associated with x does not change during the lifetime of the data structure, although our solution, as well as the previous solutions, require that the data structure is destroyed and rebuilt after $\Theta(n)$ update operations. In addition, we would like ρ to be not "much" more than n. Finally, the space usage of a compact hash ID map should be close to $\mathcal{B}(u, n)$, which is the information-theoretic lower bound for storing $\hat{S}$. A compact hash ID map has many applications, including compact representations of tries [16, 11] (and potentially other compact data structures), LRU cache management [19], and naming in string processing [13]. Implicit in the work of Darragh et al. [7] is a compact hash ID map whose space usage is $(1+\epsilon)\mathcal{B}$ bits, has $\rho = \Theta(n \lg n / \lg\lg n)$ with high probability, supports $O(\epsilon n)$ updates before requiring rebuilding, and performs insert and lookup in $O(1/\epsilon^2)$ expected time. Implicit in the work of Poyias et al. [16] is a compact hash ID map whose space usage is $(1+\epsilon)\mathcal{B}$ bits, has $\rho = O(n)$, supports $O(\epsilon n)$ updates before requiring rebuilding, and performs insert and lookup in $O(1/\epsilon)$ expected time. Here $\epsilon > 0$ is a user-specified constant.

In this paper we consider the use of *bucketing* to design practical CHTs and compact hash ID maps, an approach that is distinguished by its simplicity, and is the basis of the theoretical work of Raman and Rao [18], as well as earlier CHTs.

Our Contributions. We propose simple and practical CHTs with:

- $\mathcal{B} + nv + O(n \lg\lg u)$ bits, performing insert and lookup in $O(\lg u)$ expected time. This approach, despite being theoretically inferior, is extremely simple. It also yields a compact hash ID map with $\rho = O(n)$, $\mathcal{B} + O(n \lg\lg u)$ bits of space usage with support for $\Omega(n)$ updates before requiring rehashing, with the same operation complexities as above.
- $\mathcal{B} + nv + O(n)$ bits, performing insert in $O(\lg u)$ worst-case time and lookup in $O(1)$ expected time.

Despite their poor asymptotic complexities, these approaches are simple and designed for practical performance, which we verified in the evaluation described in Section 4. In this evaluation, we consider two distinct scenarios:

- $\lg(u/n)$ and the value bit width v are both small (using only a few bits), motivated by the CDRW-array [16], which compactly stores a dynamic array A, most of whose entries can be stored in very few bits. For every integer $v \geq 1$, the CDRW-array takes all indices in A containing v-bit values, and stores the key-value pair $(i, A[i])$ in a CHT for that integer v. For small v, the set of indices containing v-bit values are often a large proportion of all indices, so "$\lg u/n$" is small as well. In this scenario, keeping space usage low is a priority.
- $\lg(u/n)$ and the value bit width v are both relatively large. This models a number of scenarios such as storing the adjacency matrix of a fairly sparse weighted or labeled graph. In this scenario, we would be competing against other memory-efficient implementations of conventional hash tables, and speed would also be an important criterion.

On the implementation side, we offer two novel contributions. Firstly, our hash table needs to be periodically resized as elements are added. Rather than resizing based on the overall number of keys, we resize based on the size of the maximum bucket. Secondly, we use SIMD accelerated techniques [20] to accelerate searches in a bucket.

Discussion. We briefly summarize how our approaches differ from other compact hash tables. Hash tables are usually implemented either (a) as *open addressing*, whereby all keys are stored in a single table, or (b) as *closed addressing*, where keys are mapped to buckets. However, combining open addressing with compact hashing leads to difficulties, as compact hashing does not store entire keys, but only *quotient* information [6], and the

overhead of storing and maintaining additional information to recover the keys from the quotients is quite high, since open addressing schemes need to have some mechanism for resolving collisions, such as linear probing [6, 16]. The problem is exacerbated by the use of quite high *load factors* to keep the space usage low. The use of high load factors can be avoided by switching to a *sparse* hash table layout[4]. In contrast to standard open addressing (storing elements in a single array), the sparse hash table layout uses a bit-string to mark positions at which elements are stored, and represents the keys themselves in a collection of small, variable-sized arrays. This allows low load factors with a moderate space overhead, and works well with compact hashing [9, Outlook]. However, the overhead of decoding the displacement information for restoring a key from its quotient remains a concern.

In contrast, closed addressing does not use collision resolution; all keys are simply stored in their buckets. The usual representation of a bucket is via *chaining*, i.e., storing a bucket as a linked list, as in the `unordered_map` of the C++ standard library `libstdc++` [5, Sect. 22.1.2.1.2]. The overhead of chaining can make hash tables using it space-consuming and slow, and modern implementations of hash tables tend to focus on open addressing.

In a compact hash setting, the buckets contain quotients of keys. In contrast to hashing via open addressing, we do not need to maintain any information to recover a key from its quotient. However, buckets have obvious overheads such as pointers to them and auxiliary information such as their sizes. The challenge is how to balance the overheads of the buckets (which grow as the number of buckets increases) with the size of the quotients stored in the buckets (which reduces as the numbers of buckets increase). Theoretical solutions (such as [18]) to this problem are complex (e.g., recursing on the quotients in a bucket). We give up on asymptotic worst-case performance in order to find solutions that work in practice.

Paper Overview. We begin with a theoretical description of our algorithms in Section 2. In Section 3 we describe the implementations, which depart from theory in some ways. Section 4 describes the experimental evaluation and Section 5 concludes and states directions for further work.

2 Compact Hashing via Bucketing: a Theoretical View

In this section, we outline the approach that we take from a theoretical perspective. This section assumes the *transdichotomous model* with word size $w = \lg u$ bits. For simplicity we consider insertion-only hash tables. Let N be a parameter with $N \leq n < 2N$ (we discuss what happens when this is violated in Section 2.3), and b the number of buckets of our hash table. We start with an invertible function $f : [u] \to [u]$, and use f to map key-value pairs to the b buckets. We assume that b, the size of the universe u, and N are powers of two.

Now, given a key-value pair $\langle x, y\rangle$, we compute $f(x)$, and assign $\langle x, y\rangle$ to the bucket $r = f(x) \bmod b$. In the bucket r, the key x is represented by its quotient value $q = f(x) \operatorname{div} b$; observe that the key x can be recovered as $f^{-1}(qb + r)$ and that the quotient value takes $\lg u - \lg b$ bits. We analyze the hash tables under the assumption of uniform and fully independent hashing. The key design decisions are the representation of the buckets and the choice of b, both of which we discuss in Section 2.3.

[4] `https://github.com/sparsehash/sparsehash`

2.1 Simple Compact Hashing

We start with the simple variant, which maintains $b = O(N/\lg u)$ buckets. The buckets are represented by an array of b pointers, each pointing to an array of w-bit words that stores the key-value pairs of the respective bucket. The overhead per bucket is $O(w)$ bits; summed over all buckets this overhead is $O(n)$ bits. Since each key is represented by a quotient using $\lg u - \lg b = \lg(u/N) + \lg\lg u$ bits, the overall space bound is $\mathcal{B} + nv + O(n \lg\lg u)$ bits as claimed, where v is the bit width of the values. To perform a lookup, an entire bucket is scanned. To perform an insert operation, a new array of the appropriate size is allocated, the existing bucket is copied over to the new array, and the new key is added to the end of the new array. A standard argument [17] shows that the maximum bucket size $b_{\max}$ is bounded by $O(\lg u)$ with probability $1 - 1/u^{O(1)}$. The running times of insert and lookup are therefore $O(\lg u)$ with high probability.

To use this as a compact hash ID map, we fix $b_{\max} = c \lg u$ for a constant $c > 0$ chosen large enough such that the probability of every bucket exceeding $b_{\max}$ is at most $(1/u)^2$ [17]. Suppose that a key[5] is stored at the j-th position of the i-th bucket and $j < b_{\max}$. Then the ID associated with this key is just $i \cdot b_{\max} + j$. We discuss the length of the "lifetime" of this data structure below.

2.2 Space-Efficient Compact Hashing

Our space-efficient variant maintains $b = N$ *sub-buckets*. Each sub-bucket is not represented individually, but $\lg u$ sub-buckets are *grouped* together in a bucket, which we call *group-bucket* in the following to avoid confusion. A group-bucket, with a total of s key-value pairs in it, is represented as an array of size s. In this array, all keys hashed to the same sub-bucket are stored in a consecutive range, and a bit-string of length $\lg u + s$ demarcates the sub-bucket boundaries by concatenating the sizes of the sub-buckets, written in unary, for example. The overheads of this approach (including the bit-string) are at most $O(n)$ bits. However, the quotients stored in each sub-bucket now only take $\lg u - \lg N = \lg(u/n) + O(1)$ bits. Thus, the overall space usage is $\mathcal{B} + nv + O(n)$ bits. We now consider the time for an operation. Using the same argument as in the previous section (Section 2.1), we can see that the size s of each group-bucket is at most $O(\lg u)$ with high probability, and the bit-string is therefore of length $O(\lg u)$ bits with high probability as well, which means that the range in the group-bucket corresponding to a sub-bucket can be located with a broadword select operation in $O(1)$ time with high probability. Since the expected sub-bucket size is $O(1)$, search within a sub-bucket takes $O(1)$ expected time, and lookup is consequently supported in $O(1)$ expected time. insert is done by rewriting the entire group-bucket each time, and takes $O(\lg u)$ time with high probability as a result. (Since each insert causes a group-bucket to be re-written, this approach is not suitable for use as a hash ID map.)

2.3 Rebuilding

Insertions will cause the invariant $N \le n < 2N$ to be violated. A standard approach would double N and rehash. From a coding perspective, however, it can speed up search if it is known that all buckets have a fixed maximum size $b_{\max}$ that stays unchanged throughout the execution of the algorithm – for example, a loop that searches through a bucket can be unrolled. Asymptotically, $b_{\max}$ cannot be a constant. Even if we use N buckets, it is well known that some bucket will have $\Theta(\log n / \log\log n)$ keys [17, Thm. 1].

[5] More precisely, the quotient of this key.

Despite this, we set $b_{\max}$ to be a fixed parameter, and let b be the current number of buckets. As soon as any bucket exceeds $b_{\max}$ in size we rebuild the data structure by doubling b. When we do this, the keys in the old bucket i are distributed at random into new buckets $2i$ and $2i+1$ (details in the next section). Once the rebuilding is complete, the insertions can resume. We say that a value of $b_{\max}$ is *stable* for a particular range of n, if the space between rebuildings is roughly $\Theta(n)$.

The stability can be improved through the use of *overflow* tables: when a new key is inserted and is hashed to a bucket of size $b_{\max}$, we put the key-value pair into a single global hash table, the *overflow table* (obviously, when searching, we may need to search the overflow table as well). The benefits of using an overflow table are as follows.

If we throw $n \ln n$ balls into n buckets randomly, the expected maximum bucket size is approximately $e \ln n$ [17]. Heuristically, this would suggest $b_{\max}$ should be set to be three times the average bucket size to ensure stability. However, the probability that a bucket exceeds $\ln n + O(\sqrt{\lg n})$ balls is $1/(\log n)^{O(1)}$. Thus, we can set $b_{\max}$ to much closer to the average bucket size. By doing so, the number of keys that go to the overflow table would only be $n/(\log n)^{O(1)}$, and the overflow table would therefore not be a major drag on either the time or the space performance of the hash table. This also suggests that with the appropriate parameter settings, a given constant $b_{\max}$ will continue to be stable for a *much* larger value of n (possibly quadratically bigger).

In addition, an overflow table can reduce ρ for compact hash ID maps. Keys inserted into the overflow table are given consecutive IDs beginning at $b \cdot b_{\max}$ and stored together with their IDs in an overflow table. If we reduce $b_{\max}$ by a constant factor for a given n, while placing a few keys into the overflow table, ρ will also be reduced by a constant factor.

3 Implementation

We implemented the simple approach and the space-efficient approach described in Sections 2.1 and 2.2, and call the implementations cht and grp, respectively. Each of the two implementations maintains b buckets, where b is a power of two. Following the discussion of different invertible functions given in [9, Sect. 3.2], we select a fixed multiplicative function $f : [u] \to [u]$ for the experiments. We use the last $\lg b$ bits of its return value for the index of the assigned bucket and the other bits for the quotient.

3.1 Maximum Bucket Size and Rehashing

In both cases we choose $b_{\max} = 255$. Our simulations suggest that $b_{\max} = 63$ is almost adequate for n up to 10^9. Intuitively, for stability $b_{\max}$ should be logarithmically dependent on n, so it would appear that $b_{\max} = 255$ should ensure stability for n up to 10^{30}, even without the use of overflow tables. The larger choice of $b_{\max}$ also helps to reduce the per-bucket overhead (at least 17 bytes, see Section 3.2).

When we try to insert an element into a bucket of size $b_{\max}$, we create a new hash table with twice the number of buckets and move the elements from the old table to the new one, bucket by bucket. After a bucket of the old table becomes empty, we can free up its memory, thus significantly reducing the peak memory during resizing. Note that the rebuilding is particularly efficient since the contents of bucket i in the old data structure is distributed between buckets $2i$ and $2i+1$ in the new data structure (the last $1 + \lg b$ bits of a hash value now determine the bucket).

The main additional parameter in grp versus cht is the number of sub-buckets in a group-bucket. We also deviate from theory with respect to the number of sub-buckets assigned to a group-bucket, which we call m: Given that the average number of total elements in a

group-bucket is s, a bit-string demarcating the sub-bucket boundaries in a group-bucket costs us $m+s$ bits on average, and therefore a group-bucket costs us $m+s+sv+sq$ bits on average, where $q = \lg u - \lg b$ is the quotient bit width. By doubling the number of sub-buckets per group-bucket (but keeping the total number of group-buckets), the quotient bit width decreases by one such that the average space of a group-bucket becomes $2m+s+sv+s(q-1)$ bits, which is smaller than the original size if $m < s$. However, changing m has an effect on s. In our experiments, determining m by counting s in the old hash table during a rehashing resulted in an overestimation of s. Therefore, we stuck with the empirically evaluated constant $m = 64$.

3.2 Buckets

We represent a bucket (resp. group-bucket for grp) as a quotient and a value bucket. The quotient bucket stores the quotients bit-compact in a byte array by using bit operations. Consequently, the number of bits used by a quotient bucket is quantized at eight bits (the last byte of the array might not be full). We resize a bucket with the C function `realloc`. Whether we need to resize a bucket on inserting an element depends on the policy we follow. With the incremental policy, we increase the size of the bucket to be exactly one element longer, just enough for a new element to fit in. This policy saves memory as only the minimum required amount of memory is allocated. Because buckets store at most $b_{\max}$ elements, the resize takes $O(b_{\max})$ time. In practice, much of the time for resizing depends on the speed of the memory allocator. Our second resize policy, *half increase*, increases the bucket size by 50%, taking the burden off the allocator at the expense of having some unused memory.[6]

4 Experiments

We implemented the approaches described in Section 2.1 and Section 2.2 in C++17, and refer to them cht and grp, respectively. We subscript cht with '++' or '50' to indicate whether the hash table resizes a bucket by, respectively, one element or by 50% (cf. Section 3.2). Implementations are available at `https://github.com/koeppl/separate_chaining`.

Evaluation Setting. Our experiments were run on an Ubuntu Linux 18.04 machine equipped with 32 GiB of RAM and an Intel Xeon CPU E3-1271 v3 clocked at 3.60GHz, having, respectively, 32KB, 256KB, and 8192KB of L1, L2, and L3 cache. We measured memory usage by instrumenting calls to `malloc`, `realloc`, and `free`. The compiler was g++-7.5.0 with flags `-O3 -DNDEBUG -march=native`. Our benchmarks for the operations insert, lookup, and delete are available at `https://github.com/koeppl/hashbench`.

4.1 Bonsai Tables

We compare our implementations cht and grp with practical implementations of compact hash tables implemented in the tudocomp project[7]. The first is called cleary, which is an implementation of Cleary's CHT [6] using linear probing. The second, called layered, is based on the dCHTs of Poyias et al. [16]: layered stores the displacement information in

[6] Since grp is the more memory-efficient variant, there is no grp_{50}. Moreover, since a group-bucket of grp must support insertions at all ending positions of its sub-buckets, such an insertion is much more involving than merely appending an element to a bucket of cht. We are unsure whether a different resize policy pays off.

[7] `https://github.com/tudocomp/compact_sparse_hash`

two associative array data structures. The first is an array storing 4-bit integers, and the second is an `unordered_map` (the C++ STL hash table implementation) for displacements larger than 4 bits. All tables apply linear probing and support a sparse table layout. We refer to these methods collectively as *Bonsai tables*, and append in subscript 'P' or 'S' if the respective variant is in its plain or sparse form, respectively. We used a maximum load factor of 0.95 for all Bonsai table implementations.

4.1.1 Insertions and Lookups

Our first and main experiment measures the time and space requirements when (a) filling the hash tables with data (insert) and (b) querying the data afterwards (lookup). We filled the hash tables with 32-bit keys and 1-bit values, with keys generated via `std::rand`. Figure 1 shows the measured time and peak memory usage when inserting an increasing number of elements into the hash tables (*construction* in the plots) and also times for lookup queries.

Time. Considering construction time, $\mathsf{layered}_{\mathrm{P}}$ and cht_{50} are the fastest options, while the sparse Bonsai tables $\mathsf{layered}_{\mathrm{S}}$ and $\mathsf{cleary}_{\mathrm{S}}$ are the slowest options. Between them are cht_{++}, $\mathsf{cleary}_{\mathrm{P}}$ and grp. Considering the query time for large instances, the difference to the construction time is that cht and grp are here slower than all variants of cleary and layered, where again $\mathsf{layered}_{\mathrm{P}}$ is the fastest option.

Unsuccessful Search. Figure 2 shows times for unsuccessful searches (i.e., when the query is for a key not present in the table). $\mathsf{layered}_{\mathrm{P}}$ is again generally the fastest (though with somewhat less consistent performance). grp is faster than cht for unsuccessful searches. While grp spends additional time for finding the queried sub-bucket in a group-bucket, this pays off since the sub-buckets in grp are far smaller than the buckets in cht.

Space. grp, followed by cht_{++} and then by cht_{50}, has the lowest peak memory requirements during the construction. The main reason is that, unlike the other approaches, we do not need to store displacement information. The size of a group-bucket in grp approaches $b_{\max}$ much better than a bucket in cht, which helped grp to delay rehashings. $\mathsf{cleary}_{\mathrm{S}}$ beats on some instances cht_{++} (but not grp) while having significantly slower construction times than cht or grp. However, the relatively large memory reallocation during a rehashing prevents the memory requirement of $\mathsf{cleary}_{\mathrm{S}}$ to stay below cht_{++} for a longer time.

4.1.2 Spikes in Time and Memory

The Bonsai tables have clear spikes in their time and space-usage plots, which our hash tables do not have. To understand this phenomena, recall that the Bonsai tables use linear probing with a maximum load factor of 0.95. As the Bonsai tables approach fullness, insertion and query times deteriorate, which is the case just before a peak in the space usage plot (Figure 1, bottom left). The peak itself is the consequence of rehashing having been performed. Peaks are more pronounced for the non-sparse variants, which keep all buckets of both the old and new hash tables in memory during rehashing.

We observe that, while the query times for cleary degrade dramatically before rehashing, query times improve considerably immediately afterwards. This reflects the way in which cleary and layered deal with displacement information. Due to the highly set maximum load factor (0.95), with high probability elements with the same hash value become mixed, resulting long lists of consecutive elements. Given that we consult an element at the i-th

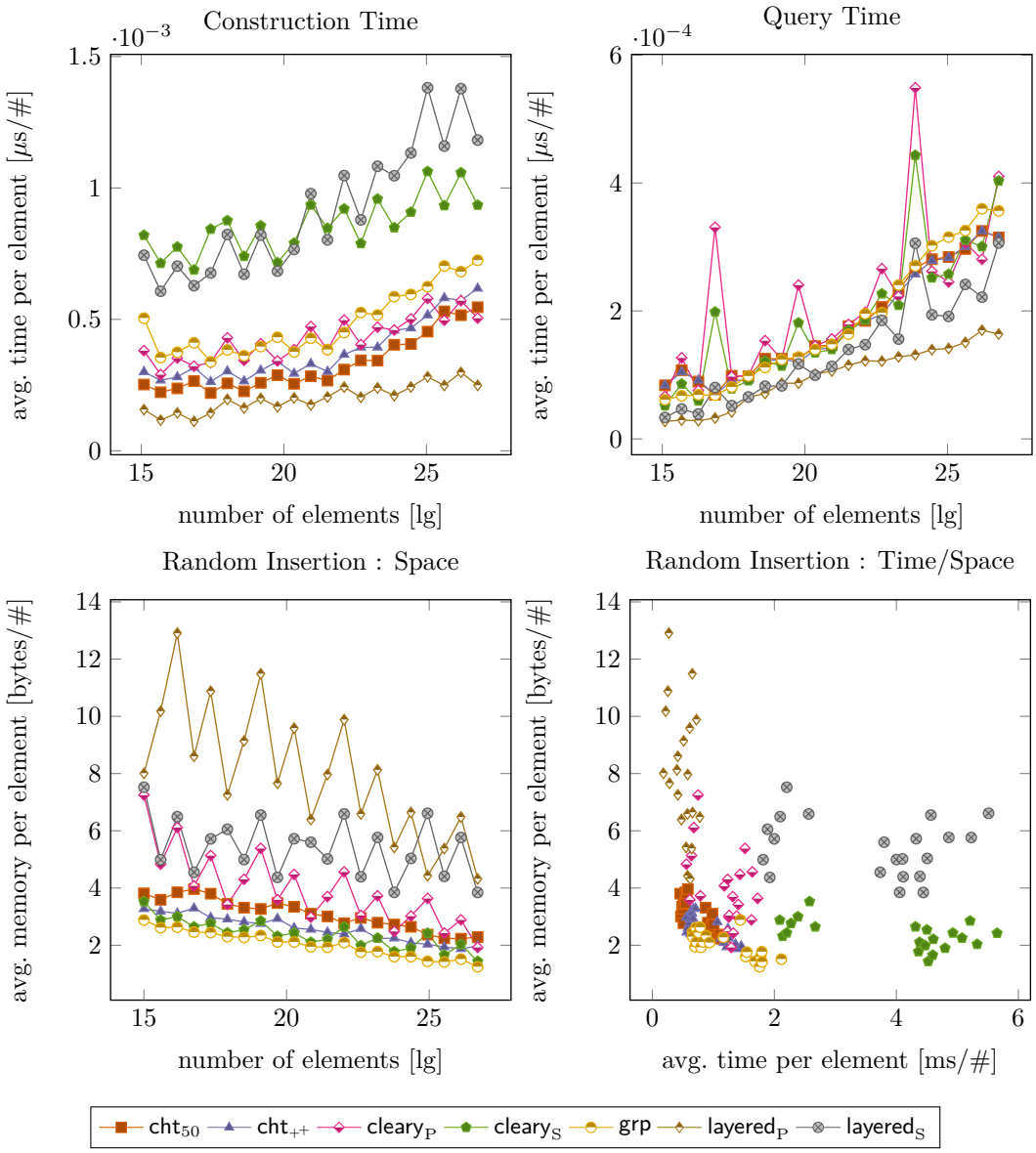


Figure 1 *Top Left*: Time for inserting $2^{10} \cdot (3/2)^n$ randomly generated 1-bit values and 32-bit keys into a compact hash table, for $n \geq 0$ (cf. Section 4.1). *Top Right:* Time for querying all inserted elements. *Bottom Left*: Peak memory needed during construction. *Bottom Right*: Memory and time per stored element.

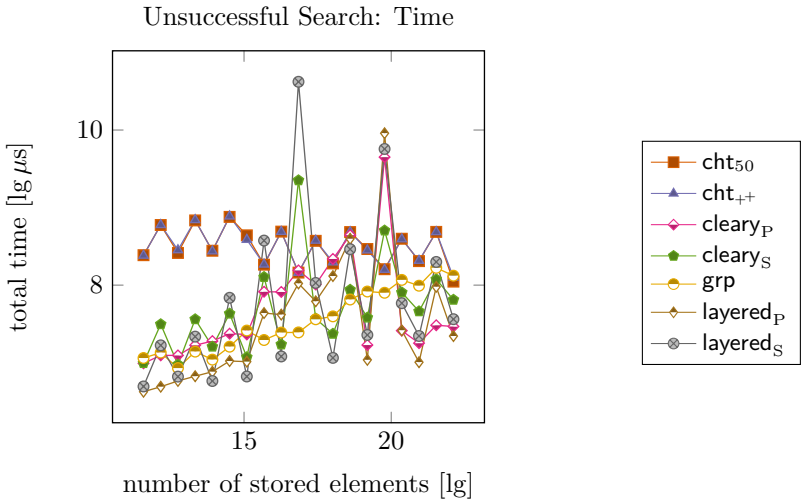


Figure 2 Time for looking up 2^{10} random keys that are not present in the hash tables.

position in the hash table at which such a long list of elements is stored, layered and cleary have to consult the displacement information of i. While layered stores this information in two separate data structures, cleary may have to scan all consecutive elements to the left of i. When inserting an element at the i-th position, linear probing scans to the right end of this list, requiring Bonsai tables to lookup displacements of all visited elements. Our new hash tables cht and grp have smoother performance because of the way they handle rehashing: the contents of the i-th bucket is moved to the $2i$-th and $(2i+1)$-th bucket after which the original bucket is freed.

In summary, layered_P is consistently the fastest compact hash table in our experiments for both insert and lookup queries and is also the most space consuming. cleary_P, layered_S, cleary_S offer different trade-offs, being either faster at insertions, lookups, or using less space. cht and grp have the lowest space requirements, but relatively high query times. The maximum bucket size $b_{\max}$ gives us a dial for trading speed for space-usage with cht and grp.

4.2 Non-Compact Hash Tables

Figure 3 and Figure 4 show an additional comparison with 8-bit values and two highly-optimized non-compact hash tables, namely:[8] Google's sparse hash table[4] google and Tessil's sparse map[9] tsl. Both google and tsl are *sparse*, resolve collisions with quadratic probing, use the SplitMix hash function [21], and had maximum load factor set to 0.95.

4.2.1 Insertions and Lookups

In Figure 3, we conducted the same experiment as in Section 4.1.1 on the non-compact hash tables. While our implementations are slower for queries (grp is sometimes almost three times slower than tsl), they consistently use half of the memory, sometimes even less. cht_{50} also shades google during construction.

[8] Unfortunately, we could not evaluate other hash tables mentioned in the introduction. The implementation of [10] lacks resize capabilities (`http://algo2.iti.kit.edu/sanders/programs/cuckoo/`), and the implementation of [12] has a bug (`https://github.com/TooBiased/DySECT/issues/2`).

[9] `https://github.com/Tessil/sparse-map`

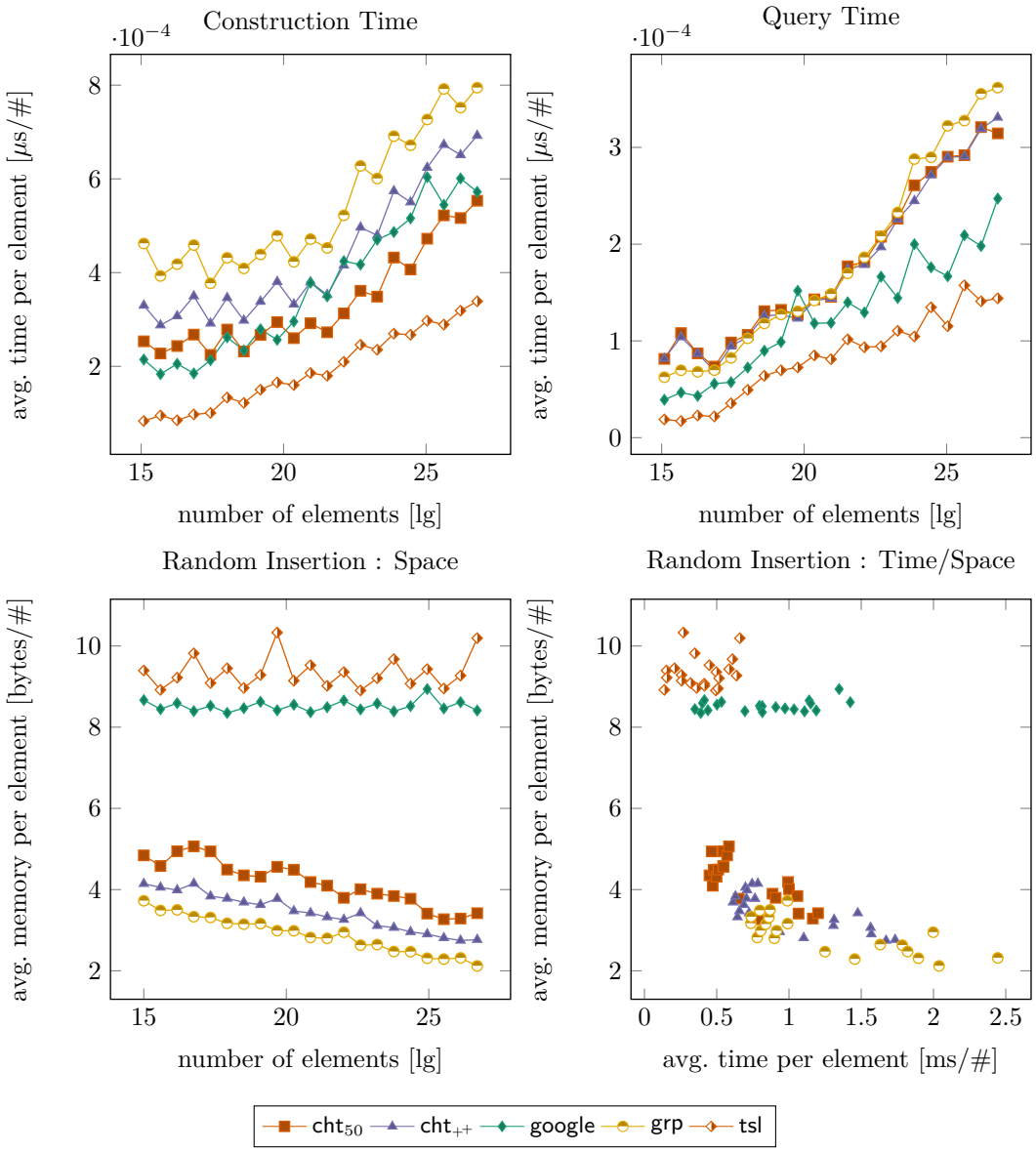


Figure 3 *Top Left*: Time for inserting $2^{10} \cdot (3/2)^n$ randomly generated 8-bit values and 32-bit keys into a not-necessarily compact hash table, for $n \geq 0$ (cf. Section 4.2). *Top Right:* Time for querying all inserted elements. *Bottom Left*: Peak memory needed during construction. *Bottom Right*: Memory and time per stored element.

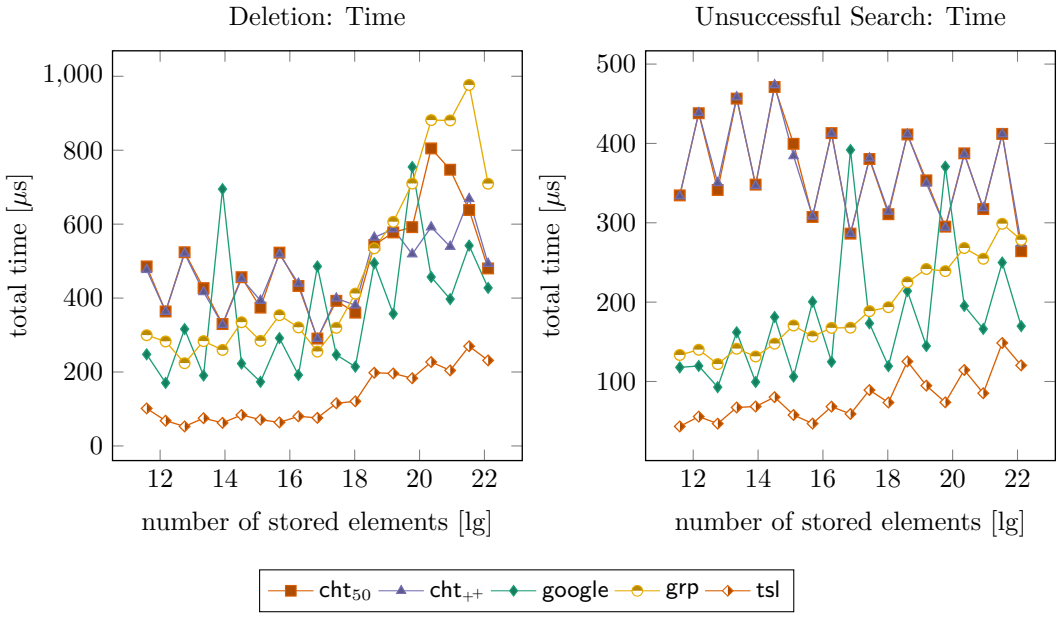


Figure 4 *Left*: Time for erasing 2^{10} random keys that are present in the hash tables. *Right*: Time for looking up 2^{10} random keys that are not present in the hash tables. In both figures, the number of elements (x-axis) is the number of elements a hash table contains (cf. Section 4.2.2).

4.2.2 Removing Elements

Figure 4 left shows the time to remove 2^{10} random elements. We used the hash tables created during the construction benchmark (Figure 3) with 8-bit values. Bonsai tables are not included because their current implementations do not support element removal. Our hash tables are again consistently slower than tsl, while times for google fluctuate above and below ours. grp becomes slower than cht on large instances. Experiments for unsuccessful searches (Figure 4 right) show a similar pattern.

5 Conclusions and Future Work

We have suggested a simple approach for implementing compact hash tables, and our implementations show positive results. The experiments reveal our hash tables grp and cht use the least space of all tested approaches. Moreover, their time and space requirements scale smoothly with the problem size, unlike the other compact (i.e., Bonsai) tables tested, whose performance is periodically adversely affected by rehashing. The new tables are also faster to construct than other hash tables with similar memory requirements – only hash tables with much higher memory requirements have faster construction times.

The main weakness of grp and cht is the slower lookup time, both for successful and unsuccessful searches. This is the price we pay for low space usage, which is achieved by keeping all buckets of cht and grp as close to $b_{\max}$ as possible, resulting in long scan times.

There are numerous avenues for future work. An analytical treatment of the space usage of the implemented hash table (whose rebuilding is triggered by the parameter $b_{\max}$), and the expected frequency of rehashes, as well as a better understanding of the use of overflow hash tables, would be welcome. In the experiments, the measured memory is the number of allocated bytes. The resident set sizes of our hash tables differ significantly to this quantity,

as we allocate many tiny fragments of space. A dedicated memory manager can reduce this space overhead and may also reduce the memory requirement of the bucket pointers by allocating a large contiguous array, pointers into which may require 32 bits, or less.

The AVX2 SIMD instruction set provides a major performance boost over earlier instruction sets like SSE – with benchmarks for comparing strings[10] indicating a speed boost of more than 50% for long strings. We wonder whether we can gain an even steeper acceleration in our hash tables when working with the newer AVX256 instruction set.

References

1 Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Proc. FOCS*, pages 787–796, 2010.

2 Daniel K. Blandford and Guy E. Blelloch. Compact representations of ordered sets. In *Proc. SODA*, pages 11–19, 2004.

3 Daniel K. Blandford and Guy E. Blelloch. Compact dictionaries for variable-length keys and data with applications. *ACM Trans. Algorithms*, 4(2):17:1–17:25, 2008.

4 Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.

5 Paolo Carlini, Phil Edwards, Doug Gregor, Benjamin Kosnik, Dhruv Matani, Jason Merrill, Mark Mitchell, Nathan Myers, Felix Natter, Stefan Olsson, Silvius Rus, Johannes Singler, Ami Tavory, and Jonathan Wakely. *The GNU C++ Library Manual*. FSF, 2018.

6 John G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Trans. Computers*, 33(9):828–834, 1984.

7 John J. Darragh, John G. Cleary, and Ian H. Witten. Bonsai: a compact representation of trees. *Softw., Pract. Exper.*, 23(3):277–291, 1993.

8 Erik D. Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Patrascu. De dictionariis dynamicis pauco spatio utentibus (*lat.* on dynamic dictionaries using little space). In *Proc. LATIN*, volume 3887 of *LNCS*, pages 349–361, 2006.

9 Johannes Fischer and Dominik Köppl. Practical evaluation of Lempel-Ziv-78 and Lempel-Ziv-Welch tries. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 191–207, 2017.

10 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.

11 Shunsuke Kanda, Dominik Köppl, Yasuo Tabei, Kazuhiro Morita, and Masao Fuketa. Dynamic path-decomposed tries. *arXiv 1906.06015*, 2019.

12 Tobias Maier, Peter Sanders, and Stefan Walzer. Dynamic space efficient hashing. *Algorithmica*, 81(8):3162–3185, 2019.

13 Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

14 Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

15 Andreas Poyias, Simon J. Puglisi, and Rajeev Raman. Compact dynamic rewritable (CDRW) arrays. In *Proc. ALENEX*, pages 109–119, 2017.

16 Andreas Poyias, Simon J. Puglisi, and Rajeev Raman. m-Bonsai: A practical compact dynamic trie. *Int. J. Found. Comput. Sci.*, 29(8):1257–1278, 2018.

17 Martin Raab and Angelika Steger. "balls into bins" - A simple and tight analysis. In *Proc. RANDOM*, volume 1518 of *LNCS*, pages 159–170, 1998.

18 Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proc. ICALP*, volume 2719 of *LNCS*, pages 357–368, 2003.

[10] `https://github.com/koeppl/packed_string`

19 John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. SIGMETRICS*, pages 134–142, 1990.

20 Kenneth A. Ross. Efficient hash probes on modern processors. In *Proc. ICDE*, pages 1297–1301, 2007.

21 Guy L. Steele Jr., Doug Lea, and Christine H. Flood. Fast splittable pseudorandom number generators. In *Proc. OOPSLA*, pages 453–472, 2014.

Effect of Initial Assignment on Local Search Performance for Max Sat

Daniel Berend
Departments of Mathematics and of Computer Science,
Ben-Gurion University, Beer Sheva 84105, Israel
berend@cs.bgu.ac.il

Yochai Twitto
Department of Computer Science, Ben-Gurion University, Beer Sheva 84105, Israel
twittoy@cs.bgu.ac.il

Abstract

In this paper, we explore the correlation between the quality of initial assignments provided to local search heuristics and that of the corresponding final assignments. We restrict our attention to the Max r-Sat problem and to one of the leading local search heuristics – Configuration Checking Local Search (CCLS). We use a tailored version of the Method of Conditional Expectations (MOCE) to generate initial assignments of diverse quality.

We show that the correlation in question is significant and long-lasting. Namely, even when we delve deeper into the local search, we are still in the shadow of the initial assignment. Thus, under practical time constraints, the quality of the initial assignment is crucial to the performance of local search heuristics.

To demonstrate our point, we improve CCLS by combining it with MOCE. Instead of starting CCLS from random initial assignments, we start it from excellent initial assignments, provided by MOCE. Indeed, it turns out that this kind of initialization provides a significant improvement of this state-of-the-art solver. This improvement becomes more and more significant as the instance grows.

2012 ACM Subject Classification Theory of computation → Theory of randomized search heuristics; Theory of computation → Stochastic approximation

Keywords and phrases Combinatorial Optimization, Maximum Satisfiability, Local Search, Probabilistic Algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.8

Funding This research was partially supported by the Milken Families Foundation Chair in Mathematics and by the Lynne and William Frankel Center for Computer Science.

Acknowledgements We thank Shaowei Cai for providing us access to the original authors' implementation of the CCLS solver used in Max Sat Evaluation 2016, and André Abramé for providing us access to the Abramé-Habet benchmark used (partially) in that evaluation. We also thank Gregory Gutin and Shahar Golan for their helpful comments on this paper.

1 Introduction

In the Maximum Satisfiability (Max Sat) problem [31], we are given a sequence of clauses over some boolean variables. Each clause is a disjunction of literals over different variables. A literal is either a variable or its negation. We seek a truth (`true`/`false`) assignment for the variables, maximizing the number of satisfied (made `true`) clauses.

In the Max r-Sat problem, each clause is restricted to consist of at most r literals. Here we restrict our attention to instances with clauses consisting of exactly r literals each (sometimes called Max Er-Sat). We denote by n the number of variables and by m the number of clauses. The density of the instance is $\alpha = m/n$. As is customary in the literature, we focus on the case where r and α are constant.

As Max r-Sat (for $r \geq 2$) is NP-hard [5], it cannot be exactly solved in polynomial time (unless $P = NP$), and one must resort to approximation algorithms and heuristics. Numerous methods have been suggested for solving Max r-Sat, e.g. [20, 44, 43, 32, 14, 37, 3, 19, 29], and an annual competition of solvers has been held since 2006 [4]. Satisfiability related questions attracted a lot of attention from the scientific community. As an example, one may consider the well-studied satisfiability threshold question for random instances [15, 25, 2, 35, 16, 21]. For a comprehensive overview of the whole domain of satisfiability we refer to [7, 22].

1.1 Local search

Local search heuristics [30] explore the assignment space. They usually start from a randomly generated assignment, and traverse the search space by flipping variables, usually one at a time. The leading solver Configuration Checking Local Search (CCLS) [32] follows this scheme and flips variables until some predefined number of flips is executed or the allotted time has been used up. Of course, if a satisfying assignment has been found, the execution is stopped as well.

CCLS performs two types of flips: random ones, with some predefined probability p, and greedy ones, with probability $1 - p$. Random flips just flip a randomly selected variable from a randomly selected unsatisfied clause. Greedy flips are ones that flip the seemingly best possible variable among all the variables whose configuration has been changed [32] and who satisfy at least one currently unsatisfied clause. This variable is the one with the maximum score out of those variables, i.e., the one whose flipping will lead to the maximum number of satisfied clauses. Ties are broken randomly.

Recent works, related to local search, configuration checking, CCLS, and algorithms of the same spirit, include [38, 11, 10, 33, 34, 1, 8, 9, 12, 42, 36, 45, 13, 24].

1.2 The Method of Conditional Expectations

The simple randomized approximation algorithm, which assigns to each variable a uniformly random truth value, independently of all other variables, satisfies $1-1/2^r$ of all clauses on the average. Furthermore, this simple algorithm can be easily derandomized using the Method of Conditional Expectations (MOCE) [23, 47], yielding an assignment that is guaranteed to satisfy at least this proportion of clauses.

In a sense, this method is optimal for Max 3-Sat, as no polynomial-time algorithm for Max 3-Sat can achieve a performance ratio exceeding $7/8$ unless $P = NP$ [28]. We note that, typically, this method yields assignments that are much better than this worst-case bound.

MOCE iteratively constructs an assignment by going over the variables in some (arbitrary) order. At each iteration, it sets the seemingly better truth value to the currently considered variable. This is done by comparing the expected number of satisfied clauses under each of the two possible truth values it may set to the current variable.

For a given truth value, the expected number of satisfied clauses is the sum of three quantities. The first is the number of clauses already satisfied by the values assigned to the previously considered variables. The second is the additional number of clauses satisfied by the assignment of the given truth value to the current variable. The third is the expected number of clauses that will be satisfied by a random assignment to all currently unassigned variables. The truth value, for which the sum in question is larger, is the one selected for the current variable. Ties are broken arbitrarily or randomly. The whole process is repeated until all variables are assigned.

Recent theoretical and empirical works related to MOCE, and algorithms of the same spirit, include [17, 39, 41, 40, 18].

1.3 Overview

In Section 2, we explore the correlation between the quality of the initial assignments provided to local search heuristics and the quality of the final assignments resulting from them. We restrict our attention to CCLS, and use a tailored version of MOCE to generate initial assignments of diverse quality, to accommodate the exploration of the correlation.

We show that there is a strong long-lasting correlation between the quality of the initial assignment, from which the local search heuristic starts, and the quality of the final assignment provided by it. This implies that, even when we delve deeper into the local search, we are still in the shadow of the initial assignment. Thus, the quality of the initial assignment is crucial under practical time constraints. The observed correlation decays slower for denser instances, and faster for sparser ones. We show that the correlation is statistically significant, and estimate the impact of the improvement in the quality of the initial assignment on the quality of the final assignment.

In Section 3, we demonstrate our point by improving CCLS. Instead of starting CCLS from a random initial assignment, we start it from excellent initial assignments, provided by MOCE. This kind of initialization provides a significant improvement of this state-of-the-art solver. Moreover, the improvement becomes more and more significant as the instance grows. It has been noticed in other problems, such as TSP and QAP, that local search heuristics yield excellent results when started from initial solutions selected greedily with respect to expectation [27, 26]. A summary and conclusions are presented in Section 4.

2 Correlation between the quality of initial and final assignments

In this section, we explore the correlation between the number of clauses unsatisfied by an initial assignment and the number of those unsatisfied by the corresponding final assignment, where the transition is by CCLS. We explore the ongoing correlation during the execution as well. We have chosen CCLS for its excellent performance; a local search heuristic of lower quality may well be expected to yield an even stronger correlation.

To generate initial assignments of diverse quality, we manipulate MOCE by adding to it a parameter that allows us to invert its decision regarding the truth value for the current variable. This parameter, to which we refer as the inversion probability, is the probability to assign to a variable the truth value opposite to the one chosen by MOCE. Namely, for a given inversion probability $0 \leq p \leq 1$, at each step, we assign to the current variable the truth value chosen by MOCE with probability $1 - p$, and the opposite truth value with probability p. Thus, for $p = 0$ the algorithm is simply MOCE, while for $p = 1$ it is "anti-MOCE". We refer to this tailored algorithm as PMOCE.

We have generated a benchmark, consisting of 5 families of instances of Max 3-Sat. Each of the families consists of 150 instances over 100,000 variables. The densities of the 5 families are 5, 7, 9, 12, 15. The instances in each family were generated uniformly at random as follows. The clauses of an instance were generated independently of each other. Each of the clauses was generated by selecting 3 distinct variables uniformly at random, and then negating each of them with probability 1/2, independently.

2.1 End-to-end correlation

In the following, we describe what we have done in the experiment for each family. For each instance in the family, we executed PMOCE with 51 inversion probabilities, ranging from 0 to 1 in steps of 0.02. Thus, we obtained 51 initial assignments with presumed diverse

Table 1 End-to-end correlation coefficients and regression slopes.

	correlation coefficient			regression slope	
density	**mean**	**std**	***p*-value**	**mean**	**std**
5	0.52	0.11	$1.7 \cdot 10^{-3}$	$0.5 \cdot 10^{-3}$	$0.1 \cdot 10^{-3}$
7	0.74	0.06	$3.6 \cdot 10^{-7}$	$1.5 \cdot 10^{-3}$	$0.2 \cdot 10^{-3}$
9	0.79	0.12	$2.1 \cdot 10^{-3}$	$2.2 \cdot 10^{-3}$	$0.5 \cdot 10^{-3}$
12	0.73	0.17	$1.2 \cdot 10^{-3}$	$2.4 \cdot 10^{-3}$	$1.0 \cdot 10^{-3}$
15	0.83	0.08	$1.1 \cdot 10^{-5}$	$3.4 \cdot 10^{-3}$	$0.7 \cdot 10^{-3}$

quality. From each of these initial assignments, we started a local search using CCLS, and thus obtained 51 final assignments. By the end of the 51 executions, we had 51 pairs of numbers. Each pair consisted of the number of clauses unsatisfied by the initial assignment generated by PMOCE, and the number of unsatisfied clauses at the end of the search done by CCLS. The cutoff time of CCLS was set to 30 minutes, measured in CPU time.

For each instance, we calculated the correlation coefficient over the corresponding 51 pairs. After going over the whole family, we had 150 correlation coefficients – one for each instance. Then, we calculated the mean and standard deviation of these 150 values of correlation coefficients.

For each of the correlation coefficients, we also calculated the p-value. The p-value is the probability that we would have found this correlation, or a higher one, if the correlation coefficient was in fact zero (null hypothesis). If this probability is lower than the conventional 5% (i.e., the p-value is less than 0.05), the correlation coefficient is considered statistically significant. For each family, we calculated the average p-value over the 150 correlation coefficients as a measure of the statistical significance of the results.

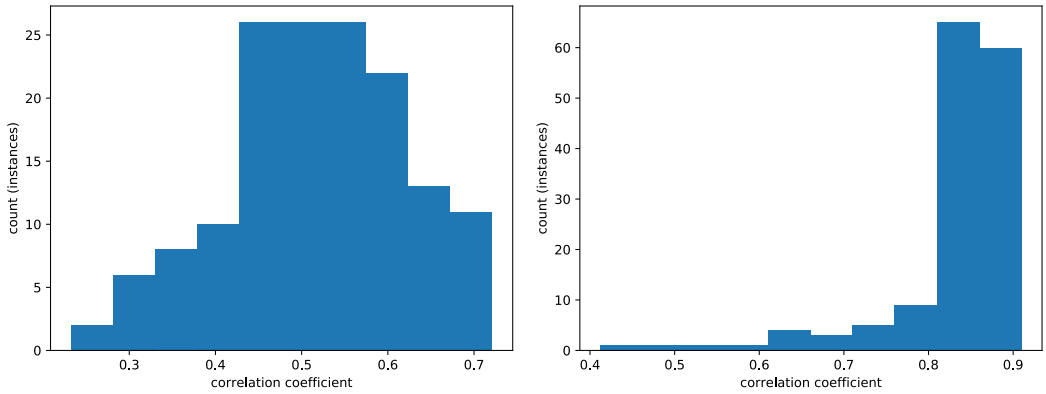
To measure the impact of the improvement of the quality of an initial assignment on the quality of the corresponding final assignment, we applied regression analysis. Specifically, we calculated the regression line of each of the instances of a family, and took its slope as a measure of the strength of the impact. We took the average of these 150 slopes as a measure of the strength of this impact in a given family.

The results are provided in Table 1. Each line summarizes the results of one family. For example, the first line summarizes the results of the family with density 5. In this family, instances are of 100,000 variables and 500,000 clauses. The mean correlation coefficient measured (over 150 random instances) was 0.52, with a standard deviation of 0.11. The mean p-value was $1.7 \cdot 10^{-3}$, and the mean and standard deviation of the regression slope were $0.5 \cdot 10^{-3}$ and $0.1 \cdot 10^{-3}$, respectively.

Figure 1 depicts histograms of the 150 end-to-end correlation coefficients of the family of density 5 (Figure 1a) and for the family of density 15 (Figure 1b).

The results show a strong positive correlation between the quality of the initial and final assignment for all densities. The correlation is stronger for denser families. The p-value is lower by far than the conventional 0.05, which indicates that the correlation coefficients obtained in the experiments are statistically very significant.

While the correlation is strong, the regression slope suggests that a large improvement in the initial assignment yields only a small improvement in the final assignment. As CCLS eventually converges to the optimal solution, there is little room for improvement by the end of its execution, so that this regression slope makes sense. Moreover, it is to be expected that the slope becomes even smaller as one runs CCLS longer.



(a) Family of density 5. **(b)** Family of density 15.

Figure 1 Histograms of end-to-end correlation coefficients.

Note that, after 30 minutes of execution, CCLS is way beyond its rapid improvement stage. In fact, it is deep in its convergence stage and shows relatively minor improvements as time goes by. This validates the correlation observed as meaningful.

Figure 2 depicts the number of unsatisfied clauses as a function of the number of flips made, for an arbitrary (but representative) instance from the family of density 15. The graph shows this number for inversion probability of 0 (MOCE) and 1 (anti-MOCE). We see that CCLS enters its convergence stage quite early in the execution.

We also emphasize the two phases seen in the graphs. The first phase is the rapid improvements phase. In this phase, the number of unsatisfied clauses is decreasing rapidly. This phase ends after about 100,000 flips. The second phase, which we call the convergence phase, continues from there onward. In this phase, the improvements are rarer and smaller.

2.2 Ongoing correlation

Besides the end-to-end correlation, we explored the ongoing correlation during the experiment. To this end, for each initial assignment, we recorded the minimum number of unsatisfied clauses found so far, not only at the end of the execution, but also after every 1000 flips made by CCLS. Then we calculated the correlation coefficient between the number of clauses unsatisfied by the initial assignment and the number of unsatisfied clauses recorded at each 1000 flips snapshot.

The number of flips made during the execution is very different for different families. In a denser instance, a flip takes longer, so that less flips are made. Even for instances of the same family, the number of flips varies. We provide statistics only up to the minimal number of flips made, over all instances in the family.

Figure 3 depicts the decay in the correlation as a function of time, where time is measured in number of flips made from the beginning of the local search. It seems that the number of flips is the natural time scale to measure the correlation decay. While the graphs are noisy, the trend is clear – the correlation gradually decays as a function of the number of flips made, and it does so slower for denser families. Moreover, as the density grows larger, the differences in the decay seem to be smaller and the graphs are almost overlapping.

Figure 3a shows the full results. It provides the graphs of correlation decay of all the families. In the figure, one may observe that the number of flips made in denser families is much smaller than the number of flips made in sparser ones. For example, the minimal

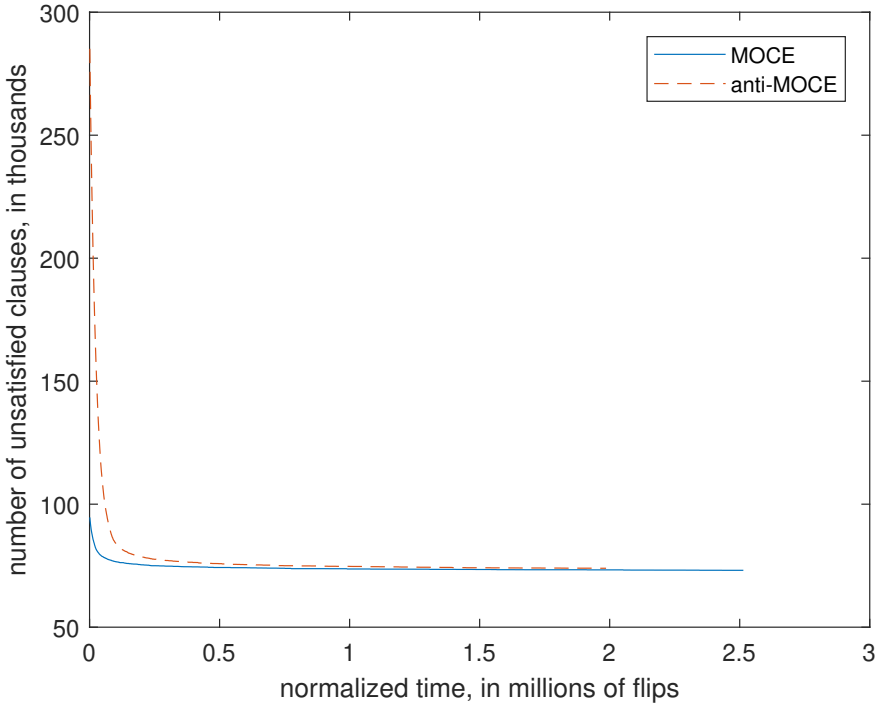


Figure 2 Number of unsatisfied clauses as function of the number of flips.

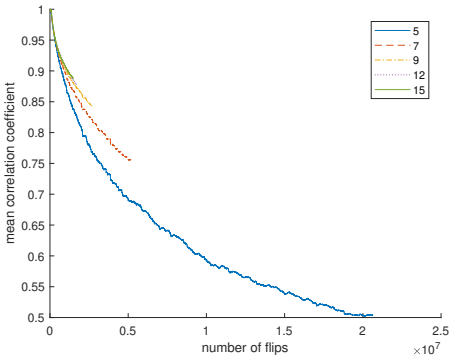
number of flips over all instances and inversion probabilities for the family of density 5 was about 20,600,000, while for the family of density 15 it was about 1,500,000. The reason is that, in denser families, each variable appears in a larger number of clauses, and a flip makes a larger number of variables available for selection subsequently. Thus, at each step, CCLS has to deal with a larger pool of candidates for flipping, which consumes more time per flip.

Figure 3b depicts the same graphs, but only up to about 1,500,000 flips, which is the place where the graph of the family of density 15 ends. In this figure, we see clearly the faster decay of the correlation in sparser families, as well as the smaller differences between the decay in denser families.

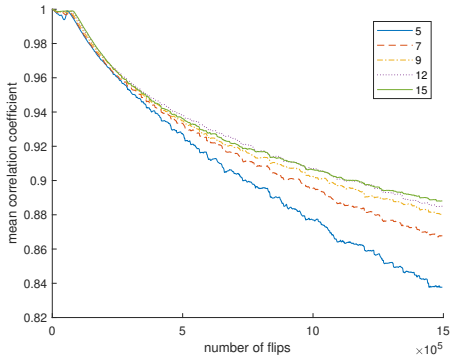
Figure 3c zooms in on the first 150,000 flips. During this stage, we observe a phenomenon of phase transition in the decay of the correlation. The empirical results suggest two phases of decay. The first phase starts at the beginning and ends after about 60,000-80,000 flips. In this phase, the correlation decays very slowly. This phase is characterized by a rapid decrease in the number of unsatisfied clauses, and is aligned with the rapid decrease shown in Figure 2.

The second phase is from about 60,000-80,000 flips onward. This phase is characterized by a faster decay in the correlation. It is aligned with the convergence stage of CCLS, shown in Figure 2, in which the number of unsatisfied clauses is decreasing slowly over time.

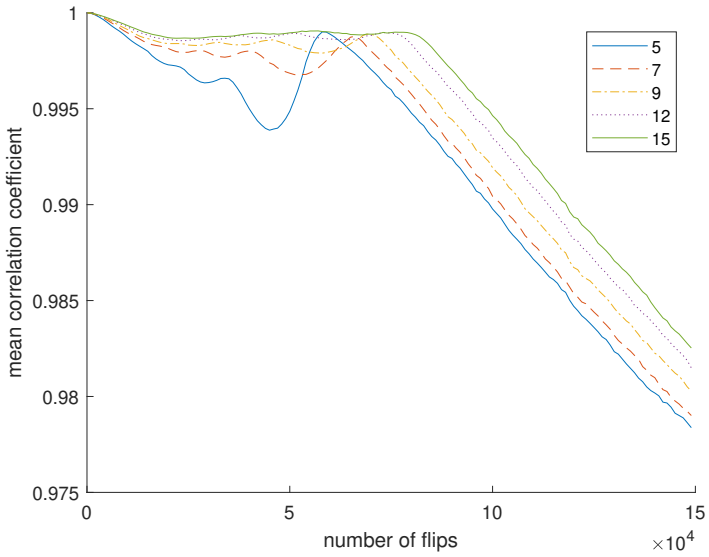
The position of the phase transition around 60,000-80,000 flips may be explained by the fact that the initial assignment provided by MOCE is expected to be at a distance of about 50,000 flips from an optimal solution. So the first 50,000 flips are significant. But, as about 30% of the flips of CCLS are random, and not all the flips are useful in general, this area stretches further to about 60,000-80,000 flips.



(a) Full graphs.



(b) Up to 1,500,000 flips.



(c) First 150,000 flips.

Figure 3 Ongoing correlation decay as a function of the number of flips.

During the first phase, the initial assignment is very important in determining the correlation. In all the executions, the decrease in the number of satisfied clauses is rapid and considerable at this phase. Thus, the different executions maintain their relative positions, which leads to a very slow decrease in the correlation. Afterward, the correlation decays at about the same speed, as can be seen in Figure 3c.

2.3 Experimentation information

The experiments described in this section were executed on a Sun Grid Engine (SGE) [46] managed cluster of 31 identical IBM m4 servers with Intel Xeon E5-2620@2.0GHz processors. Each of the servers consists of 24 computation cores and 64GB of working memory. Thus, we had 744 computation cores and 1984GB of working memory at hand.

We limited each of the jobs submitted to the cluster to use up to 3GB of working memory. Provided the load on the cluster, we managed to achieve a parallelization of about 300 times, thus reducing the experiments overall sequential time of approximately 2.18 years to around 2.66 days of parallel execution.

3 Improving CCLS

In this section, we study the improvement obtained by letting CCLS start its execution from good initial assignments, versus starting it from a random assignment (as done originally). Specifically, the good initial assignments we use are assignments provided by MOCE. We refer to the algorithm that starts from the assignment provided by MOCE as MOCE-CCLS. To emphasize the fact that the original CCLS algorithm starts from a random assignment, we will call it RAND-CCLS.

We first conducted experiments on several families of random instances. The families have been selected in a systematic way, so as to reveal trends in the performance, and connect it to the parameters of the family. Afterward, we conducted experiments on some public benchmarks. We show that MOCE-CCLS scales much better than RAND-CCLS. In particular, as the instance size grows, so does the performance improvement provided by MOCE-CCLS over RAND-CCLS.

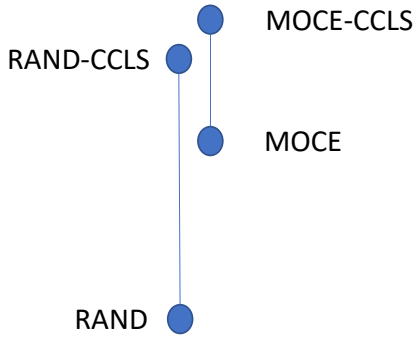


Figure 4 Performance diagram. Higher is better.

In Figure 4, we summarize qualitatively what we have observed in the experiments. The higher the algorithm appears in the diagram, the better it is. Inspecting the diagram, one can see that MOCE-CCLS performs much better than MOCE, which in turn shows performance very far away from the baseline reference RAND. MOCE-CCLS performs better than RAND-CCLS as well. The last statement holds significantly for large instances, while for small and medium instances MOCE-CCLS maintains or slightly improves the performance of RAND-CCLS.

3.1 Comparative performance on structured benchmarks

In this section we focus on random instances, for which the clauses are of length 3, the number of variables ranges from 10,000 to 1,000,000, and the density from 3 to 9. Such ranges allow us to systematically study the performance of the algorithms at hand on diverse families. For each family, we selected 100 instances uniformly at random, in the same way elaborated in Section 2.

For Max r-Sat, the reference baseline RAND unsatisfies $m/2^r$ clauses on average, with an approximate standard deviation of $\sqrt{m(1-1/2^r)/2^r}$ clauses [6]. For convenience, the (theoretical) average number of clauses unsatisfied by RAND for the families we studied (namely, $m/8$), is provided in Table 2. The rows correspond to the various numbers of variables, n, and the columns to the various densities, α.

Table 2 The number of clauses unsatisfied by RAND and MOCE.

n \ α	3		5		7		9	
	RAND	MOCE	RAND	MOCE	RAND	MOCE	RAND	MOCE
10000	3750	412	6250	1500	8750	2889	11250	4442
50000	18750	2078	31250	7459	43750	14409	56250	22209
100000	37500	4149	62500	14944	87500	28861	112500	44436
500000	187500	20790	312500	74702	437500	144296	562500	222074
1000000	375000	41559	625000	149383	875000	288572	1125000	444174
% unsat	12.5%	1.4%	12.5%	3%	12.5%	4.1%	12.5%	4.9%

Table 3 The improvement by executing CCLS after RAND and MOCE.

n \ α	3		5		7		9	
	(R−RC)/R	(M−MC)/M	(R−RC)/R	(M−MC)/M	(R−RC)/R	(M−MC)/M	(R−RC)/R	(M−MC)/M
10000	100.0%	100.0%	96.0%	83.6%	85.5%	56.2%	77.4%	42.9%
50000	100.0%	100.0%	95.5%	81.2%	84.8%	54.1%	76.7%	41.2%
100000	100.0%	100.0%	95.1%	79.9%	84.3%	52.9%	76.2%	40.2%
500000	100.0%	100.0%	90.4%	69.4%	77.1%	42.4%	68.3%	31.3%
1000000	80.6%	100.0%	48.7%	49.6%	37.4%	27.1%	31.6%	18.3%

Table 2 also presents the average number of clauses unsatisfied by MOCE. It turns out that this number scales linearly with the number of clauses, and thus can be described as a proportion of the number of clauses, for any fixed density. The proportion of clauses unsatisfied by MOCE, out of all clauses, was 1.4%, 3%, 4.1%, and 4.9% for the densities 3, 5, 7, and 9, respectively. For each family, the percentage of clauses unsatisfied by each of the algorithms is provided in the last line of the table.

MOCE is a linear time algorithm, and is extremely fast in practice. In fact, its execution time is but a few seconds for the larger instances we studied, and less than a second for the small and medium size instances.

Although MOCE returns excellent solutions, it benefits a lot from supplementing it with a highly performing local search. In fact, executing the local search part of CCLS (which we simply call CCLS), starting from the solution returned by MOCE, we obtained a significant improvement. Namely, the number of unsatisfied clauses is significantly reduced at the local search stage.

This improvement is summarized in Table 3. In this table, the columns shortly named "(M−MC)/M" present the relative improvement of MOCE-CCLS over MOCE. This relative improvement is the difference between the number of clauses unsatisfied by MOCE and the number of those unsatisfied by MOCE-CCLS, divided by the number of clauses unsatisfied by MOCE. In the table, we also present the improvement of supplementing RAND with CCLS (which is simply the standard version of CCLS), under the columns named "(R−RC)/R".

It is worth mentioning that this significant improvement comes with a caveat – a significant increase in the execution time. In fact, the results shown in Table 3 are based on 30 minutes executions of CCLS, after the initial solution (by either RAND or MOCE) has been obtained in just a few seconds. One more caveat is due to the fact that, as the instances grow larger, this improvement decreases. As the instance grows larger, the number of flips CCLS can perform during the allotted time decreases, and with it decreases the obtained improvement as well.

Table 4 MOCE-CCLS vs. RAND-CCLS.

n \ α	3			5		
	RC	MC	% improve	RC	MC	% improve
10000	0	0	NaN	248	246	0.81%
50000	0	0	NaN	1417	1403	0.99%
100000	0	0	NaN	3038	3002	1.18%
500000	0	0	NaN	29976	22894	23.63%
1000000	72642	0	100.00%	320674	75260	76.53%
n \ α	7			9		
	RC	MC	% improve	RC	MC	% improve
10000	1265	1264	0.08%	2546	2537	0.35%
50000	6647	6617	0.45%	13122	13052	0.53%
100000	13717	13588	0.94%	26770	26554	0.81%
500000	99976	83163	16.82%	178234	152512	14.43%
1000000	548044	210440	61.60%	769640	363037	52.83%

We conclude this section by comparing MOCE-CCLS and RAND-CCLS head to head. The comparison is provided in Table 4 (which is wrapped for readability). For each density, we provide the number of clauses unsatisfied by RAND-CCLS, the number of clauses unsatisfied by MOCE-CCLS, and the relative improvement of MOCE-CCLS over RAND-CCLS. The latter number is the difference between the number of clauses unsatisfied by RAND-CCLS and the number of those unsatisfied by MOCE-CCLS, divided by the number of clauses unsatisfied by RAND-CCLS.

The results demonstrate our point regarding the importance of the initial solution. Even after 30 minutes of local search, and using the excellent local search heuristics CCLS, the initialization with MOCE instead of RAND yields better solutions. Moreover, MOCE-CCLS proved to be much more scalable than RAND-CCLS. As the instance grows larger, the improvement of MOCE-CCLS over RAND-CCLS becomes more significant. Whereas, for small instances, MOCE-CCLS improves RAND-CCLS by less than 1%, for large instances the improvement exceeds 50%.

In view of the above, when using a local search algorithm for Max Sat, one should strive to start the search from very good assignments. This holds as long as it is not too much time consuming to attain such assignments.

3.1.1 Experimentation information

The experiments described in this section were carried out on the same infrastructure as in Section 2.3. Here as well, we limited each of the jobs submitted to the cluster to use up to 3GB of working memory. Provided the load on the cluster, we managed to achieve a parallelization of about 100 times, thus reducing the experiment overall sequential time of approximately 4.11 months to around 1.25 days of parallel execution.

3.2 Comparative performance on public benchmarks

The random instances of Maximum Satisfiability Evaluation 2016 were tailored mainly for complete solvers. Thus, they are very small and less adequate for evaluation of local search heuristics. Indeed, most of the solvers participating in that evaluation found solutions with the same number of unsatisfied clauses most of the time; the ranking was only according to the time they consumed to reach their best solutions. In our comparison of RAND-CCLS and MOCE-CCLS, the situation was no different.

In the following, we consider three additional benchmarks in the same spirit as the 2016 Evaluation – but larger ones. As we wanted to keep the exact same blend of instances, we created the new benchmarks by blowing up the original ones. We enlarged the number of variables and that of clauses in each instance, while keeping the density the same as in the evaluation. We created three expanded benchmarks by enlarging the original one by factors of 10, 100, and 1000.

We compared MOCE-CCLS and RAND-CCLS on the enlarged instances using the Instance Won measure. This measure is the one used in the Max Sat Evaluation [4] held in 2016, from which we took the original instances. We ran each of the two competitors on each of the instances for a few minutes (CPU time). For each instance, the winner is the competitor that provides the smaller number of unsatisfied clauses. Ties are broken by the time it took each competitor to arrive at its best solution. The overall winner is the heuristic that wins more instances.

While RAND-CCLS wins on the competition instances, it is enough to blow up the instances tenfold to have MOCE-CCLS achieve an overall draw. When scaling the instances by a factor of 100, MOCE-CCLS wins decisively, and when scaling by a factor of 1000, it beats RAND-CCLS by a knockout. In fact, MOCE-CCLS wins on the expanded benchmarks in terms of the number of unsatisfied clauses, and not merely by time. Namely, MOCE-CCLS provides solutions with a strictly smaller number of unsatisfied clauses.

Finally, we note that MOCE alone is not enough. It is the value from the combined solver MOCE-CCLS that leads to the extra satisfied clauses. Moreover, CCLS provides a significant improvement to the excellent initial solutions of MOCE. Thus, the state-of-the-art performance of MOCE-CCLS is attributed to both its ingredients: MOCE and CCLS.

4 Summary and conclusions

In this paper, we have explored the correlation between the quality of initial assignments provided to local search heuristics and that of the corresponding final assignments. We have shown that this correlation is significant and long-lasting. Thus, under practical time constraints, the quality of the initial assignment is crucial to the performance of local search heuristics.

We demonstrated our point by improving the state-of-the-art solver CCLS, by combining it with MOCE. Instead of starting CCLS from random initial assignments, we started it from excellent initial assignments, provided by MOCE. The combined MOCE-CCLS solver provided a significant improvement over CCLS. Moreover, MOCE-CCLS proved to be much more scalable. Namely, it handles larger instances better, and shows superior performance on them.

Given the above, we recommend MOCE-CCLS over RAND-CCLS. Furthermore, we recommend starting CCLS from solutions even better than those provided by MOCE, as long as such may be obtained in linear time or slightly longer (say, by a logarithmic factor).

References

1 André Abramé, Djamal Habet, and Donia Toumi. Improving configuration checking for satisfiable random k-sat instances. *Annals of Mathematics and Artificial Intelligence*, 79(1-3):5–24, 2017.

2 Dimitris Achlioptas and Yuval Peres. The threshold for random k-sat is $2^k \log 2 - o(k)$. *Journal of the American Mathematical Society*, 17(4):947–973, 2004.

3 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Sat-based maxsat algorithms. *Artificial Intelligence*, 196:77–105, 2013.

4 Josep Argelich, Chu Min Li, Felip Manyà, and Jordi Planes. Maxsat evaluations. URL: `http://www.maxsat.udl.cat/`.

5 Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, 2 edition, 2003.

6 Daniel Berend and Yochai Twitto. The normalized autocorrelation length of random max r-sat converges in probability to $(1 - 1/2^r)/r$. In *The 19th International Conference on Theory and Applications of Satisfiability Testing (SAT 2016)*, pages 60–76. Springer, 2016.

7 Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of Satisfiability*, volume 185. IOS press, 2009.

8 Noureddine Bouhmala. A variable neighborhood walksat-based algorithm for max-sat problems. *The Scientific World Journal*, 2014, 2014.

9 Shaowei Cai, Zhong Jie, and Kaile Su. An effective variable selection heuristic in sls for weighted max-2-sat. *Journal of Heuristics*, 21(3):433–456, 2015.

10 Shaowei Cai, Chuan Luo, John Thornton, and Kaile Su. Tailoring local search for partial maxsat. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, page 2623–2629. AAAI Press, 2014.

11 Shaowei Cai and Kaile Su. Local search with configuration checking for sat. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pages 59–66. IEEE, 2011.

12 Shaowei Cai and Kaile Su. Local search for boolean satisfiability with configuration checking and subscore. *Artificial Intelligence*, 204:75–98, 2013.

13 Byungki Cha, Kazuo Iwama, Yahiko Kambayashi, and Shuichi Miyazaki. Local search algorithms for partial maxsat. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, page 263–268. AAAI Press, 1997.

14 Ruiwen Chen and Rahul Santhanam. Improved algorithms for sparse max-sat and max-k-csp. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT 2015)*, pages 33–45. Springer, 2015.

15 Vašek Chvátal and Bruce Reed. Mick gets some (the odds are on his side) [satisfiability]. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 620–627. IEEE, 1992.

16 Amin Coja-Oghlan. The asymptotic k-sat threshold. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 804–813. ACM, 2014.

17 Don Coppersmith, David Gamarnik, MohammadTaghi Hajiaghayi, and Gregory B Sorkin. Random max sat, random max cut, and their phase transitions. *Random Structures & Algorithms*, 24(4):502–545, 2004.

18 Kevin P Costello, Asaf Shapira, and Prasad Tetali. Randomized greedy: new variants of some classic approximation algorithms. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 647–655. Society for Industrial and Applied Mathematics, 2011.

19 Jessica Davies and Fahiem Bacchus. Solving maxsat by solving a sequence of simpler sat instances. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*, pages 225–239. Springer, 2011.

20 Pieter-Tjerk de Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 134(1):19–67, 2005.

21 Jian Ding, Allan Sly, and Nike Sun. Proof of the satisfiability conjecture for large k. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 59–68, 2015.

22 Ding-Zhu Du, Jun Gu, and Panos M. Pardalos, editors. *Satisfiability Problem: Theory and Applications, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, March 11-13, 1996*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. DIMACS/AMS, 1997.

23 Paul Erdős and John L Selfridge. On a combinatorial game. *Journal of Combinatorial Theory, Series A*, 14(3):298–301, 1973.

24 Paola Festa, Panos M Pardalos, Leonidas S Pitsoulis, and Mauricio GC Resende. Grasp with path-relinking for the weighted maximum satisfiability problem. In *International Workshop on Experimental and Efficient Algorithms*, pages 367–379. Springer, 2005.

25 Ehud Friedgut and Jean Bourgain. Sharp thresholds of graph properties, and the k-sat problem. *Journal of the American Mathematical Society*, 12(4):1017–1054, 1999.

26 Gregory Gutin and Abraham P Punnen. *The Traveling Salesman Problem and Its Variations*, volume 12. Springer Science & Business Media, 2006.

27 Gregory Gutin and Anders Yeo. Polynomial approximation algorithms for the tsp and the qap with a factorial domination number. *Discrete Applied Mathematics*, 119(1-2):107–116, 2002.

28 Johan Håstad. Some optimal inapproximability results. *Journal of the ACM (JACM)*, 48(4):798–859, 2001.

29 Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat: An efficient weighted max-sat solver. *Journal of Artificial Intelligence Research (JAIR)*, 31:1–32, 2008.

30 Holger H Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Elsevier, 2004.

31 Chu Min Li and Felip Manyà. *MaxSAT, hard and soft constraints*, volume 185, pages 613–631. IOS Press, 2009.

32 Cheng Luo, Shanyong Cai, Wenchuan Wu, Zhong Jie, and Kuan-Wu Su. Ccls: An efficient local search algorithm for weighted maximum satisfiability. *IEEE Transactions on Computers*, 64(7):1830–1843, 2014.

33 Chuan Luo, Shaowei Cai, Kaile Su, and Wei Wu. Clause states based configuration checking in local search for satisfiability. *IEEE transactions on Cybernetics*, 45(5):1028–1041, 2014.

34 Chuan Luo, Shaowei Cai, Wei Wu, and Kaile Su. Focused random walk with configuration checking and break minimum for satisfiability. In *International Conference on Principles and Practice of Constraint Programming*, pages 481–496. Springer, 2013.

35 Stephan Mertens, Marc Mézard, and Riccardo Zecchina. Threshold values of random k-sat from the cavity method. *Random Structures & Algorithms*, 28(3):340–373, 2006.

36 Patrick Mills and Edward Tsang. Guided local search for solving sat and weighted max-sat problems. *Journal of Automated Reasoning*, 24(1-2):205–223, 2000.

37 Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2717–2723. AAAI Press, 2014.

38 Denis Pankratov and Allan Borodin. On the relative merits of simple local search methods for the max-sat problem. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*, SAT'10, page 223–236, Berlin, Heidelberg, 2010. Springer-Verlag.

39 Matthias Poloczek. Bounds on greedy algorithms for max sat. In *Proceedings of the 19th European Conference on Algorithms*, ESA'11, page 37–48, Berlin, Heidelberg, 2011. Springer-Verlag.

40 Matthias Poloczek, Georg Schnitger, David P Williamson, and Anke Van Zuylen. Greedy algorithms for the maximum satisfiability problem: Simple algorithms and inapproximability bounds. *SIAM Journal on Computing*, 46(3):1029–1061, 2017.

41 Matthias Poloczek and David P Williamson. An experimental evaluation of fast approximation algorithms for the maximum satisfiability problem. In *International Symposium on Experimental Algorithms*, pages 246–261. Springer, 2016.

42 Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, page 337–343. American Association for Artificial Intelligence, 1994.

43 Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532, 1996.

44 Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446. AAAI Press, 1992.

45 Kevin Smyth, Holger H Hoos, and Thomas Stützle. Iterated robust tabu search for max-sat. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 129–144. Springer, 2003.

46 Sun grid engine (sge) quickstart. URL: `http://star.mit.edu/cluster/docs/0.93.3/guides/sge.html`.

47 Mihalis Yannakakis. On the approximation of maximum satisfiability. *Journal of Algorithms*, 17(3):475–502, 1994.

Enumerating All Subgraphs Under Given Constraints Using Zero-Suppressed Sentential Decision Diagrams

Yu Nakahata
Graduate School of Informatics, Kyoto University, Japan
nakahata.yu.27e@st.kyoto-u.ac.jp

Masaaki Nishino
NTT Communication Science Laboratories, NTT Corporation, Kyoto, Japan
masaaki.nishino.uh@hco.ntt.co.jp

Jun Kawahara
Graduate School of Informatics, Kyoto University, Japan
jkawahara@i.kyoto-u.ac.jp

Shin-ichi Minato
Graduate School of Informatics, Kyoto University, Japan
minato@i.kyoto-u.ac.jp

Abstract

Subgraph enumeration is a fundamental task in computer science. Since the number of subgraphs can be large, some enumeration algorithms exploit compressed representations for efficiency. One such representation is the Zero-suppressed Binary Decision Diagram (ZDD). ZDDs can represent the set of subgraphs compactly and support several poly-time queries, such as counting and random sampling. Researchers have proposed efficient algorithms to construct ZDDs representing the set of subgraphs under several constraints, which yield fruitful results in many applications. Recently, Zero-suppressed Sentential Decision Diagrams (ZSDDs) have been proposed as variants of ZDDs. ZSDDs can be smaller than ZDDs when representing the same set of subgraphs. However, efficient algorithms to construct ZSDDs are known only for specific types of subgraphs: matchings and paths.

We propose a novel framework to construct ZSDDs representing sets of subgraphs under given constraints. Using our framework, we can construct ZSDDs representing several sets of subgraphs such as matchings, paths, cycles, and spanning trees. We show the bound of sizes of constructed ZSDDs by the branch-width of the input graph, which is smaller than that of ZDDs by the path-width. Experiments show that our methods can construct ZSDDs faster than ZDDs and that the constructed ZSDDs are smaller than ZDDs when representing the same set of subgraphs.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Subgraph, Enumeration, Decision Diagram, Zero-suppressed Sentential Decision Diagram (ZSDD), Top-down construction algorithm

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.9

Funding This work was supported by JSPS KAKENHI Grant Number JP15H05711, JP18H04091, JP18K04610, and JP19J21000.

1 Introduction

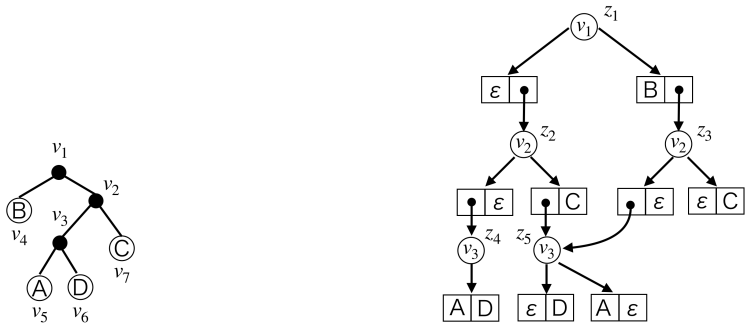
Enumerating subgraphs of a given graph under some constraint is a fundamental task in computer science. There are enumeration algorithms for several types of subgraphs such as cliques [5], paths [1], and spanning trees [21]. These algorithms list all subgraphs one by one in a small amount of time per subgraph. However, such algorithms take at least linear time and space to the number of subgraphs. Since the number of subgraphs can be exponentially larger than the size of the input graph, it is trouble when applied to practical problems.

Instead of explicitly listing subgraphs, some algorithms exploit compressed representations of sets of subgraphs. One such representation is the Zero-suppressed Binary Decision Diagram (ZDD) [15]. ZDDs are compact representations of set families. By regarding a subgraph as its edge set, we can express a set of subgraphs by a ZDD. ZDDs can not only represent set families compactly but also support several poly-time queries such as counting and random sampling [15]. In addition, given two ZDDs, we can efficiently construct a ZDD representing the union or intersection of the set families represented by the input ZDDs in polynomial time to the sizes of the input ZDDs. Such operations are called *Apply operations* [4]. Due to these merits, ZDDs appear in several graph-related applications such as network optimization [10], network reliability evaluation [7, 8], and balanced graph partitions [12, 16, 17].

A key to use ZDDs effectively for subgraph enumeration is a fast algorithm to construct a ZDD representing the set of subgraphs. It is time-consuming to construct a ZDD by first explicitly listing subgraphs and then combining ZDDs using Apply operations. In contrast, some algorithms can construct ZDDs *without explicitly listing subgraphs*. Such algorithms are called *top-down construction algorithms*, while algorithms using Apply operations are called *bottom-up*. Researchers have proposed top-down construction algorithms for ZDDs representing several sets of subgraphs [20, 7, 14]. Kawahara et al. [13] generalized the algorithms to obtain a general framework of top-down construction algorithms for ZDDs. Using the framework, we can construct ZDDs representing the sets of subgraphs under several constraints, such as the number of edges, degrees of vertices, and connectivity of vertices. By combining these fundamental constraints, we can specify several types of subgraphs, such as matchings, paths, cycles, and spanning trees.

Recently, Zero-suppressed Sentential Decision Diagrams (ZSDDs) [18] have been proposed as different representations of set families. Since ZSDDs are generalizations of ZDDs, ZSDDs are at least as compact as ZDDs. In theory, there exist set families that have polynomial ZSDD sizes but exponential ZDD sizes [3]. In addition, ZSDDs inherit some poly-time queries of ZDDs: counting, random sampling, and Apply operations. Thus, a natural question is: Can we design top-down construction algorithms for ZSDDs representing sets of subgraphs? The question is partially answered in an affirmative way by Nishino et al. [19]. They proposed top-down construction algorithms for ZSDDs representing sets of specific types of subgraphs: matchings and paths. The sizes of constructed ZSDDs by their algorithms are bounded by the *branch-width* of the input graph [19], while those of ZDDs are bounded by the *path-width* [11]. Since the branch-width of a graph never exceeds the path-width [2], ZSDDs have tighter upper bounds than ZDDs. The efficiency of their algorithms was confirmed in experiments. Despite such striking results, their algorithms are specific to matchings and paths.

In this paper, we propose a novel framework of top-down construction algorithms for ZSDDs. To design a top-down construction algorithm using our framework, one only has to prove a recursive formula for the desired set of subgraphs. Using the recursive formula, we can theoretically show the correctness and the complexity of the algorithm, which was difficult with the existing method. We apply our framework to the three fundamental constraints used in ZDDs: the number of edges, degrees of vertices, and connectivity of vertices. We show that the sizes of constructed ZSDDs are bounded by the branch-width of the input graph, not only for matchings and paths. Experiments show that proposed methods can construct ZSDDs faster than ZDDs and that the constructed ZSDDs are smaller than ZDDs representing the same sets of subgraphs.



(a) A vtree. **(b)** A ZSDD.

Figure 1 A vtree and a ZSDD that respects the vtree.

2 Preliminaries

2.1 Graphs

Let $G = (V, E)$ be an undirected graph where V is the vertex set and E is the edge set. $|V|$ and $|E|$ denote the number of vertices and edges, respectively. For edge subset $S \subseteq E$, the *induced subgraph* $G[S]$ is the subgraph $(V[S], S)$, where $V[S] \subseteq V$ is the set of vertices to which an edge in S is incident. In the following, we identify S with $G[S]$. For $S \subseteq E$ and $u \in V$, the *degree* $\deg(S, u)$ of u in S is the number of edges incident to u in S.

2.2 $(\mathbf{X}, \mathbf{Y})$-partition and vtree

Let f and g be set families. We define three operations between set families. We define *union* $\cup$, *intersection* $\cap$, and *join* $\sqcup$ as $f \cup g = \{a \mid a \in f \text{ or } a \in g\}$, $f \cap g = \{a \mid a \in f \text{ and } a \in g\}$, and $f \sqcup g = \{a \cup b \mid a \in f \text{ and } b \in g\}$, respectively.

▶ **Definition 1.** *Let f be a set family, and $\mathbf{X}, \mathbf{Y}$ be a partition of the universe of f. Set family f can be written as*

$$f = \bigcup_{i=1}^{h} [p_i \sqcup s_i], \tag{1}$$

where p_i and s_i are the set families whose universes are $\mathbf{X}$ and $\mathbf{Y}$, respectively. The equation is an $(\mathbf{X}, \mathbf{Y})$-decomposition. *We call $p_1, \ldots, p_h$* primes *and $s_1, \ldots, s_h$* subs. *If the primes are* exclusive (*$p_i \cap p_j = \emptyset$ for all $i \neq j$*), *the decomposition is an* $(\mathbf{X}, \mathbf{Y})$-partition. [1]

▶ **Example 2.** Let f_1 be the family of subsets of $U_1 = \{A, B, C, D\}$ that contain exactly two elements. It follows that $f_1 = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}\}$. For $\mathbf{X}_1 = \{B\}$ and $\mathbf{Y}_1 = \{A, C, D\}$, an $(\mathbf{X}_1, \mathbf{Y}_1)$-partition of f_1 is

$$f_1 = [\underbrace{\{\emptyset\}}_{\text{prime}} \sqcup \underbrace{f_2^1}_{\text{sub}}] \cup [\underbrace{\{\{B\}\}}_{\text{prime}} \sqcup \underbrace{f_2^2}_{\text{sub}}], \tag{2}$$

where $f_2^1 = \{\{A, C\}, \{A, D\}, \{C, D\}\}$ and $f_2^2 = \{\{A\}, \{C\}, \{D\}\}$.

[1] In [18], an $(\mathbf{X}, \mathbf{Y})$-decomposition is called an $(\mathbf{X}, \mathbf{Y})$-partition if the primes are exclusive and *consistent* ($p_i \neq \emptyset$ for all i). For simplicity, we do not require consistency for $(\mathbf{X}, \mathbf{Y})$-partitions. If we construct a ZSDD without consistency, we can make their primes consistent in linear time to the ZSDD size [19].

The universe of f_2^1 and f_2^2 is $U_2 = \{A, C, D\}$. For $\mathbf{X}_2 = \{A, D\}$ and $\mathbf{Y}_2 = \{C\}$, an $(\mathbf{X}_2, \mathbf{Y}_2)$-partition of f_2^1 is

$$f_2^1 = [\underbrace{\{\{A, D\}\}}_{\text{prime}} \sqcup \underbrace{\{\emptyset\}}_{\text{sub}}] \cup [\underbrace{\{\{A\}, \{D\}\}}_{\text{prime}} \sqcup \underbrace{\{\{C\}\}}_{\text{sub}}]. \tag{3}$$

A ZSDD represents a set family by recursively applying $(\mathbf{X}, \mathbf{Y})$-partitions to decompose the family into sub-families, where the order of partitions is determined by a *vtree*. A vtree is a rooted, ordered, and full binary tree whose leaves correspond to elements of the universe. Figure 1a shows an example. Symbols appearing in leaves represent corresponding elements, and symbols beside nodes represent their names. Each internal node represents a partition of the universe into two subsets: elements appearing in the left and right subtrees. We denote the left and right children of node v by v^l and v^r, respectively. In the figure, root node v_1 represents the $(\mathbf{X}_1, \mathbf{Y}_1)$-partition of the universe $U_1 = \{A, B, C, D\}$ where $\mathbf{X}_1 = \{B\}$ and $\mathbf{Y}_1 = \{A, C, D\}$. Similarly, node v_2 represents the $(\mathbf{X}_2, \mathbf{Y}_2)$-partition of the universe $U_2 = \{A, C, D\}$ where $\mathbf{X}_2 = \{A, D\}$ and $\mathbf{Y}_2 = \{C\}$. To avoid confusion, we call vtree nodes *vnodes*, ZSDD nodes *znodes*, and graph nodes *vertices*. We represent them as v_i, z_i, and u_i.

2.3 Zero-suppressed Sentential Decision Diagrams

A ZSDD is recursively defined as follows. ZSDD α *respects* vnode v if the order of $(\mathbf{X}, \mathbf{Y})$-partitions in α follows the vtree whose root is v. $\langle\alpha\rangle$ denotes the set family that α represents.

▶ **Definition 3.** *α is a ZSDD that respects vnode v if and only if:*

- *$\alpha = \varepsilon$ or $\alpha = \bot$. (Semantics: $\langle\varepsilon\rangle = \{\emptyset\}$ and $\langle\bot\rangle = \emptyset$.)*
- *$\alpha = X$ or $\alpha = \pm X$ and v is a leaf with element X. (Semantics: $\langle X\rangle = \{\{X\}\}$ and $\langle\pm X\rangle = \{\{X\}, \emptyset\}$.)*
- *$\alpha = \{(p_1, s_1), \ldots, (p_h, s_h)\}$, v is internal, $p_1, \ldots, p_h$ are ZSDDs that respect a vnode in the subtree whose root is v^l, $s_1, \ldots, s_h$ are ZSDDs that respect a vnode in the subtree whose root is v^r, and $\langle p_1\rangle, \ldots, \langle p_h\rangle$ are exclusive. (Semantics: $\langle\alpha\rangle = \bigcup_{i=1}^{h} [\langle p_i\rangle \sqcup \langle s_i\rangle]$.)*

If a ZSDD is either $\varepsilon, \bot, X$, or $\pm X$, it is a *terminal*. Otherwise, it is a *decomposition*. Figure 1b shows an example ZSDD that represents set family f_1 in Example 2 and respects the vtree in Figure 1a. A circle node and its child rectangle nodes represent an $(\mathbf{X}, \mathbf{Y})$-partition. The symbol in a circle node indicates the vnode that the decomposition respects. A pair of rectangle nodes represent a prime-sub pair in an $(\mathbf{X}, \mathbf{Y})$-partition where the left and right are prime p and sub s, respectively. Every p and s is either a terminal ZSDD or a pointer to a decomposition ZSDD. Circle nodes are *decomposition znodes*, and rectangle nodes are *element znodes*. For example, znodes z_1 and z_2 represent the $(\mathbf{X}, \mathbf{Y})$-partitions in Equations (2) and (3), respectively. The *size* of a ZSDD is the sum of the sizes of $(\mathbf{X}, \mathbf{Y})$-partitions in the ZSDD. The size of the ZSDD in Figure 1b is 9. [2]

3 A novel framework of top-down ZSDD construction

We present a novel framework of top-down ZSDD construction. Our framework is partially identical to that of Nishino et al.'s [19], but we modify it so that we can design algorithms easily for several constraints. Algorithm 1 shows the framework. The algorithm takes graph G and the root vnode as its inputs and returns a ZSDD representing a set of subgraphs

[2] The size of a ZDD is defined as the number of nodes. [15] This is because, every node of a ZDD has exactly two children. In contrast, nodes of a ZSDD may have different number of children, and thus the size of a ZSDD is defined as the number of arcs.

Algorithm 1 A top-down construction algorithm.

Input : A graph $G = (V, E)$ and the root vtree node v
Output : A ZSDD representing a set of subgraphs of G

```
1 Z[v] ← rootState()
2 construct(v, Z)
3 Z ← reduce(Z)
4 return Z
```

Algorithm 2 construct(v, Z).

```
1  for z ∈ Z[v] do
2      elems ← ∅
3      for (m^l, m^r) ∈ decomp(v, z) do
4          for ∘ ∈ {l, r} do
5              if v^∘ is a leaf vnode then z^∘ ← terminal(v^∘, m^∘)
6              else z^∘ ← unique(v^∘, m^∘, Z)
7          elems ← elems ∪ {(z^l, z^r)}
8      Set elems as the child znodes of z
9      for ∘ ∈ {l, r} do
10         if v^∘ is an internal vnode then construct(v^∘, Z)
```

of G. $Z[v]$ stores a set of decomposition znodes that respect vnode v. Since a ZSDD is represented as a set of decomposition znodes, the set of $Z[v]$'s for all internal vnodes v can be seen as a ZSDD. The algorithm first calls rootState(), which returns the root znode. The procedure depends on the types of subgraphs. The algorithm next calls construct(v, Z), which recursively construct child znodes of znodes respecting v. If we naively construct znodes, the number of child znodes grows exponentially. We thus merge *equivalent* znodes during the construction of a ZSDD. Here, two znodes are equivalent if they respect the same vnode and represent the same family of sets. To detect equivalent znodes efficiently, we attach a *label* to each znode. The labels must be defined depending on the types of subgraphs so that two znodes are equivalent if they respect the same vnode and have the same label. We explain how to design labels in Section 4. The constructed ZSDD may have redundant znodes. Function reduce(Z) deletes such znodes.

Algorithm 2 shows function construct(v, Z). The function is called only for internal vnodes. In [19], the procedure of construct(v, Z) was designed depending on whether v^l is a leaf or not. Instead, we treat all internal vnodes in the same way, which makes it easy to design algorithms for several constraints. For each znode z in $Z[v]$, the function calculates the prime-sub pairs corresponding to z. We first initialize the set of prime-sub pairs `elems` to the empty set (Line 2). Function decomp(v, z) receives vnode v and znode z that respects v, and returns the set of pairs of labels corresponding to the prime-sub pairs (Line 3). For each $\circ \in \{l, r\}$, if v° is a leaf vnode, we set znode z° to a terminal (Line 5). Function terminal(v, m) receives leaf vnode v and label m, and returns an appropriate terminal depending on the types of subgraphs. If v° is an internal vnode, we call unique(v, m, Z) (Line 6). The function receives vnode v and label m, and checks whether $Z[v]$ contains a znode with label m. If such a znode exists, the function returns its address. Otherwise, the function creates a new znode that respects v and has label m, stores it into $Z[v]$, and returns its address. We add the prime-sub pair (z^l, z^r) into `elems` (Line 7). After generating all the prime-sub pairs, we set `elems` as the child znodes of z (Line 8). Finally, for each $\circ \in \{l, r\}$ such that v° is an internal vnode, we call construct(v°, Z) to recursively construct sub-ZSDDs (Lines 9–10).

The functions reduce(Z) and unique(v, m, Z) can be designed regardless of the types of subgraphs [19]. In contrast, the definition of labels and the procedures of rootState(), terminal(v, m), and decomp(v, z) heavily depend on the types of subgraphs. To easily design them for several constraints, we relate a recursive formula for the desired set of subgraphs to top-down ZSDD construction. Intuitively, in our framework, internal vnodes correspond to recursion steps, while leaf vnodes correspond to base cases. Therefore, we only have to prove a recursive formula for the desired set of subgraphs. The recursive formula directly leads to the definition of labels and the procedures of subroutines. We can also show the correctness of the algorithm and the bound of the constructed ZSDD size from the recursive formula.

4 Subroutines for several constraints

We apply our framework to three fundamental constraints: the number of edges, degrees of vertices, and connectivity of vertices. By combining these constraints, we can specify several types of subgraphs. For each constraint, we show a recursive formula for the set of subgraphs satisfying the constraint. Using the recursive formula, we derive subroutines and bound the sizes of constructed ZSDDs. The proofs are omitted due to the space limitation.

4.1 Cardinality

Given graph $G = (V, E)$, vtree T whose leaves are labeled by the elements of E, and non-negative integer k^*, we construct a ZSDD that represents the family of sets with exactly k^* elements. We can also construct a ZSDD that represents the family of sets with at most or at least k^* elements (details are omitted). In the following, we focus on the "exactly k^*" constraint. For vnode v, let $E(v) \subseteq E$ be the set of graph edges that correspond to the leaf vnodes of the sub-vtree whose root is v. For vnode v and non-negative integer k, let $f(v, k)$ be the family of subsets of $E(v)$ with k elements, that is, $f(v, k) = \{S \mid S \subseteq E(v), |S| = k\}$. The desired family is $f(v^{\mathrm{root}}, k^*)$, where v^{root} is the root vnode of T. For leaf vnode v, $\ell(v)$ denotes the element corresponding to v. We show a recursive formula for $f(v, k)$.

▶ **Lemma 4.** *Let v be a vnode, and k be a non-negative integer. If v is a leaf vnode, then the following hold:*

$$f(v,k) = \begin{cases} \{\emptyset\} & (k=0) \\ \{\{\ell(v)\}\} & (k=1) \\ \emptyset & (\text{otherwise}) \end{cases} \tag{4}$$

If v is internal, the following is an $(E(v^l), E(v^r))$-partition:

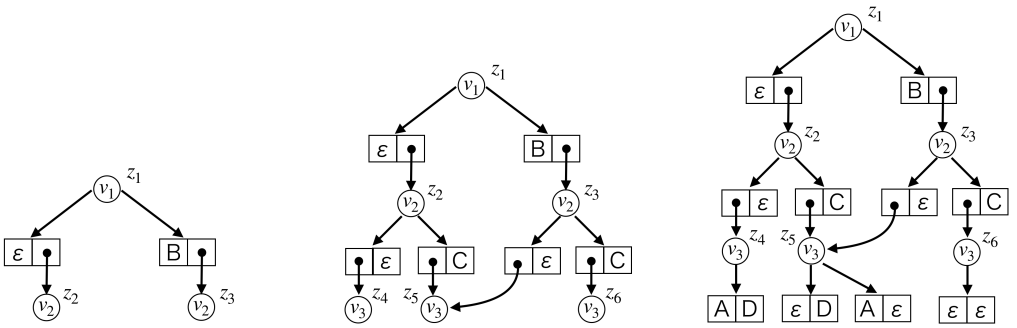
$$f(v,k) = \bigcup_{i=0}^{k} \left[f(v^l, i) \sqcup f(v^r, k-i) \right]. \tag{5}$$

Using the recursive formula, we can design the subroutines of the framework. We use non-negative integers as znode labels. For znode z that respects vnode v, the label of z indicates the number of elements that should be adopted from $E(v)$. Function rootState() returns the root znode with label k^*, since the desired family is $f(v^{\mathrm{root}}, k^*)$. Algorithm 3 shows the subroutines terminal(v, k) and decomp(v, z). terminal(v, k) is obtained from Equation (4). If $k = 0$, it returns ε since $\langle \varepsilon \rangle = \{\emptyset\}$ (Line 1). If $k = 1$, it returns $\ell(v)$ since $\langle \ell(v) \rangle = \{\{\ell(v)\}\}$ (Line 2). Otherwise, it returns $\bot$ since $\langle \bot \rangle = \emptyset$ (Line 3). Similarly, decomp(v, z) is obtained from Equation (5). The function initializes `elems` to the empty set (Line 4). Let k be the

Algorithm 3 Subroutines for the cardinality constraint.

```
   Function : terminal(v, k)
1  if k = 0 then return ε
2  else if k = 1 then return ℓ(v)
3  else return ⊥

   Function : decomp(v, z)
4  elems ← ∅
5  Let k be the label of z
6  for i ∈ [0, k] do
7      elems ← elems ∪ {(i, k − i)}
8  return elems
```



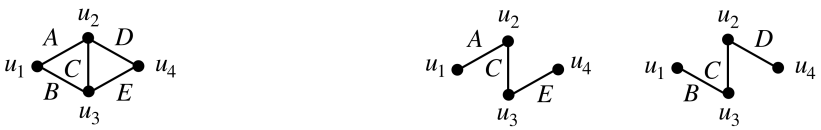
(a) After construct(v_1, Z). **(b)** After construct(v_2, Z). **(c)** After construct(v_3, Z).

Figure 2 Intermediate ZSDDs for the cardinality constraint.

label of z (Line 5). If the prime has label $0 \leq i \leq k$, then the sub has label $k - i$. Thus, we add the pair $(i, k - i)$ to `elems` (Lines 6–7). Finally, we return `elems` (Line 8). The correctness of the algorithm directly follows from the correctness of Lemma 4.

▶ **Example 5.** Let us construct a ZSDD that represents the family of subsets of $\{A, B, C, D\}$ with exactly two elements. We use the vtree in Figure 1a. First, rootState() creates root znode z_1 with label 2 and stores it into $Z[v_1]$. The function then calls construct(v_1, Z). $Z[v_1]$ contains only one znode z_1. Since z_1 has label 2, decomp(v_1, z_1) returns $\{(0, 2), (1, 1), (2, 0)\}$. The function first processes label pair $(0, 2)$. Since $v_1^l = v_4$ is a leaf vnode, the function calls terminal$(v_4, 0)$, which returns ε. Since $v_1^r = v_2$ is not a leaf vnode, the function calls unique$(v_2, 2, Z)$. It creates new decomposition znode z_2 that respects v_4 and has label 2, stores it into $Z[v_4]$, and returns its address. Similarly, for label pair $(1, 1)$, the corresponding prime-sub pair is calculated as (B, z_3), where z_3 is a new decomposition znode that respects v_2 and has label 1. As for label pair $(2, 0)$, since the universe of the prime contains only one element, we discard this pair. As a result, the function set the prime-sub pairs (ε, z_2) and (B, z_3) as child znodes of z_1. Figure 2a shows the current intermediate ZSDD. Since $v_1^l = v_4$ is a leaf vnode and $v_1^r = v_2$ is an internal vnode, the function calls only construct(v_2, Z).

We go on to construct(v_2, Z). $Z[v_2]$ contains two znodes z_2 and z_3. The function processes z_2 first. Since z_2 has label 2, decomp(v_2, z_2) returns $\{(2, 0), (1, 1), (0, 2)\}$. However, $(0, 2)$ is discarded because the universe of the sub only contains one element. As a result, the prime-sub pairs are calculated as $\{(z_4, \varepsilon), (z_5, C)\}$, where z_4 and z_5 are new decomposition znodes that respect v_3. The labels of z_4 and z_5 are 2 and 1, respectively. The function processes z_3 next. decomp(v_2, z_3) returns $\{(1, 0), (0, 1)\}$. Here, znode z_5 with label 1 already exists in $Z[v_3]$, and thus unique$(v_3, 1, Z)$ returns z_5. As a result, the set of prime-sub pairs is $\{(z_5, \varepsilon), (z_6, C)\}$, where z_6 is a new znode that respects v_3 and has label 0. Figure 2b shows the current intermediate ZSDD. Finally, construct(v_3, Z) is called and Figure 2c shows the resulting ZSDD. By calling reduce(Z), the ZSDD can be trimmed as Figure 1b.



(a) A graph. **(b)** Subgraphs.

Figure 3 A graph and its subgraphs satisfying a degree constraint.

Using Lemma 4, we can also bound the size of the constructed ZSDD.

▶ **Theorem 6.** *If α is the ZSDD obtained by Algorithm 3, the size of α is $\mathcal{O}(|E|k^2)$.*

4.2 Degree

We denote a given degree constraint by function $\delta^*\colon V \to \mathbb{N}$, where $\mathbb{N}$ is the set of non-negative integers. For subgraph $S \subseteq E$, we say that S *satisfies* δ^* if $\deg(S, u) = \delta^*(u)$ holds for all $u \in V$. For example, for the graph shown in Figure 3a and degree constraint δ^* such that $\delta^*(u_1) = \delta^*(u_4) = 1$ and $\delta^*(u_2) = \delta^*(u_3) = 2$, there are two subgraphs satisfying δ^* as shown in Figure 3b. Given G, T, and δ^*, we construct a ZSDD representing the set of all subgraphs satisfying δ^*. When a subgraph satisfies δ^*, for every vertex u, the degree of u in a subgraph must be "exactly" $\delta^*(u)$. Although we mainly discuss this "exact" constraint, we can easily modify the algorithm to deal with "at most" or "at least" constraints.

Similarly to Lemma 4, we show a recursive formula for the set of subgraphs satisfying the degree constraint. For vnode v, $V(v)$ denotes the set of vertices to which an edge in $E(v)$ is incident. Let us consider a degree constraint whose domain is limited to $V(v)$ as function $\delta\colon V(v) \to \mathbb{N}$. We define $f(v, \delta)$ as the family of subsets of $E(v)$ such that, for all $u \in V(v)$ and $S \in f(v, \delta)$, degree $\deg(S, u)$ equals $\delta(u)$. We show a recursive formula for $f(v, \delta)$.

▶ **Lemma 7.** *Let v be a vnode, and δ be a function from $V(v)$ to $\mathbb{N}$. If v is a leaf vnode, let u_1 and u_2 be the endpoints of graph edge $\ell(v)$. Then, the following hold:*

$$f(v,\delta) = \begin{cases} \{\emptyset\} & (\delta(u_1) = \delta(u_2) = 0) \\ \{\{\ell(v)\}\} & (\delta(u_1) = \delta(u_2) = 1) \\ \emptyset & (\text{otherwise}) \end{cases} \tag{6}$$

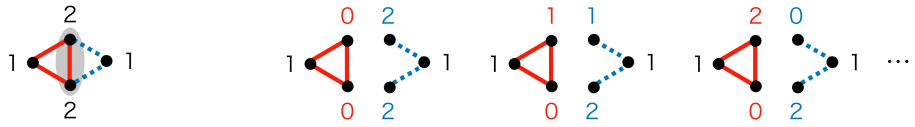
If v is internal, the following is an $(E(v^l), E(v^r))$-partition:

$$f(v,\delta) = \bigcup_{(\delta^l,\delta^r)\in P(v,\delta)} [f(v^l,\delta^l) \sqcup f(v^r,\delta^r)], \tag{7}$$

where $P(v, \delta)$ is the set of pairs of functions $\delta^l\colon V(v^l) \to \mathbb{N}$ and $\delta^r\colon V(v^r) \to \mathbb{N}$ such that

$$\forall u \in V(v^l) \cap V(v^r), \quad \delta^l(u) + \delta^r(u) = \delta(u), \tag{8}$$
$$\forall u \in V(v^l) \setminus V(v^r), \quad \delta^l(u) = \delta(u), \tag{9}$$
$$\forall u \in V(v^r) \setminus V(v^l), \quad \delta^r(u) = \delta(u). \tag{10}$$

For vnode v, the *frontier* of v is $F(v) = V(v^l) \cap V(v^r)$. Let us consider the graph shown in Figure 3a and the degree constraint δ^*, which we defined above. For vnode v, let $E(v^l) = \{A, B, C\}$ and $E(v^r) = \{D, E\}$. It follows that $F(v)$ is $\{u_2, u_3\}$. Figure 4a shows the current situation. The set of red (solid) and blue (dashed) edges are $E(v^l)$ and $E(v^r)$,



(a) A graph. **(b)** Prime-sub pairs.

Figure 4 A graph and corresponding prime-sub pairs.

respectively. The set of vertices in the shaded area is $F(v)$. We can interpret Equations (7) to (10) as follows. For vertex $u \in V(v^l) \setminus V(v^r)$, $\delta(u)$ edges in $E(v^l)$ must be incident to u, and thus $\delta^l(u) = \delta(u)$ (Equation (9)). A similar statement holds for vertices in $V(v^r) \setminus V(v^l)$ (Equation (10)). The remaining vertices are in $F(v)$. For vertex $u \in F(v)$, both edges in $E(v^l)$ and $E(v^r)$ are incident to u. Here, we guess how many edges in $E(v^l)$ are incident to u. This results in generating nine prime-sub pairs, as shown in Figure 4b. We can construct the ZSDD by recursively applying Lemma 7. Here we use δ as a label of a znode.

Let us analyze the sizes of ZSDDs constructed by our algorithm. The *width* of a vtree is $\max_{v \in \mathrm{in}(T)} V(v^l) \cap V(v^r)$, where $\mathrm{in}(T)$ is the set of internal vnodes.

▶ **Theorem 8.** *If α is the ZSDD representing $f(v^{\mathrm{root}}, \delta^*)$ obtained by our algorithm, the size of α is $\mathcal{O}(|E|d^{2W})$, where $d = \max_{u \in V} \delta^*(u) + 1$ and W is the width of the input vtree.*

There exists a vtree whose width equals the *branch-width* of the graph [19]. Given such a vtree, the ZSDD size is $\mathcal{O}(|E|d^{2\mathrm{bw}(G)})$, where $\mathrm{bw}(G)$ is the branch-width of G.

4.3 Spanning tree

We construct a ZSDD representing the set of all spanning trees of G. With a few modifications, we can also construct a ZSDD representing the set of all connected subgraphs. We introduce some notation. If vertices u, u' are connected in subgraph $S \subseteq E$, we write $u \overset{S}{\sim} u'$. Note that $\overset{S}{\sim}$ is an equivalence relation on V; an equivalence class (a set of vertices) is a *connected component* of S. Two vertex subsets $C, C' \subseteq V$ are *connected* if there exist $u \in C$ and $u' \in C'$ with $u \overset{S}{\sim} u'$; we write this as $C \overset{S}{\sim} C'$. We also write $u \overset{S}{\sim} C'$ if $C \overset{S}{\sim} C'$ for $C = \{u\}$.

For vnode v, let $\mathcal{C}$ be a partition of vertex set $F(v)$, that is, $\mathcal{C} = \{C_1, \ldots, C_g\}$ where $C_i \subseteq F(v)$ is a vertex set satisfying $C_i \cap C_j = \emptyset$ for $i \neq j$ and $\bigcup_{i=1}^{g} C_i = F(v)$. Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a disjoint set family defined over vertex sets in $\mathcal{C}$, that is, $R_i \subseteq \mathcal{C}$ and $R_i \cap R_j = \emptyset$ for all $i \neq j$. Let $U(\mathcal{R}) = \{C \mid \exists i : C \in R_i\}$. Function $\mathsf{Same}(\mathcal{R}, C, C')$ returns `true` if there exists $R_i \in \mathcal{R}$ such that $C, C' \in R_i$, otherwise `false`. To represent the set of all spanning trees, we define $f(v, \mathcal{C}, \mathcal{R})$ as the set of subgraphs $S \subseteq E(v)$ satisfying the following:

- for every $C_1, C_2 \in U(\mathcal{R})$, $C_1 \overset{S}{\sim} C_2$ holds if and only if $\mathsf{Same}(\mathcal{R}, C_1, C_2) = \mathtt{true}$,
- for every $C \in \mathcal{C} \setminus U(\mathcal{R})$, there exists a unique $C' \in U(\mathcal{R})$ such that $C \overset{S}{\sim} C'$. Similarly, for every $u \in V(v) \setminus F(v)$, there exists a unique $C' \in U(\mathcal{R})$ such that $u \overset{S}{\sim} C'$, and
- S does not contain a cycle.

Intuitively, $\mathcal{C}$ represents the sets of equivalent vertices. That is, vertices in the same vertex group $C \in \mathcal{C}$ are regarded to be connected. $\mathcal{R}$ represents the connectivity constraints over such equivalent sets of vertices. The first condition above requires that two vertex subsets C and C' must be connected in S if and only if they appear in the same $R \in \mathcal{R}$. The second condition requires that, every equivalent vertex subset appearing in $V(v)$ but does not appear in $\mathcal{R}$ must be connected to a vertex subset C' appearing in $\mathcal{R}$. The third

Algorithm 4 Subroutines for spanning trees.

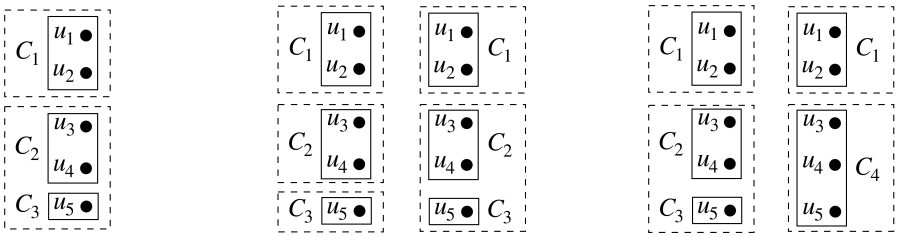
Function : terminal$(v, (\mathcal{C}, \mathcal{R}))$
1 Let u_1 and u_2 be the endpoints of the graph edge $\ell(v)$
2 **if** Same$(\mathcal{C}, u_1, u_2) =$ `true` **then**
3 Let $C \in \mathcal{C}$ be the set containing u_1 and u_2
4 **if** $C \in U(\mathcal{R})$ **then return** ε **else return** $\bot$
5 **else**
6 Let $C_1, C_2 \in \mathcal{C}$ be the sets containing u_1 and u_2, respectively
7 **if** *neither* C_1 *nor* C_2 *is in* $U(\mathcal{R})$ **then return** $\bot$
8 **else if** *exactly one of* C_1 *or* C_2 *is in* $U(\mathcal{R})$ **then return** $\ell(v)$
9 **else**
10 **if** Same$(\mathcal{R}, C_1, C_2) =$ `true` **then return** $\ell(v)$ **else return** ε

Function : decomp(v, z)
11 `elems` $\leftarrow \emptyset$
12 Let $(\mathcal{C}, \mathcal{R})$ be the label of z
13 $\mathcal{C}^l \leftarrow \left\{C \cap F(v^l) \;\middle|\; C \in \mathcal{C}, C \cap F(v^l) \neq \emptyset\right\} \cup \left\{\{u\} \;\middle|\; u \in F(v^l) \setminus F(v)\right\}$
14 **for** $\mathcal{R}^l \in$ enumPartition$(\mathcal{C}^l)$ **do**
15 **if** isCompatible$(\mathcal{C}, \mathcal{R}, \mathcal{R}^l) =$ `true` **then**
16 $\mathcal{C}^r, \mathcal{R}^r \leftarrow$ calcSubState$(\mathcal{C}, \mathcal{R}, \mathcal{R}^l)$
17 `elems` $\leftarrow$ `elems` $\cup \left\{((\mathcal{C}^l, \mathcal{R}^l), (\mathcal{C}^r, \mathcal{R}^r))\right\}$

condition is for acyclicity. The set of all spanning trees of G is $f(v^{\mathrm{root}}, \mathcal{C}^*, \mathcal{R}^*)$, where $\mathcal{C}^* = \{\{u\} \mid u \in F(v^{\mathrm{root}})\}$ and $\mathcal{R}^* = \{\{C\}\}$ for an arbitrary $C \in \mathcal{C}^*$ since initially there are no equivalent vertices and all vertices must be connected to form a spanning tree.

Unfortunately, it is quite complicated to show a recursive formula for $f(v, \mathcal{C}, \mathcal{R})$ and prove it theoretically. Thus, we show pseudo-code of subroutines and explain the behavior using an example. We use $(\mathcal{C}, \mathcal{R})$ as a znode label. rootState() returns the root znode label $(\mathcal{C}^*, \mathcal{R}^*)$. Algorithm 4 shows functions terminal$(v, (\mathcal{C}, \mathcal{R}))$ and decomp(v, z). terminal$(v, (\mathcal{C}, \mathcal{R}))$ returns an appropriate terminal with respect to the label of z. Let u_1 and u_2 be the endpoints of edge $\ell(v)$. We first consider the case that u_1 and u_2 are contained in the same vertex group $C \in \mathcal{C}$ (Lines 2–4). If $C \notin U(\mathcal{R})$, C must be connected to some $C' \in U(\mathcal{R})$. However, now we have $\mathcal{C} = \{C\}$, and thus there is no such C'. Therefore, we return $\bot$. If $C \in U(\mathcal{R})$, to avoid generating a cycle, we must not adopt edge $\ell(v)$. Thus we return ε. We next consider the case that u_1 and u_2 are contained in different sets $C_1, C_2 \in \mathcal{C}$ (Lines 5–10). If neither C_1 nor C_2 appear in constraints $\mathcal{R}$, they must be connected to some $C' \in U(\mathcal{R})$, but there are no such C'. Thus we return $\bot$ (Line 7). If either of C_1 or C_2 appears in $\mathcal{R}$, the unconstrained one must be connected with the other one, which has a constraint in $\mathcal{R}$. Thus we return $\ell(v)$ (Line 8). If both C_1 and C_2 appear in $\mathcal{R}$, we return the corresponding terminal depending on whether they appear in the same $R_i \in \mathcal{R}$ or not. If so, edge $\ell(v)$ must be adopted, and thus we return $\ell(v)$. Otherwise, the edge must not be adopted, and thus we return ε (Lines 9–10).

We go on to decomp(v, z). We first enumerate all possible set of constraints $\mathcal{R}^l$ of the prime. Since $\mathcal{R}^l$ is a partition of vertex groups $\mathcal{C}$, function enumPartition$(\mathcal{C}^l)$ enumerates all partitions of $\mathcal{C}^l$. There may be partitions of $\mathcal{C}^l$ that are not *compatible* with $(\mathcal{C}, \mathcal{R})$; If $C_1 \in R_i$ and $C_2 \in R_j$ for $R_i, R_j \in \mathcal{R}$ where $i \neq j$, they must not appear in the same $R \in \mathcal{R}^l$. In addition, for every constraint $R \in \mathcal{R}^l$, a vertex in $F(v^l)$ must appear in some $C \in R$ in order to obtain a spanning tree. If both conditions are satisfied, $\mathcal{R}^l$ is compatible with $(\mathcal{C}, \mathcal{R})$. Function isCompatible$(\mathcal{C}, \mathcal{R}, \mathcal{R}^l)$ returns `true` if $\mathcal{R}^l$ is compatible with $(\mathcal{C}, \mathcal{R})$,



(a) A state. **(b)** Prime-sub pairs.

Figure 5 Label of the connectivity constraint and corresponding prime-sub pairs.

otherwise `false`. calcSubState$(\mathcal{C}, \mathcal{R}, \mathcal{R}^l)$ calculates $\mathcal{C}^r$ and $\mathcal{R}^r$ from its arguments. Intuitively, $\mathcal{C}^r$ and $\mathcal{R}^r$ are obtained by updating equivalent vertex groups in $\mathcal{C}$ by assuming constraints in $\mathcal{R}^l$ are satisfied. Let us give an example. Figure 5a shows a label and Figure 5b shows the corresponding prime-sub pairs. Five vertices $u_1, \ldots, u_5$ are on the frontier. We assume $F(v^l) = F(v^r) = F(v)$ in this example. In Figure 5a, the vertices are partitioned into three equivalency groups $\mathcal{C} = \{C_1, C_2, C_3\}$, where $C_1 = \{u_1, u_2\}$, $C_2 = \{u_3, u_4\}$, and $C_3 = \{u_5\}$. $\mathcal{C}$ is further partitioned into $\mathcal{R} = \{\{C_1\}, \{C_2, C_3\}\}$. $\mathcal{C}$ and $\mathcal{R}$ are depicted by solid and dashed rectangles, respectively. There are only two $\mathcal{R}^l$'s that are compatible with $(\mathcal{C}, \mathcal{R})$: $\mathcal{R}^l_1 = \{\{C_1\}, \{C_2\}, \{C_3\}\}$ and $\mathcal{R}^l_2 = \{\{C_1\}, \{C_2, C_3\}\}$. calcSubState$(\mathcal{C}, \mathcal{R}, \mathcal{R}^l_1)$ returns $(\mathcal{C}^r_1, \mathcal{R}^r_1)$, where $\mathcal{C}^r_1 = \{C_1, C_2, C_3\}$ and $\mathcal{R}^r_1 = \{\{C_1\}, \{C_2, C_3\}\}$. calcSubState$(\mathcal{C}, \mathcal{R}, \mathcal{R}^l_2)$ returns $(\mathcal{C}^r_2, \mathcal{R}^r_2)$, where $\mathcal{C}^r_2 = \{C_1, C_4\}$, $\mathcal{R}^r_2 = \{\{C_1\}, \{C_4\}\}$, and $C_4 = C_2 \cup C_3 = \{u_3, u_4, u_5\}$.

Finally, the following theorem states the bound of constructed ZSDD size.

▶ **Theorem 9.** *If α is a ZSDD representing the set of all spanning trees constructed by our top-down algorithm, the size of α is $\mathcal{O}(|E|W^{3W})$, where W is the width of the vtree.*

As discussed in Section 4.2, there exists a vtree whose width equals the branch-width of the graph. Given such a vtree, the size of a constructed ZSDD is $\mathcal{O}(|E|\mathrm{bw}(G)^{3\mathrm{bw}(G)})$.

5 Experiments

We conduct experiments to evaluate the performance of the proposed top-down construction algorithms for ZSDDs in the same way as an existing paper [19]. The vtrees for ZSDDs are obtained by a practical algorithm to find a branch decomposition with a small width [6]. To implement the top-down algorithm for ZDDs, we use the top-down algorithm for ZSDDs with a limitation that vtrees must be right-linear. Here, a vtree is *right-linear* if, for every internal vnode, its left child is a leaf. Since there is a one-to-one correspondence between ZDDs with ZSDDs using right-linear vtrees, by inputting right-linear vtrees, we can simulate ZDD construction. We use two element orders for ZDDs. The first one uses the order obtained by a breadth-first traversal of input graphs, as is used in graphillion [9], a library that implements a top-down construction algorithm for ZDDs. The other one uses the order induced from the vtrees used in the proposed method. Here we say an order is induced if a left-right traversal of a vtree gives the visiting order of variables [22]. We use the benchmark graphs of [19]: TSPLIB and RomeGraph. We constructed ZSDDs representing two types of subgraphs: 1) maximum degree at most two and 2) spanning trees. All code was written in C++ and compiled by g++-5.4.0 with -O3 option. All experiments were conducted on a machine with Intel Xeon W-2133 3.60 GHz CPU and 256 GB RAM.

Table 1 Results of constructing ZSDDs and ZDDs representing the set of all subgraphs whose maximum degrees are at most 2.

			Time (ms)			Size		
instance	$\|V\|$	$\|E\|$	TD	Z(b)	Z(v)	TD	Z(b)	Z(v)
att48	48	130	381	6801	2291	194786	1065745	507169
berlin52	52	145	1021	-	36354	807660	-	5229861
eil51	51	142	1012	247736	46524	774280	27277682	5974875
grafo10106	100	119	5	2617	16	2658	15461	7529
grafo10124	100	139	9237	-	40842	3060950	-	3283397
grafo10153	100	136	3784	-	4658	832943	-	561283
grafo10183	100	132	132	-	157837	80127	-	4088915
grafo10184	100	140	4981	-	119366	1006210	-	2002968
grafo10204	100	148	156529	-	303366	15712819	-	19847326
grafo10223	100	135	863	-	5956	330554	-	826121

Table 2 Results of constructing ZSDDs and ZDDs representing the set of all spanning trees.

			Time (ms)			Size		
instance	$\|V\|$	$\|E\|$	TD	Z(b)	Z(v)	TD	Z(b)	Z(v)
att48	48	130	3494	103871	3005	279613	5098205	387715
berlin52	52	145	11826	-	62706	937746	-	3194017
eil51	51	142	25828	-	94272	838254	-	7178190
ulysses22	22	56	39	3391	65	3036	520035	16762
grafo10106	100	119	28	221161	53	1756	836212	4057
grafo10183	100	132	2866	-	538878	224373	-	16414697
grafo10223	100	135	48563	-	128097	1009299	-	7313087
grafo10248	100	126	301	195249	672	16524	1617024	47605

Tables 1 and 2 show the results. In the tables, TD means the proposed method. Z(b) and Z(v) indicate top-down methods for ZDDs that employ breadth-first ordering and vtree traversing ordering, respectively. The empty fields indicate failure to complete within 600 seconds. We omit the instances for which all the methods finished within a second and at most one method finished within 600 seconds. In almost all cases, TD ran fastest and the sizes of ZSDDs are smaller than those of ZDDs. For example, for spanning trees (Table 2), the time of TD is up to 7898 times faster than Z(b), and 188 times faster than Z(v). The size of TD is up to 476 times smaller than Z(b) and 73 times smaller than Z(v). These results show the efficiency of our method. Using constructed ZDDs and ZSDDs, we can also enumerate subgraphs explicitly in polynomial time per subgraph [15, 18].

6 Concluding remarks

We have proposed a novel framework of algorithms for top-down ZSDD construction. We have shown the solid subroutines for three fundamental constraints: the number of edges, degree of vertices, and connectivity of vertices. We have shown the sizes of constructed ZSDDs can be bounded by the branch-width of the input graph. Experiments confirmed the efficiency of our method. Using Apply operations, we can combine several constraints. For example, we can extract connected subgraphs from ZSDD α by constructing ZSDD β representing the set of all connected subgraphs and computing $\alpha \cap \beta$. We believe that our framework can be used to solve various real-world problems.

References

1 Etienne Birmelé, Rui A. Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. Optimal listing of cycles and st-paths in undirected graphs. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1884–1896. SIAM, 2013. doi:10.1137/1.9781611973105.134.

2 Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, 1998. doi:10.1016/S0304-3975(97)00228-4.

3 Simone Bova. Sdds are exponentially more succinct than obdds. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 929–935. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12270.

4 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. doi:10.1109/TC.1986.1676819.

5 Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. Sublinear-space bounded-delay enumeration for massive network analytics: Maximal cliques. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 148:1–148:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.148.

6 William J. Cook and Paul D. Seymour. Tour merging via branch-decomposition. *INFORMS J. Comput.*, 15(3):233–248, 2003. doi:10.1287/ijoc.15.3.233.16078.

7 Gary Hardy, Corinne Lucet, and Nikolaos Limnios. K-terminal network reliability measures with binary decision diagrams. *IEEE Trans. Reliability*, 56(3):506–515, 2007. doi:10.1109/TR.2007.898572.

8 Hiroshi Imai, Kyoko Sekine, and Keiko Imai. Computational investigations of all-terminal network reliability via BDDs. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E82-A:714–721, 1999.

9 Takeru Inoue, Hiroaki Iwashita, Jun Kawahara, and Shin-ichi Minato. Graphillion: software library for very large sets of labeled graphs. *Int. J. Softw. Tools Technol. Transf.*, 18(1):57–66, 2016. doi:10.1007/s10009-014-0352-z.

10 Takeru Inoue, Keiji Takano, Takayuki Watanabe, Jun Kawahara, Ryo Yoshinaka, Akihiro Kishimoto, Koji Tsuda, Shin-ichi Minato, and Yasuhiro Hayashi. Distribution loss minimization with guaranteed error bound. *IEEE Trans. Smart Grid*, 5(1):102–111, 2014. doi:10.1109/TSG.2013.2288976.

11 Yuma Inoue and Shin-ichi Minato. Acceleration of ZDD construction for subgraph enumeration via path-width optimization. *TCS-TR-A-16-80. Hokkaido University*, 2016.

12 Jun Kawahara, Takashi Horiyama, Keisuke Hotta, and Shin-ichi Minato. Generating all patterns of graph partitions within a disparity bound. In Sheung-Hung Poon, Md. Saidur Rahman, and Hsu-Chun Yen, editors, *WALCOM: Algorithms and Computation, 11th International Conference and Workshops, WALCOM 2017, Hsinchu, Taiwan, March 29-31, 2017, Proceedings*, volume 10167 of *Lecture Notes in Computer Science*, pages 119–131. Springer, 2017. doi:10.1007/978-3-319-53925-6_10.

13 Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato. Frontier-based search for enumerating all constrained subgraphs with compressed representation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E100-A(9):1773–1784, 2017.

14 Donald E. Knuth. *The art of computer programming, Vol. 4A, Combinatorial algorithms, Part 1.* Addison-Wesley Professional, 1st edition, 2011.

15 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.

16 Yu Nakahata, Jun Kawahara, Takashi Horiyama, and Shoji Kasahara. Enumerating all spanning shortest path forests with distance and capacity constraints. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E101-A(9):1363–1374, 2018.

17 Yu Nakahata, Jun Kawahara, and Shoji Kasahara. Enumerating graph partitions without too small connected components using zero-suppressed binary and ternary decision diagrams. In Gianlorenzo D'Angelo, editor, *17th International Symposium on Experimental Algorithms, SEA 2018, June 27-29, 2018, L'Aquila, Italy*, volume 103 of *LIPIcs*, pages 21:1–21:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.SEA.2018.21.

18 Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Zero-suppressed sentential decision diagrams. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 1058–1066. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12434.

19 Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Compiling graph substructures into sentential decision diagrams. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 1213–1221. AAAI Press, 2017. URL: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14919.

20 Kyoko Sekine, Hiroshi Imai, and Seiichiro Tani. Computing the tutte polynomial of a graph of moderate size. In John Staples, Peter Eades, Naoki Katoh, and Alistair Moffat, editors, *Algorithms and Computation, 6th International Symposium, ISAAC '95, Cairns, Australia, December 4-6, 1995, Proceedings*, volume 1004 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 1995. doi:10.1007/BFb0015427.

21 Akiyoshi Shioura, Akihisa Tamura, and Takeaki Uno. An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM J. Comput.*, 26(3):678–692, 1997. doi:10.1137/S0097539794270881.

22 Yexiang Xue, Arthur Choi, and Adnan Darwiche. Basing decisions on sentences in decision diagrams. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. AAAI Press, 2012. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/4977.

Engineering Exact Quasi-Threshold Editing

Lars Gottesbüren
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
lars.gottesbueren@kit.edu

Michael Hamann
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
michael.hamann@kit.edu

Philipp Schoch
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany

Ben Strasser
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
academia@ben-strasser.net

Dorothea Wagner
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
dorothea.wagner@kit.edu

Sven Zühlsdorf
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
zuehlsdorf.kit@ghostdub.de

Abstract

Quasi-threshold graphs are $\{C_4, P_4\}$-free graphs, i.e., they do not contain any cycle or path of four nodes as an induced subgraph. We study the $\{C_4, P_4\}$-free editing problem, which is the problem of finding a minimum number of edge insertions or deletions to transform an input graph into a quasi-threshold graph. This problem is NP-hard but fixed-parameter tractable (FPT) in the number of edits by using a branch-and-bound algorithm and admits a simple integer linear programming formulation (ILP). Both methods are also applicable to the general $\mathcal{F}$-free editing problem for any finite set of graphs $\mathcal{F}$. For the FPT algorithm, we introduce a fast heuristic for computing high-quality lower bounds and an improved branching strategy. For the ILP, we engineer several variants of row generation. We evaluate both methods for quasi-threshold editing on a large set of protein similarity graphs. For most instances, our optimizations speed up the FPT algorithm by one to three orders of magnitude. The running time of the ILP, that we solve using Gurobi, becomes only slightly faster. With all optimizations, the FPT algorithm is slightly faster than the ILP, even when listing all solutions. Additionally, we show that for almost all graphs, solutions of the previously proposed quasi-threshold editing heuristic QTM are close to optimal.

2012 ACM Subject Classification Information systems → Clustering; Theory of computation → Graph algorithms analysis; Theory of computation → Fixed parameter tractability; Theory of computation → Branch-and-bound

Keywords and phrases Edge Editing, Integer Linear Programming, FPT algorithm, Quasi-Threshold Editing

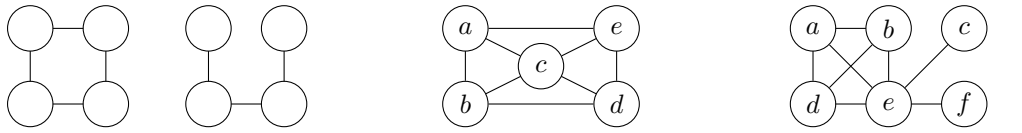
Digital Object Identifier 10.4230/LIPIcs.SEA.2020.10

Related Version A full version of the paper is available at [13], `https://arxiv.org/abs/2003.14317`.

Supplementary Material Implementation: `https://github.com/kit-algo/fpt-editing`

Funding This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grants WA654/19-2 and WA654/22-2. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

Acknowledgements We thank James Nastos and Mark Ortmann for helpful discussions.



(a) A C_4. **(b)** A P_4. **(c)** A graph with node-induced C_4. **(d)** A $\{C_4, P_4\}$-free graph.

Figure 1 A C_4, a P_4 and examples for graphs that are (not) $\{C_4, P_4\}$-free.

1 Introduction

We study graph edge editing problems. The *distance* between two graphs G and H, with the same node set, is the minimum number of edge insertions or deletions needed to transform G into H. Given a graph class $\mathcal{C}$ and a graph G, the editing problem asks for a graph $H \in \mathcal{C}$ closest to G. The corresponding decision problem asks whether k edits are sufficient to transform G into a graph $H \in \mathcal{C}$. We study algorithms to solve editing problems exactly.

A graph H is an *induced subgraph* of a graph G, if there exists an injective mapping π from the nodes of H onto the nodes of G such that there is an edge between two nodes of H, if and only if there is an edge between the corresponding nodes in G. A graph G that does not contain H as induced subgraph is *H-free*. Analogously, for a set of forbidden subgraphs $\mathcal{F}$, a graph G is $\mathcal{F}$-free, if no graph $H \in \mathcal{F}$ is an induced subgraph of G.

We denote by P_ℓ a path graph with ℓ nodes. Similarly, C_ℓ denotes a cycle graph with ℓ nodes. Figures 1a and 1b depict a C_4 and a P_4. The graph depicted in Figure 1c is not $\{C_4, P_4\}$-free as the nodes (a, b, d, e) form an induced C_4. In contrast, the graph depicted in Figure 1d is $\{C_4, P_4\}$-free. The nodes (a, b, e, c) form a P_4, however, as there is an edge between a and e, the subgraph is not an induced subgraph.

While the theoretical part of our study considers any $\mathcal{F}$-free edge editing problem for a finite set of subgraphs $\mathcal{F}$, our experimental study considers $\{C_4, P_4\}$-free graphs. These are also called *quasi-threshold* or *trivially perfect* graphs. Quasi-threshold editing has applications in detecting communities in social friendship networks. Nastos and Gao [23] detect communities in a graph G by computing a closest quasi-threshold graph H of G. Each connected component in H corresponds to a community in G.

For many choices of $\mathcal{F}$, $\mathcal{F}$-free editing is NP-hard, in particular for $\mathcal{F} = \{C_4, P_4\}$ [23]. The $\mathcal{F}$-free edge editing problem is fixed-parameter tractable (FPT) in the number of edits k [7]. This proof directly leads to a branch-and-bound algorithm, see Section 4. For $\{C_4, P_4\}$-free edge editing, it has a running time of $O(6^k \cdot (n+m))$, where n and m are the number of nodes and edges. Unfortunately, social networks typically require a large number of edits [6], which makes plain FPT algorithms impracticable. Therefore, in [23] and [6], quasi-threshold editing heuristics have been introduced for detecting communities in social friendship networks. However, as both approaches are heuristics, they might detect communities that are different from those defined by the model that assumes an optimal solution. Our goal is to improve the running time of exact $\{C_4, P_4\}$-free editing in practice in order to make it feasible at least for small networks. This allows us to study exact solutions of the community detection problem and to verify the quality of heuristics.

1.1 Related Work

For the special case of $\{C_4, P_4\}$-free edge deletion, where only edge deletion operations are allowed, optimized branching rules have been proposed that reduce the running time of the trivial algorithm from $O(4^k \cdot (n + m))$ to $O(2.42^k \cdot (n + m))$ [19]. To the best of our

knowledge, for $\{C_4, P_4\}$-free editing, no improved branching rules have been proposed so far. A polynomial kernel of size $O(k^7)$ for $\{C_4, P_4\}$-free graphs has been proposed [11], which is too large for most practical applications.

A frequently considered problem is $\{P_3\}$-free editing, better known as cluster editing [2]. Early approaches for cluster editing include a linear programming formulation with cutting planes that are incrementally added (in batches of a few hundred constraints) [14]. Later, exact algorithms based on integer linear programming as well as kernelization and more efficient FPT algorithms have been considered [4]. In [16], the authors combine the FPT algorithm with kernelization as well as upper and lower bounds. Editing to $\{P_4\}$-free graphs has been considered in phylogenomics [17] using a simple ILP-based approach.

In a bachelor thesis [5], $\{P_5\}$-free editing has been considered for community detection. They apply lower bounds, data reduction rules and rules for disallowing certain edits.

1.2 Our Contribution

In this paper, we compare two different methods for solving $\mathcal{F}$-free editing problems. The first is a branch-and-bound FPT algorithm while the second is an ILP. For the FPT algorithm, we propose a novel lower bound algorithm based on local search heuristics for independent sets as well as an improved branching strategy. Additionally, we parallelize our implementation. For the ILP, we engineer several variants of row generation. We assess the running time improvements of the different optimizations for quasi-threshold editing on a large benchmark set of 716 graphs that are connected components of a protein similarity graph. This benchmark set has previously been used to evaluate cluster editing algorithms [24, 3]. On 75% of the instances, our improved bounds and optimized branching choices yield speedups of one to three orders of magnitude for the FPT algorithm. For the ILP, we are only able to achieve small speedups. With all optimizations, in the median, the FPT algorithm is twice as fast as the ILP, even when enumerating all possible optimal solutions exactly once. Compared with the parallel execution of Gurobi [15], the FPT algorithm achieves better speedups. Additionally, we evaluate an LP relaxation as lower bound. We prove that its bounds are at least as good as our local search bounds. In our experiments, however, it is too slow to be competitive.

Further, we compare our exact solutions with heuristic solutions found by QTM [6]. It turns out that many heuristic solutions are exact and all but one of them are close to the exact solution. Additionally, we are able to solve four out of the five social networks considered in [23], of which only one was solved previously [21].

1.3 Outline

We start by introducing the preliminaries in Section 2. We describe the ILP formulation and the optimizations we apply to it in Section 3. In Section 4, we then introduce the branch-and-bound FPT algorithm including existing and novel optimizations. In Section 5, we present our experimental setup and evaluation. We conclude in Section 6.

2 Preliminaries

All graphs in this paper are undirected, unweighted, and finite. Further, no graph has self-loops or multi-edges. A graph $G = (V_G, E_G)$ consists of $n := |V_G|$ nodes and $m := |E_G|$ undirected edges. By $\overline{E_G}$, we denote the complement of the edges. In the following, k denotes the maximum number of edits.

3 Integer Linear Programming

In this section, we describe an ILP formulation for $\mathcal{F}$-free editing that is based on an existing formulation for cluster editing [14]. Further, we introduce our optimizations based on row generation and modified constraints to make the ILP practical for small instances.

For every node pair $u, v \in \binom{V_G}{2}$ we introduce a variable $x_{uv} \in \{0, 1\}$ which is 1 if the node pair is an edge in the edited graph and 0 otherwise. We add constraints to ensure that no forbidden subgraph $H \in \mathcal{F}$ can be induced in G via an injective node mapping π:

$$\forall H \in \mathcal{F}, \forall \pi \colon V_H \hookrightarrow V_G : \sum_{\{u,v\} \in E_H} (1 - x_{\pi(u)\pi(v)}) + \sum_{\{u,v\} \in \overline{E_H}} x_{\pi(u)\pi(v)} \geq 1 \tag{1}$$

The objective minimizes the number of edits:

$$\min \sum_{\{u,v\} \in E_G} (1 - x_{uv}) + \sum_{\{u,v\} \in \overline{E_G}} x_{uv} \tag{2}$$

3.1 Row Generation

Generating all of the above-mentioned constraints is infeasible, even for small instances. Row generation (also called lazy constraints) aims to speed up ILP solvers by starting with a small subset of the constraints and subsequently adding constraints that are violated in intermediate solutions. We start with constraints for forbidden subgraphs in the input graph. In our experiments, we consider two options to add constraints violated in an intermediate solution: adding either all violated constraints or only one.

The ILP solver uses LP relaxations to prune its search. These can be strengthened by adding constraints from Equation 1 that are violated by the LP relaxation. We generate constraints in three steps. First, we consider each node pair $\{u, v\}$ for which the relaxation solution has a value different from the input graph. We edit it, then enumerate the forbidden subgraph embeddings containing u and v, add the constraint that is most violated (i.e., whose left side is furthest below 1) and then revert the edit. Ties are broken uniformly at random. Second, we apply the same procedure to the best heuristic solution found so far. Third, we round the LP solution, i.e., an edge exists iff the corresponding variable is greater than 0.5. We then list forbidden subgraph embeddings in this rounded solution and add the corresponding most violated constraint if there is any. The listing skips forbidden subgraphs for which the corresponding constraint has already been added.

3.2 Optimizing Constraints for $\{C_4, P_4\}$-free Editing

If one forbidden subgraph can be transformed into another by a single edit, we can omit a node pair from the constraint for this subgraph. This is similar to the optimization described in Section 4.2. For a P_4, this is the node pair consisting of the two degree-one nodes. For a C_4, we can omit any one of its four edges. We always consider all four possibilities, and in the initial constraint generation as well as the basic row generation variant we add all of them. With this optimization, the constraints for C_4s and P_4s are identical.

We can also formulate a constraint for a C_4 that explicitly models that two deletions or one insertion are required:

$$\forall (u_1, u_2, u_3, u_4) \in V_G^4 : \; 0.5 \cdot \sum_{i=1}^{4} (1 - x_{u_i u_{i+1}}) + x_{u_1 u_3} + x_{u_2 u_4} \geq 1 \tag{3}$$

4 The FPT Branch-and-Bound Algorithm

The FPT algorithm [7] is a branch-and-bound algorithm. For a given maximum number of edits k, it either reports that no solution exists or returns a set of k edits. It works as follows: Find a forbidden subgraph H and branch on all possible edits in H. As H is induced, only edits in H can destroy it and thus one of these edits must be part of the solution. The algorithm is then recursively called for each branch with $k-1$ remaining edits.

Denote by p the maximum number of nodes in a forbidden subgraph. Finding H can be done trivially in time $O(n^p)$ by enumerating all subgraphs of the required size. For specific sets of forbidden subgraphs, such as $\{C_4, P_4\}$, this can be improved to $O(n+m)$ [8, 6].

Every pair of nodes in H is a valid edit. The branching factor is therefore $p \cdot (p-1)/2$. The depth of the recursion is bounded by the maximum number of edits k. The total running time is therefore in $O(p^{2k} \cdot n^p)$ for general families of forbidden subgraphs. For quasi-threshold editing the running time is $O(6^k \cdot (n+m))$. This can be improved to $O(5^k \cdot (n+m))$ by applying the optimization described in Section 4.2.

For finding the minimum number of edits k_{opt}, the algorithm needs to be executed for increasing values of k until a solution is returned. For a branching factor of 2 or larger, the running time of all $k < k_{\text{opt}}$ together is at most the running time for k_{opt}. Thus the total running time is dominated by the running time for k_{opt}.

In the following, we describe several optimizations to reduce the number of explored branches in practice. We describe existing techniques for avoiding redundant exploration of branches (Section 4.1), for skipping certain branches (Section 4.2) as well as lower bounds (Section 4.3). We introduce a novel local search lower bound (Section 4.4), optimized branching choices (Section 4.5), early pruning of branches (Section 4.6) and a simple parallelization (Section 4.7). The appendix of the full version [13] provides in-depth implementation details.

4.1 Avoiding Redundancy

Damaschke [9] proposes to block node pairs to list every solution exactly once. When spawning a search tree node x through editing a node pair, it is neither useful to undo that edit in the sub-search-tree rooted at x, nor is it useful to perform the edit in sibling search trees. While this has been introduced for cluster editing, the technique can be applied to arbitrary $\mathcal{F}$-free editing problems. We explain this technique in detail in the full version [13].

4.2 Skip Forbidden Subgraph Conversion

▶ **Lemma 1.** *If each forbidden subgraph $A \in \mathcal{F}$ can be transformed into another one $B \in \mathcal{F}$ by one edit, the branching factor of the FPT algorithm can be reduced from $\binom{p}{2}$ to $\binom{p}{2} - 1$.*

There is an edit that transforms a P_4 into a C_4. Clearly, this edit can be skipped. Further, there are four edge deletions that transform a C_4 into a P_4. One of these can be skipped [22]. We can choose which one, but as any pair of two edge deletions eliminates the forbidden subgraph, skipping more than one of them might eliminate a necessary branch. Since the branching factor is reduced, this decreases the worst-case running time from $O(6^k \cdot (n+m))$ to $O(5^k \cdot (n+m))$ for quasi-threshold editing.

4.3 Existing Lower Bound Approaches

At each branching node, we have a certain number k of edits left. If we can show that the graph needs at least $k+1$ edits, we do not need to explore further branches below that node. Lower bounds have been used for cluster editing [4, 16] and $\{P_5\}$-free editing [5]. Commonly, they are based on an LP relaxation of the ILP [16], or on a disjoint packing argument [5, 16].

Subgraph Packing. A *node-pair disjoint subgraph packing* P is a set of induced forbidden subgraphs that do not share a node pair. As no edit can eliminate more than one subgraph, $|P|$ is a lower bound on the number of edits required. Taking the previously mentioned optimizations into account, we can include more subgraphs in P by allowing to share blocked node pairs, as they cannot be edited. Further, for each forbidden subgraph a node pair that transforms it into another forbidden subgraph may be shared. In the case of $\mathcal{F} = \{C_4, P_4\}$, the pair of degree-1 nodes of an induced P_4 can be shared. For C_4, we can choose any edge to share, but it remains the same as long as the C_4 is in the packing.

Finding such a packing can be modeled as an independent set problem [16]. The forbidden subgraphs are nodes and every pair of forbidden subgraphs that shares a non-shareable node pair is connected by an edge. A natural greedy heuristic for independent sets is to iteratively add the node that has the smallest degree and then remove all its neighbors from the graph. This can be implemented in linear time by splitting nodes into buckets according to their degree (see e.g. [1]). This heuristic has also been used to calculate lower bounds for cluster editing [16]. We are not aware of complexity results of the independent set problem on this special graph class.

In our experiments, we evaluate three bounds based on subgraph packing: 1) A *basic* bound that iteratively adds subgraphs to the packing as they are found. 2) An incremental version of 1) that *updates* the packing as the graph is modified in the branch-and-bound algorithm. After applying an edit, we remove the subgraph that contains the edited node pair. After both editing and blocking, we enumerate and add subgraphs to the bound until it is maximal. 3) A greedy bound based on the *minimum degree* heuristic. In contrast to the first two, this requires storing all forbidden subgraphs. To avoid this in trivial cases, we first apply 2) to the previous bound and only compute a new packing if this fails to prune the branch.

LP relaxation. The optimal solution of the LP relaxation provided in Section 3 is an upper bound for the node-pair-disjoint packing problem. This can be shown by considering an LP with just the constraints that correspond to the subgraphs in a packing. Each subgraph in the packing is a node-induced subgraph of G. Therefore, the terms on the left side of its corresponding constraint appear in the objective function exactly as they appear in the constraint, confer Equations 1 and 2. Each term in the objective function is at least 0, and each group of terms corresponding to a fulfilled constraint sums to at least 1. Since the packing is node-pair disjoint, the constraints do not share any variables and thus groups do not overlap. Therefore, the objective value is at least the number of subgraphs in the packing. Adding more constraints can only increase the objective and thus improve the bound. We can also model blocked node pairs by replacing the corresponding variable by its value. The variables in the constraints are then disjoint again and thus the same argument applies.

4.4 Local Search Lower Bound

We propose a lower bound based on a subgraph packing that is computed using an adaptation of the 2-improvements local search heuristic [1] for independent sets. Our local search starts with an initial packing and works in rounds. In each round, it iterates over all forbidden subgraphs in the packing and tries to replace one by two forbidden subgraphs. If this is not possible, it tries to replace one by one. Preliminary experiments have shown that choosing this replacement from those candidates which cover the fewest other forbidden subgraphs leads to significantly higher bounds than considering all candidates. We also found that using this strategy only 70% of the time and otherwise choosing a random replacement

is even better. We repeat this procedure until in five consecutive rounds only one-by-one replacements were found. We also terminate the search if the packing remains completely unchanged in a round, or if the packing is large enough to prune the current branch in the search tree. To make this efficient, we approximate the number of forbidden subgraphs that are covered by a certain forbidden subgraph H, by adding up the number of forbidden subgraphs each node pair of H is part of. For the latter we can efficiently maintain counters.

The initial packing is computed with the basic greedy bound. For recursive calls, we update the previous bound as discussed above, before employing local search.

4.5 Branch on Most Useful Node Pairs

We can choose any forbidden subgraph for branching on its possible edits, e.g., the first we find. If there is a forbidden subgraph with only one non-blocked node pair, we choose it, as this will lead to just one recursive call. Otherwise, the first node pair we try to edit should ideally lead to a solution, or blocking the edit should prune the search. We propose to prefer forbidden subgraphs whose non-blocked node pairs are part of many other forbidden subgraphs. Then, a single edit can eliminate many forbidden subgraphs (possibly leading to a solution) and blocking the node pairs allows adding many subgraphs to the lower bound. For each forbidden subgraph, we sort its non-blocked node pairs in decreasing order by the number of forbidden subgraphs that contain the respective node pair. The edits of the selected forbidden subgraph are also tried in this order. We select the subgraph to branch on using a lexicographical ordering on these counts. The last node pair is excluded, as there are no branches left to prune. Additionally, if two subgraphs have identical count sequences (up to the length of the shorter one), we prefer the subgraph with the shorter sequence.

4.6 Prune Branches Early

Normally, we attempt to prune a branch after applying an edit and descending into recursion. With the optimization from Section 4.1, the edited node pair of a recursive call remains blocked after returning from recursion. We update the lower bound to consider this blocked node pair. If the new lower bound already exceeds the remaining number of edits, we can directly prune all subsequent recursive calls, instead of pruning them individually. There are two cases for which we skip the bound update to save running time: If there is only one subsequent recursive call, as we would only prune a single branch, and if the blocked node pair is only part of a single forbidden subgraph, as it cannot yield a better lower bound.

4.7 Parallelization

The algorithm can be parallelized by letting different cores explore different branches. Due to our optimizations, not every branch needs the same running time. Therefore, just executing the first branches in parallel is not scalable. Instead, we use a simple work stealing algorithm. Whenever a thread has fully explored its branch, it steals a branch on the highest available level from another thread and further explores it.

5 Experimental Evaluation

In the appendix of the full version [13] we discuss implementation details. The C++ source code[1] of all discussed variants is available online. We use the C++ interface of Gurobi [15] to solve ILPs and LPs. We evaluate our algorithms on a set of 3964 graphs that are connected components of the COG protein similarity data[2] that has already been used for the evaluation of cluster editing algorithms [24, 3]. The dataset consists of a similarity matrix for each graph. We treat all non-negative scores as edges. Unless stated otherwise, we restrict our evaluation to the 716 graphs that require at least 20 edits. On the 3248 excluded graphs, the maximum running time is less than 0.43 seconds for the FPT algorithm using our local search lower bound. Of these graphs, 1666 require no edits at all. Further, we evaluate our algorithms on a set of 5 small social networks that were already considered by Nastos and Gao [23], namely `karate` [25], `grass_web` [10], `lesmis` [18], `dolphins` [20], and `football` [12].

All experiments were performed on systems with two 8-core Intel Xeon E5-2670 (Sandy Bridge) processors and 64 GB RAM. We set a global time limit of 1000 seconds. Experiments comparing just FPT variants were executed on 16 different node orders, running 16 node orders in parallel. Due to the memory requirements of Gurobi, this is not feasible for the ILP and the LP bound. For these variants, we run just one instance at a time. For experiments involving ILP variants, we also limit the experiments to 4 node orders, and, for better comparability, we run one instance at a time also for the FPT comparison runs in Figure 4. By default, all algorithms terminate at the first found solution, as the ILP is unable to enumerate solutions. Variants with the suffix -All enumerate all solutions. Further, variants with the suffix -MT are parallelized using 16 cores.

5.1 Variants of the FPT Algorithm

The baseline branching strategy -F uses the first found forbidden subgraph. Our `Most` branching strategy from Section 4.5 is denoted by -M, additional early pruning by -MP. The basic greedy bound is denoted by -G, the incremental update bound by -U, the min-degree heuristic by -MD, our local search lower bound by -LS, and LP relaxations by -LP. The comparison includes the nine variants FPT-G-F-All, FPT-G-MP-All, FPT-U-MP-All, FPT-MD-F-All, FPT-MD-MP-All, FPT-LP-MP-All, FPT-LS-F-All, FPT-LS-M-All and FPT-LS-MP-All.

Figure 2 shows how many of the COG dataset instances can be solved within a certain time limit and with a certain number of recursive calls – added over all k's. Additional lower bound calls due to -MP count extra. An instance is a single node id permutation of a graph, i.e., every graph is counted 16 times. Of the 716 graphs we are able to solve 547 within the 1000 second time limit. Below, we also compare calls and running times per instances.

For comparing branching strategies, we fix the local search algorithm -LS as the lower bound. The median factor of additional calls needed by -M over -MP is 1.9 and by -F over -MP is 3.36, restricted to instances solved by both algorithms. While the median speedup of -MP over -F is 3.11, it is just 1.06 over -M. On 5% of the instances, the speedup is at least 56.62 and 1.24, respectively. This shows that for -M the improvement in the number of calls directly leads to similar running time improvements, while early pruning just reduces calls.

For comparing lower bound algorithms, we fix -MP as the branching strategy. There is an inherent trade-off between the number of recursive calls and the time spent per call, with

[1] `https://github.com/kit-algo/fpt-editing`
[2] `https://bio.informatik.uni-jena.de/data/#cluster_editing_data`

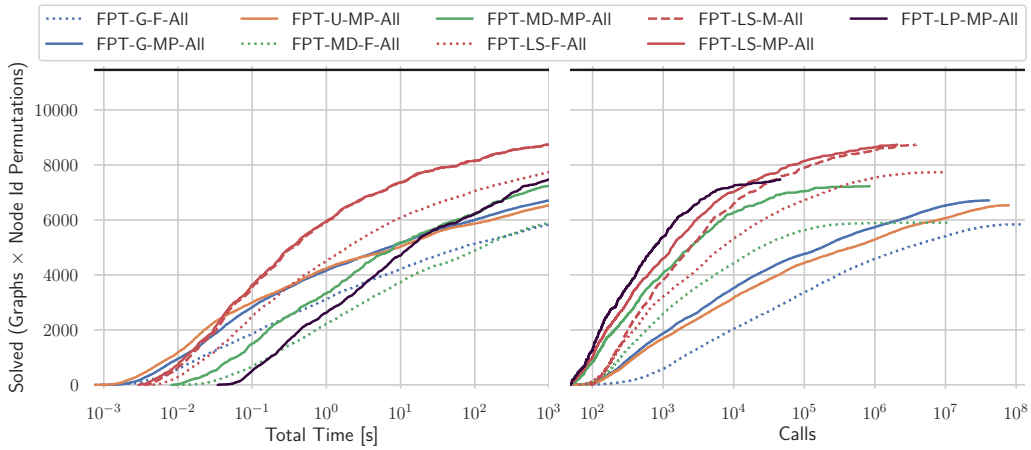


Figure 2 Number of permutations of graphs of the COG dataset that require at least 20 edits and can be solved within a certain total running time / with a certain number of recursive calls (and extra lower bound updates for -MP). The horizontal black line indicates the total number of graphs and node permutations that require 20 or more edits, including unsolved instances.

a sweet spot that gives the best overall running time. The basic greedy bounds need 10 to 24 times as many calls as the other bounds in the median. The recomputed greedy bound -G is slightly better than the updated one -U, -LP is the best, followed by -LS and -MD.

Nonetheless, for very small time limits, -U solves the highest number of instances. For larger time limits, reducing the number of calls pays off, though not at any cost. The median speedup of min-degree over the LP is 2.16 while needing 47% more calls in the median. Local search avoids their substantial memory overhead and spends significantly less time per call than both. It needs 83% of the calls of -MD while being a factor of 12.36 faster in the median. It is never slower, and on 5% of the instances even more than 137 times faster than -MD.

Comparing the state-of-the-art FPT-MD-F-All algorithm to our FPT-LS-MP-All algorithm, we need 4.33 times less calls and are 46.06 times faster in the median. We are never slower, on 75% of the instances more than 16 times and on 5% of the instances more than 1044 times faster. In conclusion, our local search lower bound gives high-quality bounds while being fast. Our branching rules reduce the running time by another small factor while early pruning mainly reduces the number of calls. Overall, we achieve a speedup of one to three orders of magnitude over the state-of-the-art.

5.2 Parallelization

The left part of Figure 3 reports the speedup of FPT-LS-MP-All-MT over its sequential counterpart FPT-LS-MP-All, the sequentially fastest variant on the COG dataset. We show the speedup with 1, 2, 4, 8 and 16 cores in comparison to the number of recursive calls and lower bound calculations. For each graph and permutation, we plot the speedup on the last value of k for which the sequential version of the algorithm terminated within the time limit. With only few recursive calls we cannot expect a good speedup. For a high number of recursive calls, FPT-LS-MP-All-MT achieves almost perfect speedup for all numbers of cores on many graphs. As the algorithm is executed with increasing values of k, for some graphs only the last value of k needs a high number of calls and thus the overall speedup is not perfect even though in sum the number of calls is high.

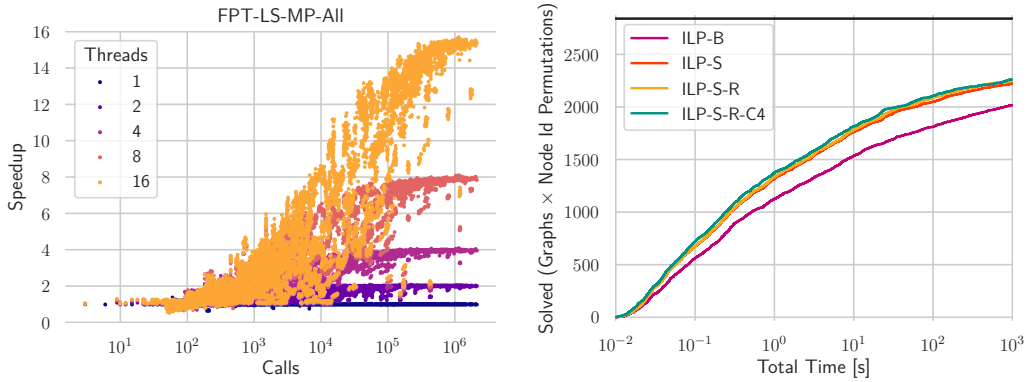


Figure 3 Speedup of FPT-LS-M-All-MT and comparison of the different ILP variants on 16 (left) and 4 (right) node id permutations of the 716 COG graphs that require at least 20 edits.

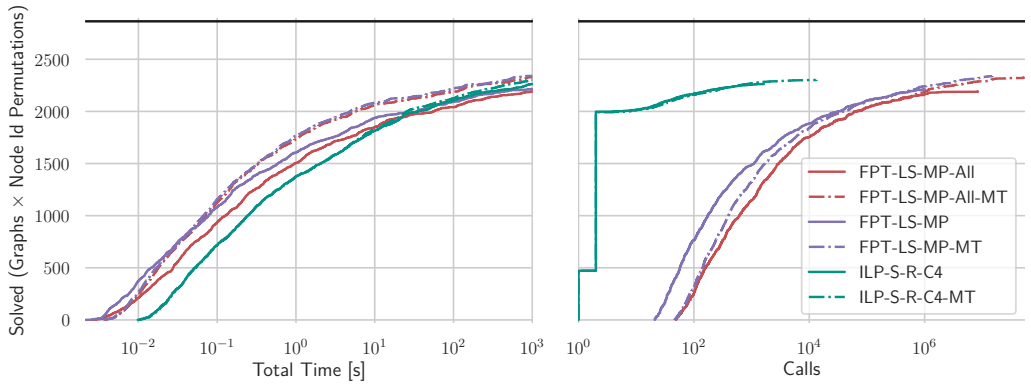


Figure 4 Comparison of the ILP to the FPT algorithm on 4 node id permutations of the 716 COG graphs that require at least 20 edits.

5.3 Variants of the ILP

Figure 3 (right) shows the impact of the different optimizations on the ILP, when enabled one after another. We denote adding only one violated constraint by -S, adding constraints during relaxations by -R, and specialized C_4 constraints by -C4. The baseline is ILP-B, where row generation always adds all violated constraints for intermediate solutions.

In the median, ILP-S is just 5% faster than ILP-B. While on 95% of the instances it is at most 20% slower, it is more than 44 times faster on 5% of them, which explains the gap in Figure 3. ILP-S-R is not faster in the median, but on 95% of the instances at most 12% slower and on 5% it is at least 73% faster. The C_4 constraints make the ILP 12% faster in the median, at most 26% slower on 95% and at least 95% faster on 5% of the instances. With all optimizations, the ILP solves 568 graphs. We also tried providing a heuristic solution from QTM [6] to Gurobi, but the improvement was even smaller and disappeared in parallel.

Figure 4 compares the best ILP and FPT algorithms with and without -MT in terms of running time and recursive calls. For the FPT algorithm, stopping at the first solution is not slower on 95%, more than 52% faster on 50% and more than 3 times faster on 5% of the instances. Multi-threading incurs a measurable overhead. Compared to FPT-LS-MP-All,

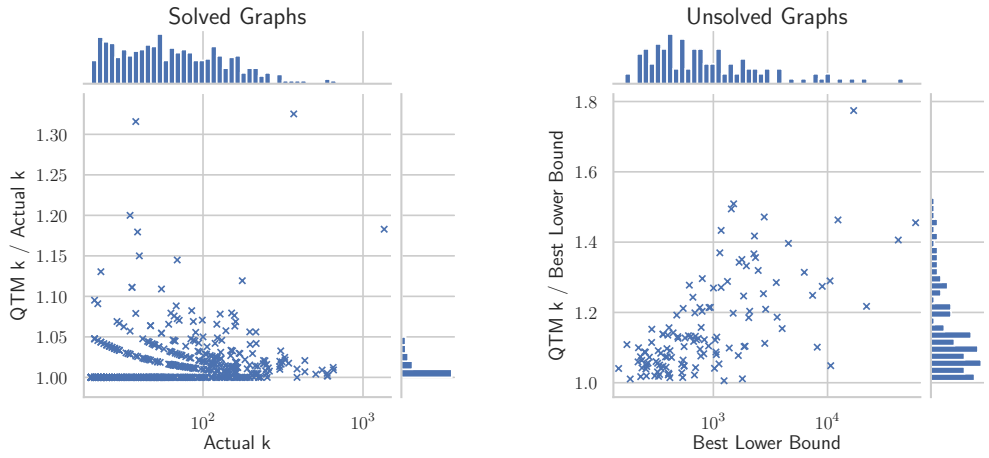


Figure 5 Comparison of heuristic solutions of QTM and the exact number of edits k for solved graphs (left) or the best lower bound for unsolved graphs (right) achieved by the FPT algorithm or the ILP. For readability, we exclude one solved graph at $k = 64$, where QTM needed 202 edits.

FPT-LS-MP-All-MT is at most 16% slower on 95%, 78% faster in the median and more than 12 times faster on 5% of the instances. When stopping at the first solution, this decreases to 24% slower, 1% faster and 10 times faster, as more branches that do not lead to a solution are explored in multi-threaded mode. FPT-LS-MP-MT is still 4% faster than FPT-LS-MP-All-MT in the median, at most 3% slower on 95% and at least 68% faster on 5% of the instances.

The parallel ILP is at most 5% slower on 95%, as fast in the median and more than 52% faster on 5% of the instances than the sequential ILP. Thus, the parallelization helps the FPT algorithm more than the ILP. A likely cause is that Gurobi needs much less search nodes than the FPT algorithm which offer less potential for parallelism – on 50% of the instances at least 185 times less, and on many graphs even just one or two, see Figure 4.

The speedup of FPT-LS-MP over ILP-S-R-C4 is at least 0.59 on 95%, 3.25 in the median and at least 10.72 on 5% of the instances. For FPT-LS-MP-All, this decreases to 0.29, 2.10 and 7.02. In parallel, the speedups are 1.09, 3.41 and 16.45 for all solutions, and 1.34, 3.67 and 18.14 for the first solution. Single-threaded, the ILP solves more instances within 1000 seconds than the FPT algorithm, indicating that for difficult instances better bounds are more important. Overall, the FPT algorithm is often faster than the ILP, in particular in parallel and even when listing all solutions.

5.4 Comparison to QTM

Figure 5 compares the results of the heuristic Quasi-Threshold Mover (QTM) [6] with exact results for solved and the best lower bounds for unsolved graphs. We use the maximum value of k achieved for any permutation by FPT-LS-MP-MT and by ILP-S-R-C4 with and without -MT. If any of the algorithms solved the graph, we list it in the left part, otherwise in the right part. For QTM, we report the minimum k that QTM found over 16 runs. Again, the plot excludes 3248 graphs that require less than 20 edits. Of those, QTM solved 3172 exactly, 56 with offset 1, 15 with offset 2 and 5 with offset 3. Of the remaining graphs, 588 are solved and 128 are unsolved. Of the solved graphs, QTM solved 319 graphs exactly. For none of the unsolved graphs, QTM matches the lower bound. For 95% of the 716 graphs, QTM needs at most 1.22 times the edits of the exact solution or the lower bound.

Table 1 Overview of the social network graphs. Using the algorithms FPT-LS-MP and ILP-S-R-C4 with 1 and 16 cores, we report the maximum k that finished within 1000 seconds, and the minimum time over all permutations that is needed to find the first solution. In the case of `football`, we report the time needed to show that there is no solution with that k.

			FPT				ILP			
			1 core		16 cores		1 core		16 cores	
Graph	n	m	k	Time [s]	k	Time [s]	k	Time [s]	k	Time [s]
karate	34	78	21	0.01	21	0.01	21	0.02	21	0.03
lesmis	77	254	60	0.17	60	0.13	60	0.96	60	0.97
grass_web	75	113	34	1.81	34	0.21	34	2.91	34	2.83
dolphins	62	159	70	126.54	70	18.57	70	23.81	70	12.10
football	115	613	223	929.55	228	649.94	235	1000.01	237	1000.05

5.5 Social Network Instances

Table 1 shows an overview of the social networks with results for FPT-LS-MP and ILP-S-R-C4. Both solve `karate` and `lesmis` in less than a second, and `grass_web` within 3 seconds, with the FPT algorithm being faster. Even though `lesmis` is both larger than `grass_web` and requires 60 edits instead of 34, both algorithms are significantly slower on `grass_web`. This shows that their performance depends on the specific structure of the graph and not just the graph size and k. For `dolphins`, the ILP is faster than the FPT algorithm. For all graphs, the FPT algorithm scales better with the number of cores. None of the algorithms can solve the `football` network. We show that there is no solution for $k \leq 223$, $k \leq 228$ using the FPT algorithm with 1 or 16 cores respectively, and $k \leq 235$, $k \leq 237$ using the ILP with 1 or 16 cores respectively. The previously best known upper bound was 251, computed with QTM [6] in 2.5ms. In 1000 seconds, the ILP shows a new upper bound of 250. For the smallest three social networks, we verify that the best heuristic solutions in [6] are exact. QTM needs 72 edits on `dolphins`, whereas 70 edits are optimal. The appendix of the full version [13] contains a detailed analysis of the solution space with a focus on the community detection application.

6 Conclusion

We have introduced optimizations for two different approaches to solving any $\mathcal{F}$-free edge editing problem. We evaluate our optimizations for the special case of quasi-threshold editing on a set of 716 protein interaction graphs. For the first approach, the FPT algorithm, we show that the combination of good lower bounds with careful selection of branches allows to reduce the running time by one to three orders of magnitude for 75% of the instances. For the second approach, an ILP, we evaluate several variants of row generation and show that they achieve small speedups. We show that the FPT algorithm is slightly faster than the ILP, with a larger margin in parallel, and it can easily enumerate all optimal solutions. For the heuristic editing algorithm QTM, we show that on 95% of the instances, it needs at most 22% more edits than our exact solutions or lower bounds indicate.

Comparing the structure of exact vs. heuristic solutions might give further insights how to improve heuristics. Exact FPT algorithms could be further improved by better bounds, possibly based on LP relaxations. As the COG benchmark set actually contains edit costs, an extension of our optimizations to the weighted editing problem could be investigated.

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 Sebastian Böcker and Jan Baumbach. Cluster Editing. In *Proceedings of the 9th Conference on Computability in Europe (CiE'13)*, volume 7921 of *Lecture Notes in Computer Science*, pages 33–44. Springer, 2013. doi:10.1007/978-3-642-39053-1_5.

3 Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truß. A fixed-parameter approach for Weighted Cluster Editing. In *Proceedings of the 6th Asia-Pacific Bioinformatics Conference (APBC 2008)*, volume 6, pages 211–220, 2008. doi:10.1142/9781848161092_0023.

4 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.

5 Felix Bohlmann. Graphclustern durch Zerstören langer induzierter Pfade. Bachelor thesis, TU Berlin, 2015. URL: http://fpt.akt.tu-berlin.de/publications/theses/BA-felix-bohlmann.pdf.

6 Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. Fast Quasi-Threshold Editing. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, volume 9294 of *Lecture Notes in Computer Science*, pages 251–262. Springer, 2015. doi:10.1007/978-3-662-48350-3_22.

7 Leizhen Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58(4):171–176, May 1996. doi:10.1016/0020-0190(96)00050-6.

8 Frank Pok Man Chu. A simple linear time certifying LBFS-based algorithm for recognizing trivially perfect graphs and their complements. *Information Processing Letters*, 107(1):7–12, June 2008. doi:10.1016/j.ipl.2007.12.009.

9 Peter Damaschke. Fixed-Parameter Enumerability of Cluster Editing and Related Problems. *Theory of Computing Systems*, 46(2):261–283, 2008. doi:10.1007/s00224-008-9130-1.

10 Hassan Ali Dawah, Bradford A. Hawkins, and Michael F. Claridge. Structure of the Parasitoid Communities of Grass-Feeding Chalcid Wasps. *Journal of Animal Ecology*, 64(6):708–720, 1995. doi:10.2307/5850.

11 Pål Grønås Drange and Michał Pilipczuk. A Polynomial Kernel for Trivially Perfect Editing. *Algorithmica*, 80(12):3481–3524, December 2017. doi:10.1007/s00453-017-0401-6.

12 Michelle Girvan and Mark E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Science of the United States of America*, 99(12):7821–7826, 2002. doi:10.1073/pnas.122653799.

13 Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. Engineering exact quasi-threshold editing. *arXiv e-prints*, 2020. arXiv:2003.14317.

14 Martin Grötschel and Yoshiko Wakabayashi. A cutting plane algorithm for a clustering problem. *Mathematical Programming*, 45(1-3):59–96, 1989. doi:10.1007/BF01589097.

15 Gurobi Optimization, LLC. Gurobi optimizer reference manual, 2020. URL: http://www.gurobi.com.

16 Sepp Hartung and Holger H. Hoos. Programming by Optimisation Meets Parameterised Algorithmics: A Case Study for Cluster Editing. In *Proceedings of the 9th International Conference on Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 43–58. Springer, 2015. doi:10.1007/978-3-319-19084-6_5.

17 Marc Hellmuth, Nicolas Wieseke, Marcus Lechner, Hans-Peter Lenhof, Martin Middendorf, and Peter F. Stadler. Phylogenomics with paralogs. *Proceedings of the National Academy of Science of the United States of America*, 112(7):2058–2063, 2015. doi:10.1073/pnas.1412770112.

18 Donald E. Knuth. *The Stanford GraphBase : a platform for combinatorial computing*. Addison-Wesley, 1993.

19 Yunlong Liu, Jianxin Wang, Jie You, Jianer Chen, and Yixin Cao. Edge deletion problems: Branching facilitated by modular decomposition. *Theoretical Computer Science*, 573:63–70, 2015. doi:10.1016/j.tcs.2015.01.049.

20 David Lusseau, Karsten Schneider, Oliver Boisseau, Patti Haase, Elisabeth Slooten, and Steve Dawson. The Bottlenose Dolphin Community of Doubtful Sound Features a Large Proportion of Long-Lasting Associations. *Behavioral Ecology and Sociobiology*, 54(4):396–405, September 2004. doi:10.1007/s00265-003-0651-y.

21 James Nastos. *Utilizing graph classes for community detection in social and complex networks*. PhD thesis, University of British Columbia, 2015. doi:10.14288/1.0074429.

22 James Nastos and Yong Gao. A Novel Branching Strategy for Parameterized Graph Modification Problems. In *Proceedings of the 4th International Conference on Combinatorial Optimization and Applications*, volume 2 of *Lecture Notes in Computer Science*, pages 332–346. Springer, 2010. doi:10.1007/978-3-642-17461-2_27.

23 James Nastos and Yong Gao. Familial groups in social networks. *Social Networks*, 35(3):439–450, July 2013. doi:10.1016/j.socnet.2013.05.001.

24 Sven Rahmann, Tobias Wittkop, Jan Baumbach, Marcel Martin, Anke Truß, and Sebastian Böcker. Exact and Heuristic Algorithms for Weighted Cluster Editing. In *Proceedings of the 6th Annual International Conference on Computational Systems Bioinformatics (CSB 2007)*, volume 6, pages 391–401, 2007. doi:10.1142/9781860948732_0040.

25 Wayne W. Zachary. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 33:452–473, 1977. doi:10.1086/jar.33.4.3629752.

Advanced Flow-Based Multilevel Hypergraph Partitioning

Lars Gottesbüren
Karlsruhe Institute of Technology, Germany
lars.gottesbueren@kit.edu

Michael Hamann
Karlsruhe Institute of Technology, Germany
michael.hamann@kit.edu

Sebastian Schlag
Karlsruhe Institute of Technology, Germany
sebastian.schlag@kit.edu

Dorothea Wagner
Karlsruhe Institute of Technology, Germany
dorothea.wagner@kit.edu

Abstract

The balanced hypergraph partitioning problem is to partition a hypergraph into k disjoint blocks of bounded size such that the sum of the number of blocks connected by each hyperedge is minimized. We present an improvement to the flow-based refinement framework of KaHyPar-MF, the current state-of-the-art multilevel k-way hypergraph partitioning algorithm for high-quality solutions. Our improvement is based on the recently proposed HyperFlowCutter algorithm for computing bipartitions of unweighted hypergraphs by solving a sequence of incremental maximum flow problems. Since vertices and hyperedges are aggregated during the coarsening phase, refinement algorithms employed in the multilevel setting must be able to handle both weighted hyperedges and weighted vertices – even if the initial input hypergraph is unweighted. We therefore enhance HyperFlowCutter to handle weighted instances and propose a technique for computing maximum flows directly on weighted hypergraphs.

We compare the performance of two configurations of our new algorithm with KaHyPar-MF and seven other partitioning algorithms on a comprehensive benchmark set with instances from application areas such as VLSI design, scientific computing, and SAT solving. Our first configuration, KaHyPar-HFC, computes slightly better solutions than KaHyPar-MF using significantly less running time. The second configuration, KaHyPar-HFC*, computes solutions of significantly better quality *and* is still slightly faster than KaHyPar-MF. Furthermore, in terms of solution quality, both configurations also outperform all other competing partitioners.

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

Keywords and phrases Hypergraph Partitioning, Maximum Flows, Refinement

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.11

Related Version `https://arxiv.org/abs/2003.12110`

Supplementary Material Source code `https://github.com/larsgottesbueren/WHFC`

Funding This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grants WA654/19-2, WA654/22-2, and SA933/11-1. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

1 Introduction

Graphs are a common way to model pairwise relationships (edges) between objects (vertices). However, many real-world problems involve more complex interactions [7, 29]. Hypergraphs are a generalization of graphs, where each hyperedge can connect an arbitrary number of vertices. Thus, hypergraphs are well-suited to model such higher-order relationships.

The balanced k-way hypergraph partitioning problem (HGP) asks to compute a partition of the vertices into k disjoint blocks of bounded weight (at most $(1+\varepsilon)$ times the average block weight) such that few hyperedges are cut, i.e., connect vertices in different blocks [7, 5, 35]. Since hyperedges can connect more than two vertices, several notions of cuts exit in the literature [5]. In this work, we consider the *connectivity* objective, which aims to minimize $\sum_{e \in E} \omega(e)(\lambda(e) - 1)$, where E is the set of hyperedges, $\omega(e)$ denotes the weight of hyperedge e, and $\lambda(e)$ denotes the number of blocks connected by hyperedge e. Well-known applications of hypergraph partitioning include VLSI design [5], the parallelization of sparse matrix-vector multiplications [9], and storage sharding in distributed databases [25]. We refer to a survey chapter [39, Ch. 3] and two survey articles [5, 35] for an extensive overview.

Since HGP is NP-hard [31], heuristic algorithms are used in practice. The most successful heuristic for computing high-quality solutions is the three-phase *multilevel* paradigm. In the *coarsening phase*, multilevel algorithms first successively contract the input hypergraph to obtain a hierarchy of smaller, structurally similar instances. After applying an *initial partitioning* algorithm to the smallest hypergraph, the contractions are undone and, at each level, refinement algorithms are used to improve the partition induced by the coarser level.

Related Work. The most well-known and practically relevant multilevel hypergraph partitioners from different application areas are PaToH [9] (scientific computing), hMetis [26, 27] (VLSI design), KaHyPar [24, 23, 2, 40] (general purpose, n-level), Zoltan [13, 41] (scientific computing, parallel), Mondriaan [45] (sparse matrices), and Par$k$way [42] (parallel). Additionally there are MLPart [4] (restricted to bipartitioning), and HYPE [33], a single-level algorithm that grows k blocks using a neighborhood expansion [46] heuristic.

With the exception of KaHyPar-MF [24], all multilevel HGP algorithms solely employ variations of Kernighan-Lin [28] (KL) or Fiduccia-Mattheyses [17] (FM) heuristics in the refinement phase. These algorithms repeatedly move vertices between blocks prioritized by the improvement in the objective function. While they perform well for hypergraphs with small hyperedges, their performance deteriorates in the presence of many large hyperedges [44]. In this case, many single-vertex moves have no immediate effect on the objective function because the vertices of large hyperedges are likely to be distributed over multiple blocks. KaHyPar-MF [24] therefore additionally employs a refinement algorithm based on maximum-flow computations between pairs of blocks. Since flow algorithms find minimum s-t cuts, this refinement technique does not suffer the drawbacks of move-based approaches.

The flow-based refinement framework is a generalization of the approach used in the graph partitioner KaFFPa [38]. To refine a pair of blocks of a k-way partition, KaHyPar first extracts a subhypergraph induced by a set of vertices around the cut of these blocks. This subhypergraph is then transformed into a graph-based flow network, using techniques due to Lawler [30], and Liu and Wong [32], on which a maximum flow is computed. The vertices of the subhypergraph are reassigned according to the corresponding minimum cut. The size of the subhypergraph (and thus the size of the flow network) is chosen adaptively, depending on the outcome of the previous refinement. While larger flow networks may produce better but potentially imbalanced solutions, the smallest flow network guarantees a balanced partition. We further discuss KaHyPar and its flow-based refinement framework in Section 2.

HyperFlowCutter (HFC) [22] computes bipartitions of unweighted hypergraphs by solving a sequence of incremental maximum flow problems. Its advantage over computing a single maximum flow is that it does not reject almost balanced solutions, but systematically trades cut-size for increased balance. ReBaHFC [22] uses HFC as postprocessing to improve an initial bipartition computed with PaToH. HFC computes unit-capacity flows directly on the hypergraph (i.e., without using a graph-based flow network) using a technique of Pistorius and Minoux [37] extended to Dinic's flow algorithm [14].

Contribution and Outline. Multilevel refinement algorithms must be able to handle both weighted hyperedges and weighted vertices because vertices and hyperedges are aggregated during the coarsening phase. In this work, we improve KaHyPar's flow-based refinement framework – with regard to both running time and solution quality – by using a *weighted* version of HFC instead of the maximum flow computations on differently-sized graph-based flow networks. After introducing notation and briefly presenting additional details about KaHyPar and HFC refinement in Section 2, we discuss how HFC can simulate an approach of KaHyPar and KaFFPa for balancing partitions, extend HFC's existing balancing approach to weighted instances, and propose a heuristic for guiding its incremental maximum flow problems in Section 3. In Section 4, we present our approach for computing maximum flows on weighted hypergraphs, generalizing the technique of Pistorius and Minoux [37] to weighted hypergraphs and arbitrary flow algorithms.

In our experiments (Section 5), we compare two configurations of our new approach with KaHyPar-MF, and seven other partitioning algorithms on a large benchmark set containing hypergraphs from the VLSI, SAT solving, and scientific computing community [23]. While our first configuration, KaHyPar-HFC, computes slightly better solutions than KaHyPar-MF using significantly less running time, the second configuration, KaHyPar-HFC*, computes solutions of significantly better quality and is still slightly faster than KaHyPar-MF. Furthermore, in terms of solution quality, both configurations outperform all other competing partitioners. We conclude the paper in Section 6 and suggest future work.

2 Preliminaries

Hypergraphs. A *hypergraph* $H = (V, E)$ consists of a set of vertices V and a set of hyperedges E, where a hyperedge e is a subset of the vertices V. Additionally, we associate weights $\omega\colon E \to \mathbb{N}^+$, $\varphi\colon V \to \mathbb{N}^+$ with the hyperedges and vertices. A vertex $v \in V$ is *incident* to hyperedge $e \in E$ if $v \in e$. The vertices incident to a hyperedge e are called the *pins* of e. We denote the pin u in hyperedge e as (u, e). By $H[V'] = (V', \{e \cap V' \mid e \in E\})$, we denote the hypergraph induced by the vertex set V'. The *star expansion* of H represents the hypergraph as a bipartite graph $G = (V \dot\cup E, \{(v, e) \in V \times E \mid v \in e\})$ with bipartite node set $V \dot\cup E$ and an edge for every pin. To avoid confusion, we use the terms vertices, hyperedges, and pins for hypergraphs, and the terms nodes and edges for graphs. We extend functions to sets using $f(X) = \sum_{x \in X} f(x)$ for some function f.

Hypergraph Partitioning. A k-way partition $\pi(H)$ of a hypergraph $H = (V, E)$ is a partition of its vertices into non-empty disjoint *blocks* $V_1, \ldots, V_k \subset V$, i. e., $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset$ for $i = 1, \ldots, k$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. For some parameter $\varepsilon \in [0, 1)$ we call $\pi(H)$ ε-balanced, if each block V_i satisfies the *balance constraint* $\varphi(V_i) \leq (1 + \varepsilon)\frac{\varphi(V)}{k}$. Let $\Lambda(e) = \{V_i \in \pi(H) \mid V_i \cap e \neq \emptyset\}$ denote the blocks that are connected by hyperedge $e \in E$. The *connectivity* of a hyperedge e is defined as $\lambda(e) := |\Lambda(e)|$. Given parameters ε and k, and an input hypergraph H, the balanced k-way hypergraph partitioning problem asks for an ε-balanced k-way partition of H that minimizes the *connectivity-metric* $\sum_{e \in E} \omega(e)(\lambda(e) - 1)$.

Maximum Flows. A flow network is a symmetric, directed graph $\mathcal{N} = (\mathcal{V}, \mathcal{E})$ with two disjoint non-empty *terminal* node sets $S, T \subsetneq \mathcal{V}$, the source and target node set, as well as a capacity function $c\colon \mathcal{E} \to \mathbb{N}_0$. A flow is a function $f\colon \mathcal{E} \to \mathbb{Z}$ subject to the *capacity constraint* $f(e) \leq c(e)$ for all edges e, *flow conservation* $\sum_{(u,v)\in\mathcal{E}} f(u,v) = 0$ for all non-terminal nodes v, and *skew symmetry* $f(u,v) = -f(v,u)$ for all edges (u,v). The *value* of a flow $|f| := \sum_{s\in S,(s,u)\in\mathcal{E}} f(s,u)$ is the amount of flow leaving S. The *residual capacity* $r_f(e) := c(e) - f(e)$ is the additional amount of flow that can pass through e without violating the capacity constraint. The residual network with respect to f is the directed graph $\mathcal{N}_f = (\mathcal{V}, \mathcal{E}_f)$, where $\mathcal{E}_f := \{e \in \mathcal{E} | r_f(e) > 0\}$. A node v is *source-reachable* if there is a path from S to v in $\mathcal{N}_f$, it is *target-reachable* if there is a path from v to T in $\mathcal{N}_f$. We denote the source-reachable and target-reachable nodes by S_r and T_r, respectively. An *augmenting path* is an S-T path in $\mathcal{N}_f$. The flow f is a *maximum flow* if $|f|$ is maximal of all possible flows in $\mathcal{N}$. This is the case if and only if there is no augmenting path in $\mathcal{N}_f$. An S-T edge cut is a set of edges whose removal disconnects S and T. The value of a maximum flow equals the weight of a minimum-weight S-T edge cut [18]. The *source-side cut* consists of the edges from S_r to $\mathcal{V} \setminus S_r$ and the *target-side cut* consists of the edges from T_r to $\mathcal{V} \setminus T_r$. The bipartition $(S_r, \mathcal{V} \setminus S_r)$ is induced by the source-side cut and $(\mathcal{V} \setminus T_r, T_r)$ is induced by the target-side cut. Note that $\mathcal{V} \setminus S_r \setminus T_r$ is not necessarily empty. We also call S_r and T_r the *cutsides* of a maximum flow.

Maximum Flows on Hypergraphs. Lawler [30] uses maximum flows to compute minimum S-T hyperedge cuts. On the star expansion, the construction to model node capacities as edge capacities [1] is applied to the hyperedge-nodes. A hyperedge e is expanded into an *in-node* e_i and an *out-node* e_o joined by a directed *bridge edge* (e_i, e_o) with capacity $c(e_i, e_o) = \omega(e)$. For every pin $u \in e$ there are two directed *external edges* $(u, e_i), (e_o, u)$ with infinite capacity. The transformed graph is called the *Lawler network*. A minimum S-T edge cut in the Lawler network consists only of bridge edges, which directly correspond to S-T cut hyperedges in H. Via the Lawler network, the above notions translate naturally from graphs to hypergraphs, and we use the same terminology and notation for hypergraphs.

The KaHyPar Framework. Since our algorithm is integrated into the KaHyPar framework, we briefly review its core components and outline its k-way flow-based refinement. In contrast to traditional multilevel HGP algorithms that contract matchings or clusterings and therefore work with a coarsening hierarchy of $O(\log n)$ levels, KaHyPar removes only a single vertex between two levels, resulting in almost n levels. Coarsening is restricted to clusters found with a community detection algorithm [23]. Initial partitions of the coarsest hypergraph are computed using a portfolio of simple algorithms [40]. During uncoarsening, it employs a combination of localized FM local search [2, 40] and flow-based refinement [24].

Given a k-way partition $\pi(k) = \{V_1, ..., V_k\}$ of a hypergraph $H = (V, E, \varphi, \omega)$, the flow-based refinement works on pairs (V_i, V_j) of blocks that share cut hyperedges. The blocks are scheduled for refinement as long as this improves the solution (better connectivity, or equal connectivity and less imbalance). To construct a flow problem for a pair of blocks, the algorithm performs two randomized weight-constrained breadth-first searches (BFS) restricted to $H[V_i]$ and $H[V_j]$, respectively. The BFSs are initialized with the vertices of V_i (resp. V_j) that are incident to hyperedges in the cut between V_i and V_j. The first BFS stops if the weight of the visited vertices would exceed $(1 + \alpha \cdot \varepsilon)\lceil \frac{\varphi(V)}{k} \rceil - \varphi(V_j)$, where the scaling parameter α is used to control size of the flow problem. The second BFS visits the vertices of V_j analogously. The hypergraph induced by the visited vertices is then used to

(a) Compute minimum S-T cuts. **(b)** Add S_r to S and choose a piercing vertex.

Figure 1 Flow augmentation and computing S_r, T_r in Fig.1a; adding S_r to S and piercing the source-side cut in Fig.1b. S in blue, $S_r \setminus S$ in yellow, T in green, $T_r \setminus T$ in red, $V \setminus S_r, \setminus T_r$ in white. Taken from Gottesbüren et al. [22] with minor adaptations.

build the Lawler network, with size optimizations due to Liu and Wong [32] and Heuer [24]. A minimum S-T cut in the Lawler network induces a bipartition of $H[V_i \cup V_j]$. If the flow computation resulted in an ε-balanced partition, the improved solution is accepted and α is increased to $\min(2\alpha, \alpha')$ for a predefined upper bound of $\alpha' = 16$. Otherwise, α is decreased to $\max(\alpha/2, 1)$. This scaling scheme continues until a maximal number of rounds is reached or a feasible partition that did not improve the cut is found. KaHyPar runs flow-based refinement on exponentially spaced levels, i. e., after 2^i uncontractions for increasing i, since flow-based refinement is too expensive to be run after every uncontraction.

ReBaHFC. ReBaHFC avoids the need for Lawler networks and the corresponding size optimizations [32, 24] by directly constructing a *flow hypergraph* by contracting vertices not visited by a BFS to S or T. Furthermore, it does not require adaptive rescaling because HFC guarantees balanced partitions.

3 Weighted HyperFlowCutter

To keep this paper self-contained, we briefly explain the core HFC algorithm in Section 3.1. Subsequently, we discuss further details of multilevel refinement with HFC in Section 3.2 and our approach for adapting two balancing strategies in Section 3.3.

3.1 The Core Algorithm

The core HFC algorithm computes bipartitions with monotonically increasing cut size and balance by solving a sequence of incremental maximum flow problems, until both blocks satisfy the balance constraint. Given some initial terminal vertex sets S and T, HFC first computes a maximum flow and derives the cutsides S_r and T_r. If either the source-side cut or the target-side cut induces blocks that fulfill the balance constraint, the algorithm terminates. Otherwise, it adds all vertices on the smaller cutside to its corresponding terminal side, i. e., $S := S_r$ if $|S_r| \leq |T_r|$ and $T := T_r$ otherwise. Then, one additional vertex – the *piercing vertex* – is added to the terminal side and the previous flow is augmented to respect the new terminals. This ensures that HFC finds a different cut with each new maximum flow. By growing the smaller side, the algorithm ensures that it finds a balanced partition after at most $|V|$ iterations. Figure 1 illustrates the HFC phases. HFC selects piercing vertices for S from the boundary vertices of the source-side cut, i. e., the vertices incident to hyperedges in the source-side cut that are not already contained in S. Analogously, the candidates for T are the boundary vertices of the target-side cut. Note that the previous flow still satisfies the

flow constraints, and only the piercing vertex can break the maximality of the flow. HFC prefers piercing vertices that maintain the maximality of the current flow, i. e., do not create an augmenting path. This strategy is called the *avoid-augmenting-paths* piercing heuristic. Adding a vertex u to S creates an augmenting path if and only if $u \in T_r$.

3.2 Multilevel Refinement Using HyperFlowCutter

For multilevel k-way refinement with HyperFlowCutter we use the block-pair scheduling of KaHyPar-MF and the flow model construction of KaHyPar-MF and ReBaHFC. Unlike ReBaHFC, we do not explicitly contract unvisited vertices to S (resp. T). Instead, we build the flow hypergraph during the BFS and mark pins that will not be in the flow hypergraph as terminals. Hyperedges containing both terminals are removed as they cannot be eliminated from the cut. Subsequently, our weighted HFC algorithm is run on the weighted flow hypergraph. We stop once the flow exceeds the weight of the remaining hyperedges from the original cut. The difference between the weight of the original cut hyperedges and the flow value equals the decrease in connectivity.

Distance-Based Piercing. We can use the original cut to guide HFC. To avoid that bad piercing decisions make it impossible for HFC to recover parts of the original cut, we use BFS-distances from the original cut as an additional piercing heuristic. We prefer larger distances, secondary to avoiding augmenting paths. Vertices from the other side of the original cut are rated with distance -1, i. e., chosen only after one side has been entirely added to the corresponding terminal vertices. This is similar to the flow network rescaling of KaHyPar, as we first use vertices as terminals that could only be contained in larger flow networks. We maintain the boundary vertices in a bucket priority queue and select candidates uniformly at random from the highest-rated non-empty bucket. New terminal vertices are removed lazily.

3.3 Improved Balance

KaHyPar uses both flows and FM local search to refine a partition. Because FM only considers moves that maintain the balance constraint, partitions with small imbalance tend to give FM more leeway for improving the current solution. In this section, we discuss two approaches to improve the balance during HFC refinement. These can also improve the solution quality, since HFC would otherwise trade better balance for a larger cut.

Keep Piercing. Given a maximum flow and minimum cut, finding a most balanced cut of the same weight is NP-hard [8]. All vertices of a strongly connected component (SCC) of the residual network belong to the same side in a minimum cut. Hence, finding a most balanced minimum cut corresponds to a knapsack problem with partial order constraints induced by the directed acyclic graph (DAG) obtained from contracting all SCCs [36]. Each topological ordering of the DAG corresponds to a series of minimum cuts. KaHyPar computes several random topological orderings to find a cut with the same weight and less imbalance, which considerably improved solution quality [24].

Using the avoid-augmenting-paths piercing heuristic, we can perform such a sweep without the need to explicitly construct the DAG and to compute a topological ordering. Instead of stopping at the first balanced partition, we continue to pierce as long as no augmenting path is created. Since this process is fast and piercing decisions are randomized, we repeat it several times and select the partition with the smallest imbalance.

Reassigning Isolated Vertices. A vertex $v \notin S \cup T$ is called *isolated* if all of its incident hyperedges have pins in both S and T. An isolated vertex v can be moved without affecting the cut, because $v \notin S_r \cup T_r$ and its incident hyperedges remain in both the source- and target-side cut over the course of the algorithm. Using the terminology above, v is not affected by partial order constraints. This reduces the optimal assignment problem for isolated vertices to a subset sum problem. With unweighted vertices, we can easily distribute the total weight $|L|$ of a set L of isolated vertices among the two sides to achieve optimum balance. Introducing vertex weights makes the subset sum problem non-trivial, because arbitrary divisions of the total weight $\varphi(L)$ are not necessarily possible. Since isolated vertices remain isolated, the problem instances are incremental. To solve the problem, we use the pseudo-polynomial dynamic program (DP) for subset sum [10, Section 35.5]. It maintains a lookup table of partition weights that are summable with isolated vertices. After obtaining a new cut, we update the DP table to incorporate potential new isolated vertices. For each new isolated vertex v, we iterate over the subset sums x in the table and insert $\varphi(v) + x$ if it was not yet a subset sum. As an optimization, we maintain a list of ranges that are summable (consecutive entries). The balance check takes constant time per range. To merge ranges efficiently, we store a pointer from each DP table entry to its range in the list. When a new subset sum x is obtained, we check whether $x - 1$ and $x + 1$ are also subset sums, and extend or merge ranges as appropriate. Since entries in the DP table cannot be reverted easily, we do not add isolated vertices to the DP after the first balanced partition is found.

As the DP is only pseudo-polynomial, its running time may become prohibitive on instances with very large vertex weights. In our experiments, non-unit vertex weights are only the result of contractions during coarsening. Hence, the DP is polynomial in the size of the unit-weight input hypergraph. We plan to implement a simple classifier to decide whether the running time can be harmful, and deactivate the DP accordingly.

4 Maximum Flows on Weighted Hypergraphs

In this section, we present our technique for computing maximum flows on weighted hypergraphs. It generalizes the approach of Pistorious and Minoux [37] (which computes maximum flows directly on unweighted hypergraphs using the Edmonds-Karp algorithm [16]) to weighted hypergraphs and to arbitrary flow algorithms.

Let P denote the set of pins, and let $\widetilde{f}(u, e)\colon P \to \mathbb{Z}$ denote the amount of flow that vertex u sends into hyperedge e. Negative values indicate that u receives flow from e. Let $\widetilde{f}(u, e)^+ := \max(\widetilde{f}(u, e), 0)$ denote the net flow u sends into e and $\widetilde{f}(u, e)^- := \max(-\widetilde{f}(u, e), 0)$ the net flow u receives from e. Then, $f(e)$ can also be written as $\sum_{u \in e} \widetilde{f}(u, e)^+ = \sum_{u \in e} \widetilde{f}(u, e)^-$. We can push up to $c(e) - f(e) + \widetilde{f}(u, e)^- + \widetilde{f}(v, e)^+$ flow from u via e to another pin $v \in e$. The advantage of implementing flow algorithms directly on the hypergraph is the ability to identify and skip cases in which we cannot push any flow. If $c(e) - f(e) = 0$ and $\widetilde{f}(u, e) \geq 0$, we only need to iterate over the pins $v \in e$ with $\widetilde{f}(v, e) > 0$. A graph-based flow algorithm on the residual Lawler network would scan the edge (e_i, v) for *every* pin $v \in e$. For unweighted hypergraphs, each $e \in E$ has at most one pin $v \in e$ with $\widetilde{f}(v, e) > 0$, which can be stored in a separate array of size $|E|$. Since HFC needs to compute flows in forward and backward direction, an additional array is needed for pins $v' \in e$ with $\widetilde{f}(v', e) < 0$. A weighted hyperedge can have many such pins, which would require arrays of size $|P|$. Instead, we divide the pins of e into subranges $\widetilde{f}(v, e)\{< 0, = 0, > 0\}$. When the sign of $\widetilde{f}(v, e)$ changes, we insert pin v into the correct subrange by performing swaps with the range boundaries.

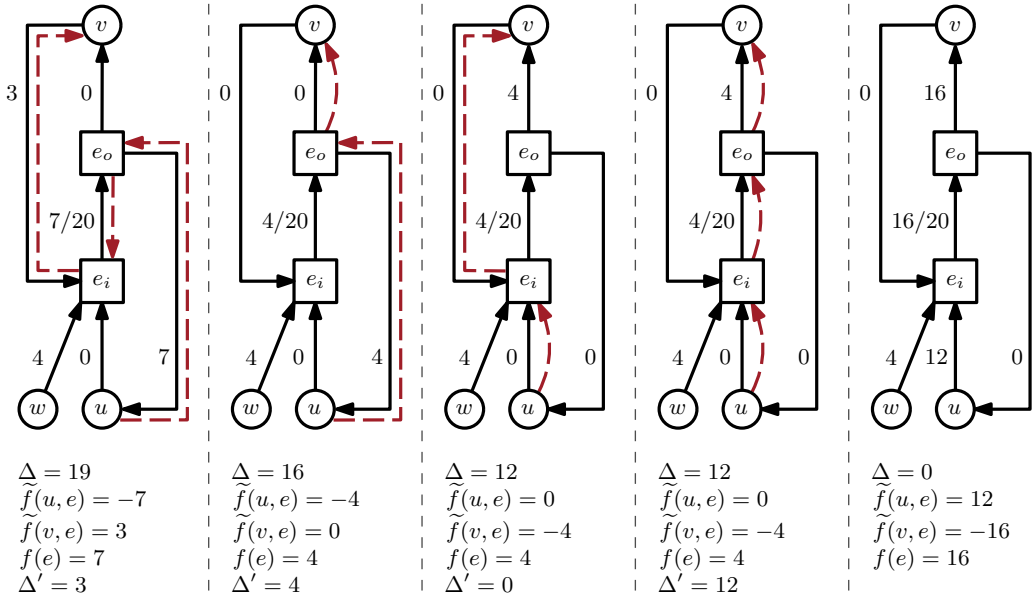


Figure 2 Example illustrating the four steps for pushing $\Delta = 19$ units of flow from u via hyperedge $e = \{u, v, w\}$ to v. Black edges show the direction of the flow, dashed red arrows the direction we want to push flow in the Lawler network. The edge (e_0, w) is omitted for readability. Current values of Δ, $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, $f(e)$, and Δ' are shown at the bottom for each state.

In addition to scanning hyperedges like vertices, pushing flow over a hyperedge is the elementary operation necessary to implement any flow algorithm on hypergraphs. Let Δ be the amount of flow to push. We update the values $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, and $f(e)$ in four steps (see Figure 2 for an example). These steps correspond to paths $p_1 = (u, e_o, e_i, v)$, $p_2 = (u, e_o, v)$, $p_3 = (u, e_i, v)$, $p_4 = (u, e_i, e_o, v)$ in the residual Lawler network. The order of these steps is important to correctly update $f(e)$. First, we push $\Delta' := \min(\Delta, \widetilde{f}(u,e)^-, \widetilde{f}(v,e)^+)$ along p_1 by setting $f(e) := f(e) - \Delta'$, $\widetilde{f}(u,e) := \widetilde{f}(u,e) + \Delta'$, $\widetilde{f}(v,e) := \widetilde{f}(v,e) - \Delta'$, and $\Delta := \Delta - \Delta'$. Then, we push $\Delta' := \min(\Delta, \widetilde{f}(u,e)^-)$ along p_2, by updating $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, and Δ as before. Note that we do not update $f(e)$ since the bridge edge (e_i, e_o) is not in p_2. Analogously to p_2, we push $\Delta' := \min(\Delta, \widetilde{f}(v,e)^+)$ along p_3. Finally, we push the remaining Δ along p_4 and update $f(e)$, $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, and Δ as for p_1.

Note that the Lawler network is just used as a means of illustration. In our implementation, we only update the $\widetilde{f}(u,e)$, $\widetilde{f}(v,e)$, and $f(e)$ values as shown at the bottom of Figure 2.

Flow Algorithm. We chose to implement Dinic's flow algorithm [14] with capacity scaling, because the flow problems for HFC refinement on our benchmark set predominantly have a small diameter (due to the BFS-based construction). Dinic's algorithm consists of two alternating phases: assigning hop-distance labels to vertices by performing a BFS on the residual network, and using DFS to find edge-disjoint augmenting paths with distance labels increasing by one along the path. In addition to vertex distance labels, we maintain hyperedge distance labels. For a vertex u, we only traverse those incident hyperedges e whose distance label matches that of u. The pins $v \in e$ are only traversed if the distance of v is 1 plus the distance of e. To distinguish the case that we can only push flow to pins v with $\widetilde{f}(v,e) > 0$, we actually maintain two different distance labels per hyperedge.

Optimizations. It suffices to initialize the BFS and DFS with the last piercing vertex p of a side, since only p can lead to newly reachable vertices. When Dinic's algorithm terminates, we already know S_r or T_r (depending on which side was pierced) and thus only compute the reachable vertices of the other side. While computing this set, we also compute the corresponding distance labels, so that the next flow computation can directly start with the DFS phase. Additionally, we infer the sets S_r, T_r, S, and T from the distance labels.

5 Experimental Evaluation

The C++17 source code for Weighted HyperFlowCutter[1] and KaHyPar-HFC[2] are available as open-source software. Experiments are performed sequentially on a cluster of Intel Xeon E5-2670 Sandy Bridge nodes with two Octa-Core processors clocked at 2.6 GHz with 64 GB RAM, 20 MB L3- and 8×256 KB L2-Cache, using only one core of a node.

We consider two configurations which differ in the constraint for vertices from V_i in the flow hypergraph. KaHyPar-HFC uses $\frac{1}{5} \cdot \varphi(V_i)$ (as used for RebaHFC [22]), while KaHyPar-HFC* uses $(1 + 16 \cdot \varepsilon)\lceil \frac{\varphi(V)}{k} \rceil - \varphi(V_j)$ (as used for KaHyPar-MF [24]). In the TR [21] we assess the impact of the algorithmic components on the solution quality of KaHyPar-HFC. Both configurations use all components.

Benchmark Set. We use a comprehensive benchmark set of real-world hypergraphs compiled by Schlag [23].[3] It consists of 488 unit-weight hypergraphs from four sources: the ISPD98 VLSI Circuit Benchmark Suite [3] (ISPD98, 18 hypergraphs), the DAC 2012 Routability-Driven Placement Benchmark Suite [34] (DAC, 10), the SuiteSparse Matrix Collection [12] (SPM, 184) and the international SAT Competition 2014 [6] (Literal, Primal, Dual, 92 hypergraphs each). Refer to the TR [21] for hypergraph sizes and refer to [23] for information on how the hypergraphs were derived. We compute partitions for $\varepsilon = 3\%$ and $k \in \{2, 4, 8, 16, 32, 64, 128\}$. Each combination of a hypergraph and a value of k constitutes one *instance*, resulting in a total of 3416 instances.

Methodology. In addition to the KaHyPar configurations, we consider hMetis [26, 27] in both recursive bisection (-R) and direct k-way (-K) mode, PaToH [9] with default (-D) and quality preset (-Q), Zoltan-AlgD [41], Mondriaan [45], as well as HYPE [33].

For each instance and partitioner, we perform ten runs with different random seeds. The only exception is HYPE [33] which produced worse solutions when randomized [24]. Hence, we report only one non-randomized run of HYPE. Running times and connectivity values per instance are aggregated using the arithmetic mean, while running times across instances are aggregated using the geometric mean to give instances of different sizes a comparable influence. To compare running times we use a combination of a scatter plot, which shows the arithmetic mean time per instance, and a box-and-whiskers plot [43]. Because small running times are more frequent, we use a fifth-root scale [11] on the y-axis. Runs that exceeded the 8 hour time limit count as 8 hours in the reported aggregates and plots.

[1] https://github.com/larsgottesbueren/WHFC
[2] https://github.com/SebastianSchlag/kahypar
[3] https://algo2.iti.kit.edu/schlag/sea2017/

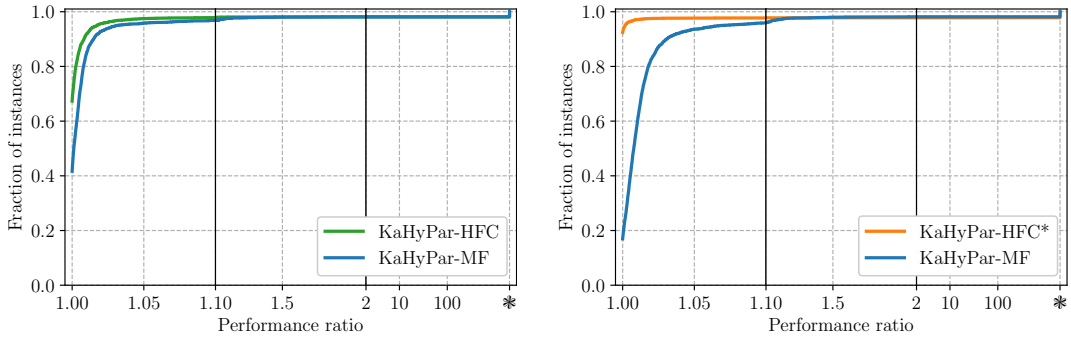


Figure 3 Performance profiles comparing our new variants with KaHyPar-MF.

For comparing solution quality we use performance profiles [15]. Let $\mathcal{A}$ denote a set of algorithms, $\mathcal{I}$ a set of instances and $\text{obj}(a, i)$ denote the objective value computed by $a \in \mathcal{A}$ on $i \in \mathcal{I}$ – in our case the arithmetic mean connectivity of 10 runs. The performance ratio

$$r(a, i) = \frac{\text{obj}(a, i)}{\min\{\text{obj}(a', i) \mid a' \in \mathcal{A}\}}$$

indicates by what factor a deviates from the best solution on instance i. In particular, algorithm a found the best solution on instance i if $r(a, i) = 1$. The performance profile

$$\rho_a \colon [1, \infty) \to [0, 1], \tau \mapsto \frac{|\{i \in \mathcal{I} \mid r(a, i) \leq \tau\}|}{|\mathcal{I}|}$$

of a is the fraction of instances for which it is within a factor of τ from the best solution. Runs that did not finish within the time limit or resulted in an error (balance violation or crash) are excluded. If this concerns all runs of an algorithm on an instance, we report the corresponding fractions as the steps at special symbols (❋, ✘ respectively).

5.1 Comparison with KaHyPar-MF

KaHyPar-HFC computes solutions with better, equal, or worse quality than KaHyPar-MF on 1933, 367, 1056 instances, respectively. On the remaining 60 instances neither finished within the time limit. As Figure 3 (left) shows, the performance ratios are consistently better, though not by a large margin. The median fraction of flow-based refinement time of KaHyPar-HFC vs KaHyPar-MF is 0.18, the 75th percentile is 0.26, and the 90th percentile is 0.51. Hence, the flow-based refinement of KaHyPar-HFC is significantly faster than that of KaHyPar-MF. Flow-based refinement constitutes about 40% of KaHyPar-MF's overall running time [24]. With a mean overall running time of 44.84s, KaHyPar-HFC is about 33% faster than KaHyPar-MF at 67.07s. The improved running time is partially due to faster flow computation and partially due to smaller flow hypergraphs. Since KaHyPar-HFC uses smaller flow hypergraphs, the improved solution quality can be attributed to the HyperFlowCutter approach. With 62.49s, KaHyPar-HFC* is moderately faster than KaHyPar-MF and computes solutions of better, equal, or worse quality on 2776, 381, 198 instances, respectively (with 61 instances on which neither the -HFC* nor the -MF variant finished within the time limit). Hence, the faster flow computation more than compensates the additional work incurred by HFC. Figure 4 shows box plots for the different phases of KaHyPar. The running times of preprocessing, coarsening, and initial partitioning remain unchanged, as they are not influenced by the refinement phase. During the refinement phase,

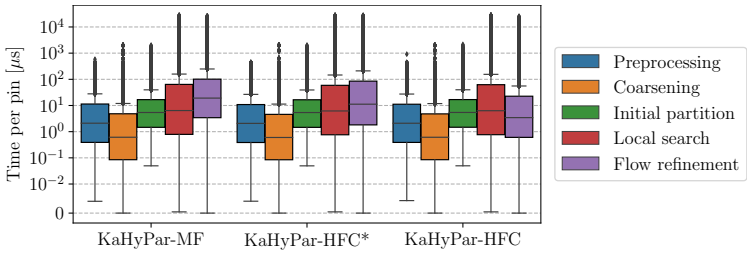


Figure 4 Box plots comparing running times (in μs per pin) of the different phases of KaHyPar.

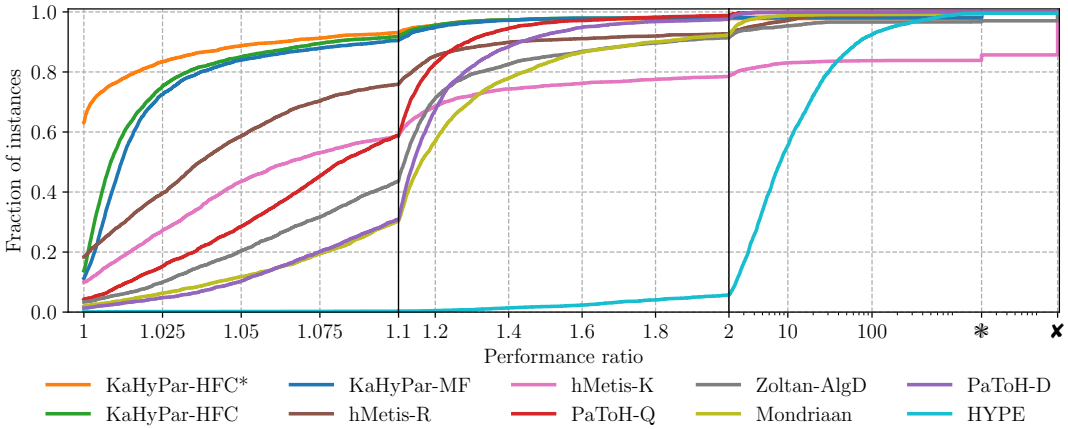


Figure 5 Performance profiles of all considered partitioners.

local search and flow-based improvement both modify the solution and thus influence one another. The plots show that the running time of local search remains largely unchanged, while our variants reduce the running time of flow-based refinement.

5.2 Comparison with other Partitioners

We now compare the three KaHyPar variants with the other state-of-the-art algorithms. Figure 5 shows that KaHyPar-HFC* outperforms all competing algorithms, and that hMetis-R emerges as the best competitor outside the KaHyPar variants. KaHyPar-HFC* computes the best solutions on 63% of all instances, KaHyPar-HFC on 14%, and hMetis-R on 18%, as shown by their $\rho_a(1)$ values. Note that these values alone do not permit a ranking between the algorithms. Both KaHyPar-MF and KaHyPar-HFC compete with KaHyPar-HFC* for the best solutions on similar instances, and thus end up with a lower $\rho_a(1)$ value. Compared individually, KaHyPar-HFC is better than hMetis-R on 69.9% of the instances. Additionally, KaHyPar-HFC and KaHyPar-MF approach the profile of KaHyPar-HFC* much faster. The KaHyPar variants are all within a factor of 1.1 of the best solution on over 90% of the instances, and within 1.4 on over 97%, whereas hMetis-R achieves 76% and 90%. PaToH-Q and PaToH-D solve more instances than hMetis-R within factors of roughly 1.2 and 1.4, and more instances than hMetis-K within 1.1 and 1.2. Mondriaan is similar to PaToH-D and Zoltan-AlgD settles between PaToH-D and PaToH-Q. The only non-multilevel algorithm HYPE is considerably worse, with only 5.7% of solutions within a factor 2 of the best.

Figure 6 shows running times for each instance. We categorize the algorithms into two groups. Algorithms in the first group, consisting of KaHyPar, hMetis and Zoltan-AlgD, invest substantial running time to aim for high-quality solutions. On the other hand, PaToH,

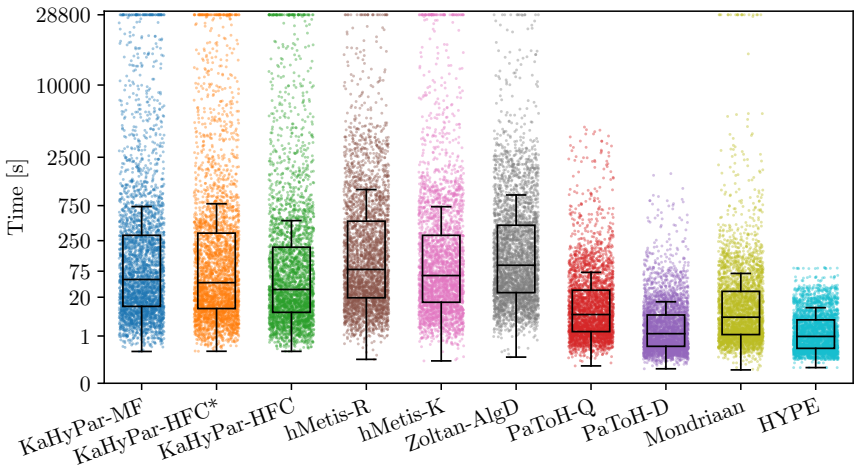


Figure 6 Box and scatter plots of arithmetic mean running times per instance.

Mondriaan and HYPE aim for fast running time and reasonable solution quality. The results show that while PaToH gives the best time-quality trade-off, KaHyPar-HFC* is the best algorithm for high-quality solutions, whereas KaHyPar-HFC offers a better time-quality trade-off than other algorithms from the first group.

In the TR [21] we report aggregate running times, timeouts and imbalanced solutions, present performance profiles for different values of k, the different instance classes of the benchmark set and a plot with only the best algorithm from each family of partitioning algorithms. The performance difference between KaHyPar and the other algorithms increases with k, which can be explained by the fact that all other partitioners except hMetis-K use recursive bisection. Furthermore, the improvement of KaHyPar-HFC* over KaHyPar-MF is especially pronounced on dual SAT instances, which have many large hyperedges.

6 Conclusion and Future Work

We leverage the powerful HyperFlowCutter refinement algorithm in the multilevel setting for k-way partitioning by integrating it into KaHyPar. For this, we extend unweighted HyperFlowCutter to weighted hypergraphs by adapting its balancing heuristics and presenting an approach to compute flows directly on weighted hypergraphs. Furthermore, we propose a distance-based piercing heuristic and use the existing avoid-augmenting-paths piercing heuristic to find partitions with small imbalance.

The most pressing area of research is to reduce the running time when using large flow hypergraphs, e. g., by further pruning of scheduled block pairs or more advanced flow algorithms like (E)IBFS [20, 19]. Furthermore, the impact of HFC refinement with small flow hypergraphs on fast and medium-quality partitioners such as PaToH could be a promising direction, since previous work on bipartitioning [22] already gave promising results.

References

1 Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

2 Yaroslav Akhremtsev, Tobias Heuer, Sebastian Schlag, and Peter Sanders. Engineering a direct k-way Hypergraph Partitioning Algorithm. In *Proceedings of the 19th Meeting on*

Algorithm Engineering and Experiments (ALENEX'17), pages 28–42. SIAM, 2017. doi:10.1137/1.9781611974768.3.

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

4 Charles J. Alpert, Jen-Hsin Huang, and Andrew B. Kahng. Multilevel Circuit Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):655–667, August 1998. doi:10.1109/43.712098.

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

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

7 Charles-Edmond Bichot and Patrick Siarry, editors. *Graph Partitioning*. Wiley, 2011. doi:10.1002/9781118601181.

8 Paul Bonsma. Most balanced minimum cuts. *Discrete Applied Mathematics*, 158(4):261–276, 2010. doi:10.1016/j.dam.2009.09.010.

9 Umit 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.

10 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

11 Nicholas J. Cox. Stata tip 96: Cube roots. *The Stata Journal*, 11(1):149–154, 2011. doi:10.1177/1536867X1101100112.

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

13 Karen Devine, Erik Boman, Robert Heaphy, Rob Bisseling, and Umit Catalyurek. Parallel Hypergraph Partitioning for Scientific Computing. In *20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE Computer Society, 2006. doi:10.1109/IPDPS.2006.1639359.

14 Yefim Dinitz. Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Mathematics-Doklady*, 11(5):1277–1280, September 1970.

15 Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002. doi:10.1007/s101070100263.

16 Jack Edmonds and Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, April 1972. doi:10.1145/321694.321699.

17 Charles M. Fiduccia and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Conference on Design Automation*, pages 175–181, 1982. doi:10.1145/800263.809204.

18 Lester R. Ford, Jr. and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

19 Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert E. Tarjan, and Renato F. Werneck. Faster and More Dynamic Maximum Flow by Incremental Breadth-First Search. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, volume 9294 of *Lecture Notes in Computer Science*, pages 619–630. Springer, 2015. doi:10.1007/978-3-662-48350-3_52.

20 Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Robert E. Tarjan, and Renato F. Werneck. Maximum Flows by Incremental Breadth-First Search. In *Proceedings of the 19th Annual European Symposium on Algorithms (ESA'11)*, volume 6942 of *Lecture Notes in Computer Science*, pages 457–468. Springer, 2011. doi:10.1007/978-3-642-23719-5_39.

21 Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. Advanced flow-based multilevel hypergraph partitioning. *arXiv preprint arXiv:2003.12110*, 2020.

22 Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA'19)*, Leibniz International Proceedings in Informatics, pages 52:1–52:17, 2019. doi:10.4230/LIPIcs.ESA.2019.52.

23 Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17)*, volume 75 of *Leibniz International Proceedings in Informatics*, 2017. doi:10.4230/LIPIcs.SEA.2017.21.

24 Tobias Heuer, Sebastian Schlag, and Peter Sanders. Network Flow-Based Refinement for Multilevel Hypergraph Partitioning. *ACM Journal of Experimental Algorithmics*, 24(2):2.3:1–2.3:36, September 2019. doi:10.1145/3329872.

25 Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner. *Proceedings of the VLDB Endowment*, 10(11):1418–1429, 2017. doi:10.14778/3137628.3137650.

26 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.

27 George Karypis and Vipin Kumar. Multilevel K-Way Hypergraph Partitioning. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, 1999. doi:10.1145/309847.309954.

28 Brian W. Kernighan and Shen Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970. doi:10.1002/j.1538-7305.1970.tb01770.x.

29 Renaud Lambiotte, Martin Rosvall, and Ingo Scholtes. From networks to optimal higher-order models of complex systems. *Nature Physics*, 15(4):313–320, 2019. doi:10.1038/s41567-019-0459-y.

30 Eugene L. Lawler. Cutsets and Partitions of hypergraphs. *Networks*, 3:275–285, 1973. doi:10.1002/net.3230030306.

31 Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley, 1990.

32 Huiqun Liu and D.F. Wong. Network-Flow-Based Multiway Partitioning with Area and Pin Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(1):50–59, 1998. doi:10.1109/43.673632.

33 Christian Mayer, Ruben Mayer, Sukanya Bhowmik, Lukas Epple, and Kurt Rothermel. HYPE: Massive Hypergraph Partitioning With Neighborhood Expansion. In *IEEE International Conference on Big Data*, pages 458–467. IEEE Computer Society, 2018. doi:10.1109/BigData.2018.8621968.

34 Viswanath Nagarajan, Charles J. Alpert, Cliff Sze, Zhuo Li, and Yaoguang Wei. The DAC 2012 Routability-driven Placement Contest and Benchmark Suite. In *Proceedings of the 49th Annual Design Automation Conference*, 2012. doi:10.1145/2228360.2228500.

35 David A. Papa and Igor L. Markov. Hypergraph Partitioning and Clustering. In Teofilo F. Gonzalez, editor, *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2007. doi:10.1201/9781420010749.

36 Jean-Claude Picard and Maurice Queyranne. On the Structure of All Minimum Cuts in a Network and Applications. *Mathematical Programming, Series A*, 22(1):121, December 1982. doi:10.1007/BF01581031.

37 Joachim Pistorius and Michel Minoux. An Improved Direct Labeling Method for the Max–Flow Min–Cut Computation in Large Hypergraphs and Applications. *International Transactions in Operational Research*, 10(1):1–11, 2003. doi:10.1111/1475-3995.00389.

38 Peter Sanders and Christian Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proceedings of the 19th Annual European Symposium on Algorithms (ESA'11)*, volume

6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer, 2011. doi:10.1007/978-3-642-23719-5_40.

39 Sebastian Schlag. *High-Quality Hypergraph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, 2020.

40 Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way Hypergraph Partitioning via n-Level Recursive Bisection. In *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX'16)*, pages 53–67. SIAM, 2016. doi:10.1137/1.9781611974317.5.

41 Ruslan Shaydulin, Jie Chen, and Ilya Safro. Relaxation-Based Coarsening for Multilevel Hypergraph Partitioning. *Multiscale Modeling and Simulation*, 17(1):482–506, 2019. doi:10.1137/17M1152735.

42 Aleksandar Trifunovic and William J. Knottenbelt. Parallel Multilevel Algorithms for Hypergraph Partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563–581, 2008. doi:10.1016/j.jpdc.2007.11.002.

43 John W. Tukey. Box-and-whisker plots. *Exploratory data analysis*, pages 39–43, 1977.

44 Bora Uçar and Cevdet Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004. doi:10.1137/S1064827502410463.

45 Brendan Vastenhouw and Rob Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, 2005. doi:10.1137/S0036144502409019.

46 Chenzi Zhang, Fan Wei, Qin Liu, Zhihao Gavin Tang, and Zhenguo Li. Graph Edge Partitioning via Neighborhood Heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 605–614. ACM Press, 2017. doi:10.1145/3097983.3098033.

Pattern Discovery in Colored Strings

Zsuzsanna Lipták
Department of Computer Science, University of Verona, Italy
zsuzsanna.liptak@univr.it

Simon J. Puglisi
Department of Computer Science, University of Helsinki, Finland
puglisi@cs.helsinki.fi

Massimiliano Rossi
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA
rossi.m@ufl.edu

Abstract

We consider the problem of identifying patterns of interest in colored strings. A colored string is a string in which each position is colored with one of a finite set of colors. Our task is to find substrings that always occur followed by the same color at the same distance. The problem is motivated by applications in embedded systems verification, in particular, assertion mining. The goal there is to automatically infer properties of the embedded system from the analysis of its simulation traces. We show that the number of interesting patterns is upper-bounded by $\mathcal{O}(n^2)$ where n is the length of the string. We introduce a baseline algorithm with $\mathcal{O}(n^2)$ running time which identifies all interesting patterns for all colors in the string satisfying certain minimality conditions. When one is interested in patterns related to only one color, we provide an algorithm that identifies patterns in $\mathcal{O}(n^2 \log n)$ time, but is faster than the first algorithm in practice, both on simulated and on real-world patterns.

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

Keywords and phrases property testing, suffix tree, pattern mining

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.12

Related Version A full version of the paper is available at `https://arxiv.org/abs/2004.04858`.

Supplementary Material An implementation of the algorithms is available online at `https://github.com/maxrossi91/colored-strings-miner`.

Funding *Simon J. Puglisi*: partially funded by the Academy of Finland via grant 319454.
Massimiliano Rossi: partially funded by the Ph.D. School of the University of Verona.

Acknowledgements We thank Johannes Fischer, Travis Gagie, and Ferdinando Cicalese for interesting discussions, and Alessandro Danese for providing an updated data set of traces.

1 Introduction

In recent years, embedded systems have become increasingly pervasive and are becoming fundamental components of everyday life. In line with this, embedded systems are required to perform increasingly demanding tasks, and in many circumstances, peoples' lives are now dependent on the correct functioning of these devices. This, in turn, has led to an increasingly complex design process for embedded systems, where a major design task is to evaluate and check the correctness of the functionality from the early stages of the development process. This functionality checking is usually done using *assertions* – logic formulae expressed in temporal logic such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) – that provide a way to express desirable properties of the device. Assertions are typically written by hand by the designers and it might take months to obtain a set of assertions

(a) Simulation trace.

T	i_1	i_2	i_3	o_1	o_2
1	0	1	0	0	0
2	1	1	0	1	0
3	0	1	0	0	0
4	1	1	0	1	1
5	0	1	0	0	0
6	1	1	0	1	0
7	1	0	1	1	1
8	0	1	0	1	0
9	1	1	0	0	0
10	0	1	0	0	0
11	1	0	1	1	1

(b) Mapping of the input and output alphabet.

Input alphabet.				Output alphabet.		
i_1	i_2	i_3	Σ	o_1	o_2	Γ
0	1	0	a	0	0	x
1	0	1	b	1	0	y
1	1	0	c	1	1	z

(c) The colored string associated with the simulation trace.

x	y	x	z	x	y	z	y	x	x	z
a	c	a	c	a	c	b	a	c	a	b
1	2	3	4	5	6	7	8	9	10	11

Figure 1 Example of a simulation trace of a device with input ports $\mathcal{I} = \{i_1, i_2, i_3\}$ and output ports $\mathcal{O} = \{o_1, o_2\}$ and the associated colored string.

that is small and effective (i.e. it covers all functionalities of the device) [15]. In order to help designers with the verification process, methodologies and tools have been developed which automatically generate assertions from simulation traces of an implementation of the device [22, 32, 8, 7]. The objective is to provide a small set of assertions that cover all behaviors of the device, in order to extend the basic manually-defined set of assertions.

A simulation trace can be viewed as a table that records, for every simulation instant T, the value assumed by the input and output ports of the device. Figure 1a shows an example of a simulation trace of a device with three input ports $\mathcal{I} = \{i_1, i_2, i_3\}$ and two output ports $\mathcal{O} = \{o_1, o_2\}$. An assertion is a logic formula expressed in temporal logic that must remain true in the whole trace. The simplest assertions involve only conditions occurring at the same simulation instant. In the simulation trace in Figure 1a, from the solid and dashed shaded boxes, we can assert that each time we have $i_1 = 1$, $i_2 = 0$, and $i_3 = 1$, then $o_1 = 1$ and $o_2 = 1$. On the other hand, we cannot assert that each time we have $i_1 = 1$, $i_2 = 1$, and $i_3 = 0$, then $o_1 = 1$ and $o_2 = 1$, because there is a counterexample in the simulation trace, namely at instant $T = 9$, where $o_1 = 0$ and $o_2 = 0$. Note assertions need not contain all input and output variables, e.g. we can assert that $i_1 = 0$ and $i_3 = 0$ implies $o_2 = 0$.

Among all possible types of assertions that can be expressed in temporal logic, an important one is given by chains of *next:* sequences of consecutive input values that, when provided to the device, uniquely determine their output, after a certain number of simulation instants. For example, in the simulation trace in Figure 1a, we can assert that each time we have, for (i_1, i_2, i_3), values $(0, 1, 0), (1, 1, 0), (0, 1, 0)$ in consecutive simulation instants, then, three instants later, we will see $o_1 = 1$ and $o_2 = 0$.

We model simulation traces with *colored strings*. A colored string is a string over an alphabet Σ, where each position is additionally assigned a color from an alphabet Γ. We will set Σ as the set of tuples of possible values for the input ports $i_1, \ldots, i_k$ and Γ as that of the output traces $o_1, \ldots, o_r$. The objective is to identify patterns in the string whose occurrence is always followed by the same color at some given distance.

Related Work. Pattern mining, with the seminal *Apriori algorithm* [1], arose from the desire to discover frequent itemsets and association rules in shopping basket data, i.e. items that were frequently bought together. Time relationhips, e.g., between entries of the database in which the basket data are stored, were later considered in so-called *sequential pattern mining* [2]. In sequential pattern mining, *episodes* are partially ordered sequences of events that appears close to each other in the sequence [25]. Given episodes of the sequence, it is

possible to build *episode rules* that establish antecedent-consequent relations among episodes. Sequential pattern mining has many applications (see, e.g., [4, 21, 10]) and has been surveyed extensively [28, 23, 16]. Unfortunately, the above setting is not applicable to our problem, since time is given only in a relative sense, i.e., whether an event happens before (or after) another, while we need to count exactly the instants occurring between two events.

In the *string mining problem* [13, 12, 14, 9, 31], one aims to discover strings that appear as a substring in more than ω strings in a collection, where ω is a user-defined parameter called the *support* of the string. The problem has been extended to mining frequent subsequences [19] and distinguishing subsequence patterns with gap constraints [20, 27, 33, 34]. String mining, however, has only superficial similarity to the colored string problems we consider.

In assertion mining, the two existing tools, *GoldMine* [32] and *A-Team* [7], are based on data mining algorithms. In particular, *GoldMine* [32] extracts assertions that predicate only on one instant of the simulation trace – i.e. they do not involve any notion of time – , using decision tree based mining or association mining [1]. Furthermore, using static analysis techniques together with sequential pattern mining, it extracts temporal assertions. The tool A-Team [7], requires the user to provide the template of the temporal assertions that they want to extract. For example, in order to extract the properties of our example in Fig. 1a, one needs to provide a template stating that we want a property of the form: "a property p_1, at the next simulation instant a property p_2, at the next simulation instant a property p_3, then after three simulation instants a property p_4". Given a set of templates, the software, using an Apriori algorithm, extracts propositions (logic formulae containing the logical connectives $\neg$, $\vee$, and $\wedge$) from the trace. Once the propositions have been extracted, the tool generates the assertion by instantiating the extracted propositions in the templates, using a decision-tree-based algorithm to find formulas that fit in the template and are verified in the simulation trace, i.e. if the trace contains no counterexample.

Our Contribution. We introduce colored strings, and propose and analyze a pattern discovery problem on colored strings which corresponds to a useful simplification of pattern mining w.r.t. assertion mining. Given a colored string and a color as input, we must find all minimal substrings that occur followed always at a the same distance by the given color. We define this problem formally in Section 2. Although this problem is simpler than the original assertion mining problem, the solution to our problem contains all the information, possibly filtered, to recover the desired set of minimal assertions in a second stage.

We first upper bound the number of minimal patterns by $\mathcal{O}(n^2)$. We then describe a suffix-tree-based algorithm to find all minimal patterns, when only one color is of interest. We go on to describe several refinements to this algorithm that significantly improve its performance in practice. Finally, we consider (practically motivated) restrictions on the patterns and show that under these restrictions performance is further improved.

2 Basics

Let $S = S[1, n]$ be a string of length $|S| = n$ on a finite ordered *alphabet* Σ. ε denotes the *empty string* of length 0. $S[i]$ denotes the i'th character of S and $S[i, j]$ the *substring* $S[i] \cdots S[j]$, if $i \leq j$, while $S[i, j] = \varepsilon$ if $i > j$. A substring T of S is called *proper* if $T \neq S$. $S^{\text{rev}} = S[n]S[n-1] \cdots S[1]$ denotes the reverse of S. For $1 \leq i \leq n$, $\text{Pref}_i(S) = S[1, i]$ is the *i'th prefix* of S, and $\text{Suf}_i(S) = S[i, n]$ is the *i'th suffix of S*.

Colored strings. Given two finite sets Σ (the alphabet) and Γ (the colors), a *colored string* over (Σ, Γ) is a string $S = S[1, n]$ over Σ together with a coloring function $f_S : \{1, \ldots, n\} \to \Gamma$. We denote by $\sigma = |\Sigma|$ and $\gamma = |\Gamma|$ the number of characters resp. of colors. Given a colored string S of length n, its reverse is denoted S^{rev}, and its coloring function $f_{S^{\text{rev}}}$ is defined by $f_{S^{\text{rev}}}(i) = f_S(n - i + 1)$, for $i = 1, \ldots, n$. When S is clear from the context, we write f for f_S and f^{rev} for $f_{S^{\text{rev}}}$.

We are interested in those substrings which are always followed by a given color y, at a given distance d. For example, let $S = \texttt{acacacbacab}$, with colors $xyxzxyzyxxz$ (see Fig. 1c). Substring `aca` occurs 3 times in S, at positions $1, 3$, and 8. In positions 1 and 3 it is followed by y at distance 3, while at position 8, the corresponding position is beyond the end of S. This leads to the following definition.

▶ **Definition 1** (y-good, y-unique, minimal)**.** *Let S be a colored string over (Σ, Γ), $y \in \Gamma$ a color, $d \leq n$ a non-negative integer, and $T = T[1, m]$ a substring of S.*

1. *An occurrence i of T is called* y-good with delay d *(or (y, d)-good) if $f(i+m-1+d) = y$.*
2. *T is called* y-unique with delay d *(or (y, d)-unique) if for every occurrence i of T, i is (y, d)-good or $i + m - 1 + d > n$.*
3. *T is called* minimally (y, d)-unique *if there exists no proper substring U of T which is y-unique with delay d', for some d' s.t. $U = T[i, j]$ and $d' = d + |T| - j$.*

In the example, the occurrence of `aca` in position 1 is $(y, 3)$- and $(y, 5)$-good, that in 3 is $(y, 1)$- and $(y, 3)$-good, while that in 8 is not (y, d)-good for any d. Therefore, the substring $T = \texttt{aca}$ is a $(y, 3)$-unique substring of S, since every occurrence i of `aca` is either $(y, 3)$-good or $i + m - 1 + d > n$. But `aca` is not minimal, since its substring `ca` is also $(y, 3)$-unique.

The introduction of minimally (y, d)-unique substrings serves to restrict the output size. Let $T = aXb$ be (y, d)-unique, with $a, b \in \Sigma$ and $X \in \Sigma^*$. We call T *left-minimal* if Xb is not (y, d)-unique, and *right-minimal* if aX is not $(y, d + 1)$-unique. We make the following simple observations about (y, d)-unique substrings. (Note that *2* is a special case of *3*.)

▶ **Observation 1.** *Let $S \in \Sigma^*$ and let T be a (y, d)-unique substring of S.*

1. *T is minimal if and only if it is left- and right-minimal.*
2. *If T is a suffix of T', then T' is also (y, d)-unique.*
3. *If $T' = UTV$ is a superstring of T such that $|V| \leq d$, then T' is $(y, d - |V|)$-unique.*

We are now ready to formally state the problem treated in this paper.

▶ **Problem 1** (Pattern Discovery Problem)**.** *Given a colored string S and a color y, report all pairs (T, d) such that T is a minimally (y, d)-unique substring of S.*

We next give an upper bound on the number of minimally (y, d)-unique substrings.

▶ **Lemma 2.** *Given string S of length n, the number of minimally (y, d)-unique substrings of S, over all $y \in \Gamma$ and $d = 0, \ldots, n$, is $\mathcal{O}(n^2)$.*

Suffix trees and indexed priority queue. We assume some familiarity with the suffix tree data structure, (see e.g. [18, 30, 24]). We denote by $\mathcal{T}(S)$ the *suffix tree* of $S\$$. $\mathcal{T}(S)$ has exactly $n + 1$ leaves, each labeled by a position from $\{1, \ldots, n + 1\}$, denoted $ln(v)$. For node v in $\mathcal{T}(S)$, $L(v)$ denotes the string represented by v, i.e., the concatenation of edge labels on the path from the root to v. We denote by $td(v)$ the *treedepth* of node v and by $sd(v) = |L(v)|$ its *stringdepth*. For internal node v, $parent(v)$ denotes v's parent and for character $c \in \Sigma$, $child(v, c)$ denotes the child of v reached by following the edge whose

label starts with c (if it exists). Given a node u with parent v, a *locus* is a pair (u,t) s.t. $sd(v) < t \leq sd(u)$. Let $[i,j]$ be the label of edge (v,u) and $k = t - sd(v)$. We define $L(u,t)$ as the string $L(v) \cdot S[i, i+k-1]$, the substring represented by locus (u,t). The one-to-one correspondence between loci of $\mathcal{T}(S)$ and substrings of $S\$$ allows us to define, for a substring T of S, the *locus of* T, $loc(T) = loc(T, \mathcal{T}(S))$ as the unique locus (u,t) in $\mathcal{T}(S)$ with the property that $L(u,t) = T$. Given $loc(T) = (u,t)$, the set of occurrences of T is the set $\{ln(v) \mid v \text{ is leaf in the subtree rooted in } u\}$. Let u be a node and $L(u) = cT$, where $c \in \Sigma$ and $T \in \Sigma^*$. The *suffix link* of u is defined as $slink(u) = loc(T)$. We also define (implicit) suffix links for loci: for locus (u,t) with $L(u,t) = cT$, define $slink(u,t) = loc(T)$. See Figure 2.

Given a suffix tree $\mathcal{T}(S)$ with k nodes, and a node u of $\mathcal{T}(S)$, let r be the rank of the node u in the breath-first search traversal of the tree. We define the reverse index BFS of u as $iBFS(u) = k - r$.

A *maximum-oriented indexed priority queue* [29, Sec. 2.4] denoted by IPQ, is a data structure that collects a set of m items with keys $k_1, \ldots, k_m$ respectively, and provides operations: `insert(`i`,`k`)` (insert the element at index i with key $k_i = k$); `demote(`i`,`k`)` (decreases the value of the key k_i, associated with i, to $k \leq k_i$); $(i,k) \leftarrow$`max()` (returns the index i and the value k of the item with maximum key k_i, breaking ties by index); $k \leftarrow$`keyOf(`i`)` (returns the value of the key k_i associated with index i). Operations `insert` and `demote` run in $\mathcal{O}(\log(m))$ time, while the operations `max`, `keyOf` and `isEmpty` are performed in $\mathcal{O}(1)$ time. We also use a function $b \leftarrow$`allNegative()` that returns *true* if all key values are negative, and *false* otherwise. We use the IPQ to store keys associated to nodes u of a suffix tree $\mathcal{T}(S)$ using $iBFS(u)$ as index. For ease of presentation, in slight abuse of notation, we will use u and $iBFS(u)$ interchangeably.

3 Pattern discovery in colored strings using the suffix tree

Our main tool will be the suffix tree of the reverse string, $\mathcal{T} = \mathcal{T}(S^{\text{rev}})$. Note that loci in $\mathcal{T}$ correspond to ending positions of substrings of S: given a locus (u,t) of $\mathcal{T}$, let $U = L(u,t)^{\text{rev}}$. Then U is a substring of S, and its occurrences are exactly the positions $i - |S| + 1$, where $i = n - ln(v) + 1$ for some leaf v in the subtree rooted in u. In the next lemma we show how to identify (y,d)-unique substrings of S with $\mathcal{T}$.

▶ **Lemma 3.** *Let U be a substring of S, $\mathcal{T} = \mathcal{T}(S^{rev})$, and $(u,t) = loc(U^{rev}, \mathcal{T})$. Then U is y-unique with delay d in S if and only if for all leaves v in the subtree rooted in u, $S^{rev}[ln(v) - d]$ is colored y under f^{rev}. In particular, U is y-unique with delay 0 in S if and only if all leaves in the subtree rooted in u are colored y under f^{rev}.*

In the following, we will refer to a *node* u of $\mathcal{T}$ as (y,d)-*unique* if $L(u)^{\text{rev}}$ is a (y,d)-unique substring of S. We can now state the following corollary:

▶ **Corollary 4.** *Let U be a substring of S, $\mathcal{T} = \mathcal{T}(S^{rev})$, and $(u,t) = loc(U^{rev}, \mathcal{T})$ s.t. u is an inner node of $\mathcal{T}(S)$. Then U is (y,d)-unique in S iff all children of u are (y,d)-unique.*

3.1 Finding all (y,d)-unique substrings

Our first algorithm Algo1 uses the suffix tree $\mathcal{T}$ of the reverse string to identify all (y,d)-unique substrings of S, not only the minimal ones, for fixed y and d. It marks the (y,d)-unique nodes of $\mathcal{T}$ in a postorder traversal of the tree. Note that if $i > n - d$, then i cannot be (y,d)-good, because the position in which we would expect a y lies beyond the end of string S. These positions are treated as if they were (y,d)-good (see Def. 1).

The function $g(u) : V(\mathcal{T}) \to \{0,1\}$ is defined as follows:

- for a leaf u with leaf number $ln(u) = i$: $g(u) = \begin{cases} 1 & \text{if either } i \leq d \text{ or } f(i-d) = y, \\ 0 & \text{otherwise,} \end{cases}$
- for an inner node u: $g(u) = \begin{cases} y & \text{if } g(v) = 1 \text{ for all children } v \text{ of } u, \\ 0 & \text{otherwise.} \end{cases}$

The algorithm computes $g(u)$ for every node u in a bottom-up fashion. If $g(u) = 1$, in addition it outputs all strings represented along the incoming edge of u, except for substrings which contain the \$-sign, i.e. suffixes of $S^{\text{rev}}\$$, see Algorithm 1.

Algorithm 1 ALGO1.

input : A colored string S, the suffix tree $\mathcal{T}$ of S^{rev}, and $y \in \Gamma$.
output : All pairs (T, d) such that T is a (y, d)-unique substring of S.

```
1  for d ← 0 to n do
2      UNIQUE(root, y, d)
3  procedure:
4      Unique(u, y, d)
5  if u is a leaf then                                          // u is a leaf
6      i ← ln(u)
7      if i ≤ d or f^rev(i − d) = y then
8          g(u) ← 1
9      else
10         g(u) ← 0
11 else                                                         // u is an inner node
12     g(u) ← ∧_{v child of u} UNIQUE(v, y, d)
13 if g(u) = 1 then
14     if u is a leaf then                                      // do not output $-substrings
15         output L(u, t)^rev for every t = sd(parent(u)) + 1, ..., sd(u) − 1
16     else
17         output (L(u, t)^rev, d) for every t = sd(parent(u)) + 1, ..., sd(u)
18 return g(u)
```

Analysis: For fixed d, computing g takes amortized $\mathcal{O}(n)$ time over the whole tree, since computing $g(u)$ is linear in the number of children of u, and therefore, charging the check whether for a child v, $g(v) = 1$, to the child node, we get constant time per node. So, for fixed d, we have $\mathcal{O}(n + K) = \mathcal{O}(n^2)$ time, where K is the number of (y, d)-unique substrings. Altogether, for $d = 0, \ldots, n$, the algorithm takes $\mathcal{O}(n^3)$ time.

In the running example (Fig. 2), for color y and delay $d = 3$, the leaf nodes $9, 2, 7, 1$, and 3 are marked with 1, thus the only inner node u which gets $g(u) = 1$ is the parent of leaves number $9, 2, 7$. Algo 1 outputs `baca`, `cbaca`, `acbaca`, `cacbaca`, `acacbaca`, `cacacbaca`, `acacacbaca`, `caca`, `acaca`, `ca`, `aca`, `ab`, `cab`, `acab`, `bacab`, `cbacab`, `acbacab`, `cacbacab`, `acacbacab`, `cacacbacab`, `bac`, `cbac`, `acbac`, `cacbac`, `acacbac`, `cacacbac`, `acacacbac`.

Remark: Note some of these substrings do not occur even once in a position such that the last character is followed by a y with delay $d = 3$. For instance, the only occurrence of the substring `bac` in S is at position 7, so we would expect to see color y at position $9 + 3 = 12$, but the string S ends at position 11. We treat this and similar questions in Section 5.

3.2 Outputting only minimally (y, d)-unique substrings

We next modify Algorithm ALGO1 to output only minimally (y, d)-unique substrings. In the suffix tree $\mathcal{T}$ of S^{rev}, minimality translates into conditions on the parent node and on the

suffix link parent node (equivalently: the suffix link) in $\mathcal{T}$. We first need another definition:

▶ **Definition 5** (Left-minimal nodes, left-minimal labels). *Let u be a node of $\mathcal{T} = \mathcal{T}(S^{rev})$, different from the root, and let $v = parent(u)$. We call u* left-minimal *for (y, d) if u is (y, d)-unique but v is not and the label of the edge (v, u) is not equal to \$. If u is (y, d)-unique and left-minimal, then we can define Left-min$(u) = x_1 \cdot L(v)^{rev}$, the left-minimal (y, d)-unique substring of S associated to u, where $x = x_1 \cdots x_k \in \Sigma^+$ is the label of edge (v, u).*

In our running example, let $u = loc(\mathcal{T}, \texttt{aca})$. Then u is left-minimal, since it is $(y, 3)$-unique but its parent is not. Its left-minimal label is *Left-min*$(u) = \texttt{ca}$. See Fig. 2.

It is easy to modify Algorithm 1 to output only left-minimal substrings: Whenever for an inner node u we get $g(u) = 0$, then for every child v of u with $g(v) = 1$, we output *Left-min*(v) (if defined). This means replacing lines 13 to 17 in Algorithm 1 (details left to the reader).

In the example, we would now output, for color y and $d = 3$, the left-minimal substrings `ca,ab,bac`. However, we are interested in substrings which are both left- *and* right-minimal. For right-minimality, Obs. 1 part (3) tells us that we need to check whether the string without the last character is $(y, d + 1)$-unique. In $\mathcal{T}$, this translates to checking the suffix link of the locus of the left-minimal substring *Left-min*(u).

▶ **Proposition 6.** *Let u be an inner node of $\mathcal{T} = \mathcal{T}(S^{rev})$, different from the root, s.t. $L(u)^{rev}$ is (y, d)-unique in S. Let $v = parent(u)$, and x_1 be the first character on the edge (v, u). Further, let $t = sd(v) + 1$, and $(u', t') = slink(u, t)$. Then the substring $U = x_1 \cdot L(v)^{rev}$ is minimally (y, d)-unique in S iff v is not (y, d)-unique and u' is not $(y, d + 1)$-unique.*

We can use Prop. 6 as follows. Once a left-minimal (y, d)-unique node u has been found, check whether u' is $(y,d+1)$-unique, where u' is the node below the locus $slink(u,sd(parent(u))+1)$. It is easy to find node u' by noting that $u' = child(slink(parent(u)), x_1)$, where x_1 is the first character of the edge label leading to u. To know whether u' is $(y, d + 1)$-unique, we process the distances d in descending order, from $d = n$ down to $d = 0$. At the end of the iteration for d, we retain the information, keeping a flag on every node u which is (y, d)-unique. See Algorithm 2.

In the running example, we know from the previous round for $d = 4$ that the only nodes that are $(y, 4)$-unique are the leaves number $4, 2, 1, 10, 3$, and 8. We can now deduce that the substring `ca` is right-minimal, because $u = loc(\texttt{ca})$ is not $(y, 4)$-unique, and $slink(loc(\mathcal{T}, \texttt{ca}^{\text{rev}})) = (u, 1)$. Looking at the string S we see that `ca` is indeed right-minimal, since `c` is not $(y, 3)$-unique: it has an occurrence, in position 6, which is not followed by a y but by an x at position $10 = 6 + 4$ (delay 4). Similarly, the other two left-minimal substrings `ab` and `bac` are also right-minimal, because their respective suffix links are not $(y, 4)$-unique.

For fixed d, the time spent on each leaf is constant (lines 5 to 10 in Algo2); we charge the check of $g(v)$ in line 12 to the child v, as well the work in lines 14 to 18 (computing *Left-min*(v) and checking the flag on v' from the previous round); these are all constant-time operations, and so we have $\mathcal{O}(n)$ time for fixed d and $\mathcal{O}(n^2)$ in total.

4 Skipping Algorithm

We now focus on discovering minimal (y, d)-unique substrings. As before, we build $\mathcal{T}(S^{\text{rev}})$ and traverse it, discovering all left-minimal (y, d)-unique substrings as we go, reporting only those that are minimal. Thus, by Proposition 6, we have to discover all left-minimal $(y, d + 1)$-unique substrings before discovering left-minimal (y, d)-unique ones.

To this end, fixing ℓ, for each node u of $\mathcal{T}(S^{\text{rev}})$, we determine the largest delay d smaller than ℓ such that $L(u)^{\text{rev}}$ can be (y, d)-unique, denoted by $h(u, \ell)$. We consider four cases:

Algorithm 2 ALGO2.

input : a colored string S, the suffix tree $\mathcal{T}$ of S^{rev} with suffix links, and $y \in \Gamma$.
output : all pairs (T, d) such that T is a minimally (y, d)-unique substring of S.

```
1  for d ← n downto 0 do
2      MinUnique(root, y, d)
3  procedure:
4      MinUnique(u, y, d)
5  if u is a leaf then                                          // u is a leaf
6      i ← ln(u)
7      if i ≤ d or f^rev(i − d) = y then
8          g(u) ← 1
9      else
10         g(u) ← 0
11 else                                                         // u is an inner node
12     g(u) ← ∧_{v child of u} MinUnique(v, y, d)
13 if g(u) = 0 then                          // outputting minimal substrings for children
14     for each child v with g(v) = 1 do
15         if Left-min(v) is defined then
16             (v', t) ← slink(v, sd(u) + 1)
17             if v' is not (y, d + 1)-unique then              // flag from previous round
18                 output (Left-min(v), d)
19 return g(u)
```

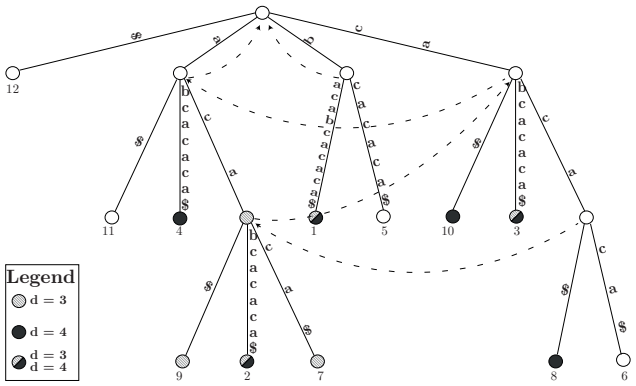


Figure 2 The suffix tree $\mathcal{T}$ of the reverse string $S^{\text{rev}} =$ `bacabcacaca`, where $S =$ `acacacbacab` (our running example). For clarity, the edges carry the label itself rather than a pair of pointers into the string. Suffix links are drawn as dotted directed edges. The nodes are colored according to function g for the character y, for $d = 3$ (dashed) and for $d = 4$ (solid).

- If u is a leaf, then $L(u)^{\text{rev}}$ is the j-prefix of S, where $j = n - ln(u) + 1 = |L(u)|$
  - If $ln(u) < \ell$, then $j + \ell - 1 > n$ thus $L(u)^{\text{rev}}$ is $(y, \ell - 1)$-unique since the position of the color is beyond the end of the string, thus $h(u, \ell) = \ell - 1$.
  - If $ln(u) \geq \ell$ and there exists an $i < \ell$ such that $f(j + i) = y$, then the highest possible value $d < \ell$ such that $L(u)^{\text{rev}}$ is (y, d)-unique is given by the position of the furthest occurrence of y within a distance of $\ell-1$ from j, thus $h(u, \ell) = \max\{i < \ell \mid f(j+i) = y\}$.
  - Otherwise, if such i does not exists, we set $h(u, \ell) = -1$.
- If u is an internal node of $\mathcal{T}(S^{\text{rev}})$, then let $k = \min\{h(v, \ell) \mid v \text{ child of } u\}$, since it is not possible that $L(u)^{\text{rev}}$ is (y, d')-unique, for any $k < d' < \ell$, thus $h(u, \ell) = k$.

When u is an inner node, in general, we do not know if $L(u)^{\text{rev}}$ is (y, d)-unique for $d = h(u, \ell)$, unless for all nodes v in the subtree rooted at u, there exists an ℓ_v such that $h(u, \ell) < \ell_v \leq \ell$ and $h(v, \ell_v) = h(u, \ell)$. This is true if $h(v, d + 1) = h(u, \ell)$ for all v.

▶ **Lemma 7.** *Let u be a node of $\mathcal{T}(S^{rev})$, fix d, $h(u, d+1) = d$ if and only if u is (y, d)-unique.*

To evaluate $h(u, \ell) = \max\{i < \ell \mid f(j + i) = y\}$ when u is a leaf and such i exists, we define a bitvector $b_y[1, 2n]$ such that $b_y[i] = 1$ only if $f(i) = y$ or $i > n$. We preprocess b_y for $\mathcal{O}(1)$-time `rank` and `select` queries [5]. Given node u with $ln(u) \geq \ell$, let $j = n - ln(u) + 1$. We have that $h(u, \ell) = \max\{\texttt{select}(b_y, \texttt{rank}(b_y, j + \ell)) - j, -1\}$.

We use the $h(u, \ell)$ function in the following way, during the discovery process of all (y, d)-unique substrings of S, provided that we have already discovered all $(y, d + 1)$-unique substrings of S. Let $\ell = d + 1$, for all nodes u of $\mathcal{T}(S^{\text{rev}})$ we store the values $h(u, \ell)$. We discover the minimally (y, d)-unique substrings of S, finding all nodes u such that $h(u, \ell) = d$. Among those, the nodes that are also left-minimal are those nodes u such that, $h(parent(u), \ell) < d$. We then check if u is also right-minimal by checking if its suffix-link parent is $(y, d + 1)$-unique, as in Algorithm 2.

The key idea of the skipping algorithm is to keep the values $h(u, \ell)$ updated during the process. Let $H(u)$ be the array that, at the beginning of the discovery of all (y, d)-unique substrings of S, stores the values $h(u, \ell)$. We want to keep the array H updated in a way such that, after we discovered all (y, d)-unique substrings of S, for all nodes u, $H(u) = h(u, \ell - 1)$. Thus, once we discover that a node u is left-minimal (y, d)-unique, we update the value of $H(u) = h(u, \ell - 1)$. We then update the following values:

- for all nodes v in the subtree rooted in u, we update the values $H(u) = h(u, \ell - 1)$.
- for all nodes p ancestors of u, we update the values $H(p) = \min(H(p), h(u, \ell - 1))$

▶ **Lemma 8.** *Given $\mathcal{T}(S^{rev})$, fix d, for all nodes u of $\mathcal{T}(S^{rev})$, let $H(u) = h(u, d + 1)$. If for all nodes u such that $H(u) = d$ we (i) set $H(u) = h(u, d)$, and (ii) for all ancestors p of u, set $H(p) = \min\{H(p), h(u, d)\}$, then, for all nodes u of $\mathcal{T}(S^{rev})$, $H(u) = h(u, d)$.*

In order to efficiently find all nodes u such that $h(u, \ell) = d$ and $h(parent(u), \ell) < d$, we use a *maximum-oriented indexed priority queue*, storing the values of $H(u)$ as keys and $iBFS(u)$ as index. Under this condition, if two nodes have the same key value, then parents have higher priority than their children in IPQ. We keep the priority queue updated using a `demote` operation while we discover left-minimal nodes and we update the values of the array H stored as keys of IPQ. Algorithm 3 shows how to compute $h(u, \ell)$ for a given node u, and how we update the values of the keys in the IPQ for all children v of u.

The skipping algorithm (see Algorithm 4), initializes priority queue IPQ by inserting all nodes of $\mathcal{T}(S^{\text{rev}})$ with key $n + 1$. We then repeat the following until there exists a node with non-negative key: extract the max element (u, ℓ) of IPQ, decide whether it has to be

Algorithm 3 $h(u, d)$.

input : A node u in suffix tree $\mathcal{T}(S^{\text{rev}})$ and integer ℓ.
output : Maximum delay $d < \ell$ s.t. $L(u)^{\text{rev}}$ can be (y, d)-unique.

1 $min_d \leftarrow \ell - 1$
2 **if** *u is a leaf* **then**
3 $\quad j \leftarrow n - ln(u) + 1$
4 $\quad min_d \leftarrow \max\{\texttt{select}(b_y, \texttt{rank}(b_y, j + \ell)) - j, -1\}$
5 **else**
6 $\quad$ **forall** *children v of u* **do**
7 $\quad\quad d = h(v, \ell)$
8 $\quad\quad$ **if** $min_d < d$ **then**
9 $\quad\quad\quad min_d \leftarrow d$
10 $IPQ.\texttt{demote}(u, min_d)$
11 **return** min_d

Algorithm 4 Skipping.

input : A colored string S, and a color $y \in \Gamma$
output : All minimal (y, d)-unique substrings of S.

1 **forall** *nodes v of $\mathcal{T}(S^{rev})$* **do**
2 $\quad IPQ.\texttt{insert}(v, n + 1)$
3 **while** $IPQ.\texttt{allNegative()} = false$ **do**
4 $\quad (u, d) \leftarrow IPQ.\texttt{max()}$
5 $\quad (u', t) = slink(u, sd(parent(u)) + 1)$
6 $\quad$ **if** *u' is not $(y, d+1)$-unique* **then**
$\quad\quad$ `// flag from previous round`
7 $\quad\quad$ output $(d, \textit{Left-min}(u))$
8 $\quad min_d = h(u, d)$
9 $\quad$ **forall** *ancestors v of u* **do**
10 $\quad\quad$ **if** $IPQ.\texttt{keyOf}(v) > min_d$ **then**
11 $\quad\quad\quad IPQ.\texttt{demote}(v, min_d)$

reported, i.e. if it is right-minimal; apply Algorithm 3 to update the key values of all nodes in the subtree at u and then update the values of the keys of all ancestors of u.

For all nodes u in $\mathcal{T}(S^{\text{rev}})$, the key value associated to u in IPQ is initially $n + 1$. Each time Algorithm 4 and Algorithm 3 visit a node, the key value of u in IPQ is decreased (via `demote()`) until it becomes negative. Thus, for each node we perform at most $n + 1$ `demote()` operations. Since the number of nodes in $\mathcal{T}(S^{\text{rev}})$ is linear in n, Algorithm 4 runs in $\mathcal{O}(n^2 \log(n))$ time.

5 Output restrictions and algorithm improvement

We now discuss some practically-minded output restrictions. These could be implemented as a filter to the output, thus discarding some solutions, but when considered as part of the problem, they lead to an improvement for the skipping algorithm.

Note that our definition of (y, d)-unique allows that a substring occurs only once, or that none of its occurrences is followed by a y with delay d, because they are all close to the end of string. We restrict our attention to (y, d)-unique substrings with at least two occurrences followed by y with delay d. Given a colored string S, let T be minimally (y, d)-unique. Now we report (T, d) if and only if the following holds: 1) there are at least two occurrences of T in S; 2) let i be the second smallest occurrence of T in S, then $i + |T| - 1 + d \leq n$.

A substring T that satisfies the above conditions is called *real type* minimally (y, d)-unique substring. In order to satisfy those conditions, it is enough to perform the output operations at line 7 of Algorithm 4 and at line 18 of Algorithm 2 if the node u is not a leaf and the value of the second greatest suffix of S^{rev} in the subtree rooted in u is greater than or equal to d. Since each node u in the suffix tree $\mathcal{T}(S^{\text{rev}})$, corresponds to an interval $[i, j]$ of the suffix array of S^{rev}, we can find the second greatest suffix using a range maximum query rMq data structure [11] built on the suffix array of S^{rev}. Then, the second greatest suffix can be found in $\mathcal{O}(1)$ time, using $2n + o(n)$ bits of extra space.

The $h(u, \ell)$ function is used in Algorithm 4 in order to find left-minimal nodes in the suffix tree. If we consider the output restrictions as part of the problem, then we do not have to report minimally (y, d)-unique substrings that occur only once, i.e., leaves in $\mathcal{T}(S^{\text{rev}})$. Then, for all nodes u such that all children of u are leaves, we can directly compute the

Table 1 Real-world datasets used in the experiments. In columns 1 and 2, we report names and descriptions of the hardware designs used to generate the simulation traces. In columns 3 and 4, we give the number of primary inputs resp. of primary outputs. In column 5 we report the length of the simulation trace, and in columns 6 and 7 the size of the alphabet and the number of colors, respectively. For each design we fixed a color y, and report in col. 8 number n_y of y characters.

Design	Description	PIs	POs	n	σ	γ	n_y
`b03`	Resource arbiter [6]	6	4	100 000	17	5	3210
`b06`	Interrupt handler [6]	4	6	100 000	5	4	44 259
`s386`	Shynthetized controller [3]	9	7	100 000	129	2	8290
`camellia`	Symmetric key block cypher [26]	262	131	103 615	70	224	2292
`serial`	Serial data transmitter	11	2	100 000	118	2	16 353
`master`	Wishbone bus master [26]	134	135	100 000	417	80	759

highest value of $d < \ell$ such that $L(u)^{\text{rev}}$ is (y, d)-unique. This leads to the definition of the $fast\_h(u, \ell)$ function for a node u of $\mathcal{T}(S^{\text{rev}})$. The function $fast\_h(u, \ell)$ is defined similarly to the function $h(u, \ell)$ with the additional following case: If all children of u are leaves, we can directly compute the highest value of $d < \ell$ such that $L(u)^{\text{rev}}$ is (y, ℓ)-unique as the largest value $d < \ell$ such that, for each child v of u, $h(v, d + 1) = d$. In other words, we want the largest $d < \ell$ such that all children of v are (y, d)-unique.

6 Experimental results

We implemented the algorithms[1] and measured their performance on randomly generated datasets and real-world datasets. Experiments were performed on a 3.4 GHz Intel Core i7-6700 CPU with 8 MiB L3 cache and 16 GiB of DDR4 RAM running Ubuntu 16.04 (64bit, kernel 4.4.0). The compiler was `g++` version 5.4.0 with `-O3 -DNDEBUG -funroll-loops -msse4.2` options. Runtimes were recorded with the C++11 `high_resolution_clock` facility.

We used two different datasets, one consisting of randomly generated data and one consisting of real-world data. Randomly generated data varied string length $n = 100$, 1000, 10 000, 100 000, alphabet size $\sigma = 2, 4, 8, 16, 32$, and number of colors $\gamma = 2, 4, 8, 16, 32$. Strings were generated one character (and color) at a time, i.e. fixing σ and γ, the string of length $n = 1000$ is a prefix of the string $n = 10\,000$. Characters and colors follow a uniform distribution. We report only on experiments for $n = 1000, 10\,000, 100\,000$, $\sigma = 2, 8, 32$, color number $\gamma = 2, 8, 32$, and seed 0, which are representative of the trend we observed for all settings. The real-world data is the result of a simulation on a set of established benchmarks in embedded systems verification [3, 6, 26], reported in Table 1. The benchmarks are descriptions of hardware design at the register-transfer level of abstraction.

We compared implementations **`base`** (the baseline, Algorithm 2); **`skip`** (the skipping algorithm in Algorithm 4, using the h function in Algorithm 3); and **`real`** (as **`skip`**, but using the $fast\_h$ function). All algorithms report minimally (y, d)-unique substrings only if they are *real type*. We used the `sdsl-lite` library [17] for compressed suffix trees and supporting data structures.

We performed all experiments five times and report the average. Results are reported in Figure 3 and Table 2. Figure 3 shows the results for **`base`**, **`skip`** and **`real`** on the random data set. We observe how the algorithms scale with respect to increasing the number of colors, which has the effect of reducing the number of y-colored characters; increasing alphabet size;

[1] Available online at: `https://github.com/maxrossi91/colored-strings-miner`.

and increasing the length of the text. In summary, for all algorithms: running time decreases with increasing number of colors and increases with alphabet size. We observe a quadratic dependence on n in line with our asymptotic analysis.

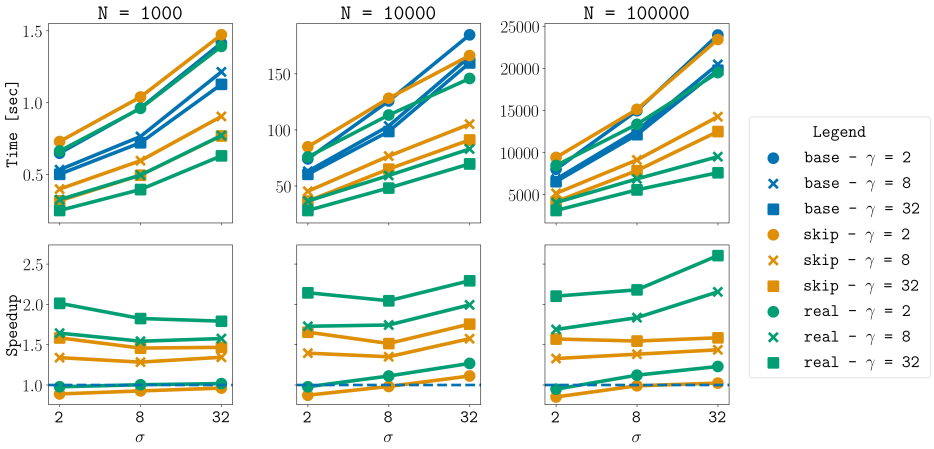


Figure 3 Results of the execution of algorithms `base` (color blue), `skip` (color orange) and `real` (color green) over the randomly generated data for $N = 10^3$, 10^4, and 10^5. The x axis represents the values of $\sigma = \{2, 8, 32\}$, and the different markers represents the values of $\gamma = \{2, 8, 32\}$ (resp. circles, crosses, and boxes). The three plots in the first row report execution times. The plots in the second row report speedups of `skip` and `real` with respect to algorithm `base` represented as the dashed blue line at constant 1.0.

Figure 3 shows that the `skip` algorithm is almost always faster than the `base` algorithm, and that the average speedup is 1.30, with a maximum of 1.75. Moreover, we have that the the `real` algorithm is almost always faster than the `skip` algorithm, and the average speedup is 1.25, with a maximum of 1.64. Finally, the average speedup between `real` and `base` is 1.65, with a maximum of 2.60 in the case of $N = 100\,000$, $\sigma = 32$ and $\gamma = 32$.

Table 2 shows the results for `base`, `skip` and `real` algorithms on the real dataset. Here, we observe a similar trend to the random data, but the speedup of `real` with respect to `base` is much higher – 3.40 on average, with a maximum of 11.88 on the `master` device. However, on three of the six datasets, `base` is faster than `skip`, and faster than `real` on one.

7 Conclusion

We studied pattern discovery problems on colored strings motivated by applications in embedded system verification. To the best of our knowledge this is the first principled

Table 2 Results of the execution of algorithms `base`, `skip` and `real` over the real-world dataset. The first column reports the name of the design from which the simulation trace is retrieved.

	Execution Time (*sec*)			Speedup (*ratio*)		
Design	`base`	`skip`	`real`	$^{\texttt{base}}/_{\texttt{skip}}$	$^{\texttt{skip}}/_{\texttt{real}}$	$^{\texttt{base}}/_{\texttt{real}}$
`b03`	2258.94	3925.08	3149.46	0.58	1.25	0.72
`b06`	3575.78	5463.50	4511.97	0.65	1.21	0.79
`s386`	3285.55	5347.90	3719.14	0.61	**1.44**	0.88
`camellia`	3015.91	1098.77	1071.75	2.74	1.03	2.81
`serial`	3325.84	989.98	1003.72	3.36	0.99	3.31
`master`	3365.56	284.19	283.24	**11.84**	1.00	**11.88**

algorithmic treatment of these problems. Our fastest algorithm stores, during the discovery process, for each distinct substring the next delay value which is (y, d)-unique, using a priority queue to find these values and to identify minimally (y, d)-unique substrings. The algorithm is especially fast on real-world instances. Under a variant of the minimality condition oriented toward real-world instances the algorithm becomes even faster. We are currently working with colleagues in embedded systems to integrate these algorithms into their analysis workflows.

References

1 Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. 20th International Conference on Very Large Data Bases (VLDB)*, volume 1215, pages 487–499. Morgan Kaufmann, 1994.

2 Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proc. Eleventh International Conference on Data Engineering (ICDE)*, volume 95, pages 3–14. IEEE Computer Society, 1995.

3 Franc Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *Proc. 1989 IEEE international symposium on circuits and systems (ISACS)*, volume 3, pages 1929–1934. IEEE, 1989.

4 Chung-Wen Cho, Ying Zheng, Yi-Hung Wu, and Arbee LP Chen. A tree-based approach for event prediction using episode rules over event streams. In *Proc. International Conference on Database and Expert Systems Applications (DEXA 2008)*, volume 5181 of *LNCS*, pages 225–240. Springer, 2008.

5 David Clark. *Compact PAT trees.* PhD thesis, University of Waterloo, 1997.

6 Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. Rt-level itc'99 benchmarks and first atpg results. *IEEE Design & Test of computers*, 17(3):44–53, 2000.

7 Alessandro Danese, Nicolò Dalla Riva, and Graziano Pravadelli. A-team: Automatic template-based assertion miner. In *Proc. 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

8 Alessandro Danese, Tara Ghasempouri, and Graziano Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In *Proc. 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 67–72. IEEE, 2015.

9 Jasbir Dhaliwal, Simon J Puglisi, and Andrew Turpin. Practical efficient string mining. *IEEE Transactions on Knowledge and Data Engineering*, 24(4):735–744, 2010.

10 Lina Fahed, Armelle Brun, and Anne Boyer. DEER: Distant and Essential Episode Rules for early prediction. *Expert Systems with Applications*, 93:283–298, 2018.

11 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

12 Johannes Fischer, Volker Heun, and Stefan Kramer. Fast frequent string mining using suffix arrays. In *Proc. Fifth IEEE International Conference on Data Mining (ICDM 2005)*, pages 609–612. IEEE, 2005.

13 Johannes Fischer, Volker Heun, and Stefan Kramer. Optimal string mining under frequency constraints. In *Proc. European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2006)*, volume 4213 of *LNCS*, pages 139–150. Springer, 2006.

14 Johannes Fischer, Veli Mäkinen, and Niki Välimäki. Space efficient string mining under frequency constraints. In *Proc. Eighth IEEE International Conference on Data Mining (ICDM 2008)*, pages 193–202. IEEE, 2008.

15 Harry D Foster, Adam C Krolnik, and David J Lacey. *Assertion-based design.* Springer Science & Business Media, 2004.

16 Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, Yun Sing Koh, and Rincy Thomas. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1(1):54–77, 2017.

17 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 2014)*, volume 8504 of *LNCS*, pages 326–337. Springer, 2014.

18 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*. Cambridge University Press, Cambridge, United Kingdom, 1997.

19 Koji Iwanuma, Ryuichi Ishihara, Yoh Takano, and Hidetomo Nabeshima. Extracting frequent subsequences from a single long data sequence a novel anti-monotonic measure and a simple on-line algorithm. In *Proc. Fifth IEEE International Conference on Data Mining (ICDM 2005)*, pages 186–195. IEEE, 2005.

20 Xiaonan Ji, James Bailey, and Guozhu Dong. Mining minimal distinguishing subsequence patterns with gap constraints. *Knowledge and Information Systems*, 11(3):259–286, 2007.

21 Srivatsan Laxman, Vikram Tankasali, and Ryen W White. Stream prediction using a generative model based on frequent episodes in event sequences. In *Proc. 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD 2008)*, pages 453–461. ACM, 2008.

22 Lingyi Liu, David Sheridan, Viraj Athavale, and Shobha Vasudevan. Automatic generation of assertions from system level design using data mining. In *Proc. Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 191–200. IEEE Computer Society, 2011.

23 Nizar R Mabroukeh and Christie I Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)*, 43(1):3, 2010.

24 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design*. CUP, 2015.

25 Heikki Mannila, Hannu Toivonen, and A Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data mining and knowledge discovery*, 1(3):259–289, 1997.

26 OpenCores. Available at `https://opencores.org/`. Accessed 05-03-2019.

27 Tinghai Pang, Lei Duan, Jesse Li-Ling, and Guozhu Dong. Mining Similarity-Aware Distinguishing Sequential Patterns from Biomedical Sequences. In *Proc. Second International Conference on Data Science in Cyberspace (DSC 2017)*, pages 43–52. IEEE, 2017.

28 Jian Pei, Jiawei Han, and Wei Wang. Constraint-based sequential pattern mining: the pattern-growth methods. *Journal of Intelligent Information Systems*, 28(2):133–160, 2007.

29 Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 2011.

30 Bill Smyth. *Computing Patterns in Strings*. Pearson Addison-Wesley, Essex, England, 2003.

31 Niko Välimäki and Simon J Puglisi. Distributed string mining for high-throughput sequencing data. In *Proc. 12th International Workshop on Algorithms in Bioinformatics (WABI 2012)*, volume 7534 of *LNCS*, pages 441–452. Springer, 2012.

32 Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *Proc. 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 626–629. IEEE, 2010.

33 Xianming Wang, Lei Duan, Guozhu Dong, Zhonghua Yu, and Changjie Tang. Efficient mining of density-aware distinguishing sequential patterns with gap constraints. In *International Conference on Database Systems for Advanced Applications (DASFAA 2014)*, volume 8421 of *LNCS*, pages 372–387. Springer, 2014.

34 Xindong Wu, Xingquan Zhu, Yu He, and Abdullah N Arslan. PMBC: Pattern mining from biological sequences with wildcard constraints. *Computers in Biology and Medicine*, 43(5):481–492, 2013.

Fast and Linear-Time String Matching Algorithms Based on the Distances of q-Gram Occurrences

Satoshi Kobayashi
Graduate School of Information Sciences, Tohoku University, Sendai, Japan
satoshi_kobayashi@shino.ecei.tohoku.ac.jp

Diptarama Hendrian
Graduate School of Information Sciences, Tohoku University, Sendai, Japan
diptarama@tohoku.ac.jp

Ryo Yoshinaka
Graduate School of Information Sciences, Tohoku University, Sendai, Japan
ryoshinaka@tohoku.ac.jp

Ayumi Shinohara
Graduate School of Information Sciences, Tohoku University, Sendai, Japan
ayumis@tohoku.ac.jp

Abstract

Given a text T of length n and a pattern P of length m, the string matching problem is a task to find all occurrences of P in T. In this study, we propose an algorithm that solves this problem in $O((n+m)q)$ time considering the distance between two adjacent occurrences of the same q-gram contained in P. We also propose a theoretical improvement of it which runs in $O(n+m)$ time, though it is not necessarily faster in practice. We compare the execution times of our and existing algorithms on various kinds of real and artificial datasets such as an English text, a genome sequence and a Fibonacci string. The experimental results show that our algorithm is as fast as the state-of-the-art algorithms in many cases, particularly when a pattern frequently appears in a text.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases String matching algorithm, text processing

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.13

Supplementary Material The implementations of our algorithms are available at `https://github.com/ushitora/distq`.

Funding *Diptarama Hendrian*: Supported by JSPS KAKENHI Grant Number JP19K20208.

1 Introduction

The exact string matching problem is a task to find all occurrences of P in T when given a text T of length n and a pattern P of length m. A brute-force solution of this problem is to compare P with all the substrings of T of length m. It takes $O(nm)$ time. The Knuth-Morris-Pratt (KMP) algorithm [17] is well known as an algorithm that can solve the problem in $O(n+m)$ time. However, it is not efficient in practice, because it scans every position of the text at least once. The Boyer-Moore algorithm [3] is famous as an algorithm that can perform string matching fast in practice by skipping many positions of the text, though it has $O(nm)$ worst-case time complexity. Like this, many efficient algorithms whose worst-case time complexity is the same or even worse than the naive method have been proposed so far [16, 21, 19]. For example, the HASHq algorithm [19] focuses on the substrings of length q in a pattern and obtains a larger shift amount. However, considering that such an algorithm is embedded in software and actually used, if the worst-case input strings are given, the operation of the software may be slowed down. Therefore, an algorithm

that operates theoretically and practically fast is important. Franek et al. [14] proposed the Franek-Jennings-Smyth (FJS) algorithm, which is a hybrid of the KMP algorithm and the Sunday algorithm [21]. The worst-case time complexity of the FJS algorithm is $O(n+m+\sigma)$ and it works fast in practice, where σ is the alphabet size. Kobayashi et al. [18] proposed an algorithm that improves the speed of the FJS algorithm by combining a method that extends the idea of the Quite-Naive algorithm [4]. This algorithm has the same worst-case time complexity as the FJS algorithm, and it runs faster than the FJS algorithm in many cases. The LWFRq algorithm [8] is a practically fast algorithm that works in linear time. This algorithm uses a method of quickly recognizing substrings of a pattern using a hash function. See [12, 15] for recent surveys on exact string matching algorithms.

This paper proposes two new exact string matching algorithms based on the HASHq and KMP algorithms incorporating a new idea based on the distances of occurrences of the same q-grams. The time complexity of the preprocessing phase of the first algorithm is $O(mq)$ and the search phase runs in $O(nq)$ time. The second algorithm improves the theoretical complexity of the first algorithm, where the preprocessing and searching times are $O(m)$ and $O(n)$, respectively. Our algorithms are as fast as the state-of-the-art algorithms in many cases. Particularly, our algorithms work faster when a pattern frequently appears in a text.

This paper is organized as follows. Section 2 briefly reviews the KMP and HASHq algorithms, which are the basis of the proposed algorithms. Section 3 proposes our algorithms. Section 4 shows experimental results comparing the proposed algorithms with several other algorithms using artificial and practical data. Section 5 draws our conclusions.

2 Preliminaries

2.1 Notation

Let Σ be a set of characters called an *alphabet* and $\sigma = |\Sigma|$ be its size. Σ^* denotes the set of all strings over Σ. The length of a string $w \in \Sigma^*$ is denoted by $|w|$. The *empty string*, denoted by ε, is the string of length zero. The i-th character of w is denoted by $w[i]$ for each $1 \leq i \leq |w|$. The substring of w starting at i and ending at j is denoted by $w[i:j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i:j] = \varepsilon$ if $i > j$. A string $w[1:i]$ is called a *prefix* of w and a string $w[i:|w|]$ is called a *suffix* of w. A string v is a *border* of w if v is both a prefix and a suffix of w. Note that the empty string is a border of any string. Moreover, a prefix, a suffix or a border v of w is called *proper* when $v \neq w$. The length of the longest proper border of $w[1:i]$ for $1 \leq i \leq |w|$ is given by

$$Bord_w[i] = \max\{\, j \mid w[1:j] = w[i-j+1:i] \text{ and } 0 \leq j < i \,\}.$$

Throughout this paper, we assume Σ is an integer alphabet.

2.2 The exact string matching problem

The exact string matching problem is defined as follows:

Input: A text $T \in \Sigma^*$ of length n and a pattern $P \in \Sigma^*$ of length m,

Output: All positions i such that $T[i:i+m-1] = P$ for $1 \leq i \leq n-m+1$.

We will use a text $T \in \Sigma^*$ of length n and a pattern $P \in \Sigma^*$ of length m throughout the paper.

Let us consider comparing $T[i:i+m-1]$ and $P[1:m]$. The naive method compares characters of the two strings from left to right. When a character mismatch occurs, the

Algorithm 1 Computing *KMP_Shift*.

```
1  Function PreKMPShift(P)
2      m ← |P|; i ← 1; j ← 0;
3      Strong_Bord_P[1] ← −1;
4      while i ≤ m do
5          while j > 0 and P[i] ≠ P[j] do j ← Strong_Bord_P[j];
6          i ← i + 1; j ← j + 1;
7          if i ≤ m and P[i] = P[j] then Strong_Bord_P[i] ← Strong_Bord_P[j];
8          else Strong_Bord_P[i] ← j;
9      for j ← 1 to m do KMP_Shift[j] ← j − Strong_Bord_P[j] − 1;
10     return KMP_Shift
```

pattern is shifted to the right by one character. That is, we compare $T[i+1:i+m]$ and $P[1:m]$. This naive method takes $O(nm)$ time for matching. There are a number of ideas to shift the pattern more so that searching T for P can be performed more quickly, using shifting functions obtained by preprocessing the pattern.

2.3 Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt (KMP) algorithm [17] is well known as a string matching algorithm that has linear worst-case time complexity. When the KMP algorithm has confirmed that $T[i:i+j-2] = P[1:j-1]$ and $T[i+j-1] \neq P[j]$ for some $j \leq m$, it shifts the pattern so that a suffix of $T[i:i+j-2]$ matches a prefix of P and we do not have to re-scan any part of $T[i:i+j-2]$ again. That is, the pattern can be shifted by $j-k-1$ for $k = \mathit{Bord}_P[j-1]$. In addition, if $P[k+1] = P[j]$, the same mismatch will occur again after the shift. In order to avoid this kind of mismatch, we use $\mathit{Strong\_Bord}[1:m+1]$ given by

$$\mathit{Strong\_Bord}_P(j) = \begin{cases} \mathit{Bord}_P(m) & \text{if } j = m+1, \\ \max\big(\{\, k \mid P[1:k] = P[j-k:j-1],\ P[k+1] \neq P[j], \\ \qquad 0 \leq k < j \,\} \cup \{-1\}\big) & \text{otherwise.} \end{cases}$$

The amount $\mathit{KMP\_Shift}[j]$ of the shift is given by

$$\mathit{KMP\_Shift}[j] = j - \mathit{Strong\_Bord}_P(j) - 1.$$

This function has a domain of $\{1, \ldots, m+1\}$ and is implemented as an array in the algorithm. Hereafter, we identify some functions and the arrays that implement them.

▶ **Fact 1.** *If $P[1:j-1] = T[i:i+j-2]$ and $P[j] \neq T[i+j-1]$, then $P[1:j-k_j-1] = T[i+k_j:i+j-2]$ holds for $k_j = \mathit{KMP\_Shift}[j]$. Moreover, there is no positive integer $k < \mathit{KMP\_Shift}[j]$ such that $P = T[i+k:i+k+m-1]$.*

Note that if the algorithm has confirmed $T[i:i+m-1] = P$, the shift is given by $\mathit{KMP\_Shift}[m+1]$ after reporting the occurrence of the pattern. Algorithm 1 shows a pseudocode to compute the array *KMP_Shift*. It runs in $O(m)$ time. By using *KMP_Shift*, the KMP algorithm finds all occurrences of P in T in $O(n)$ time.

2.4 HASHq algorithm

The HASHq algorithm [19] is an adaptation of the Wu-Manber multiple string matching algorithm [22] to the single string matching problem. Before comparing P and $T[i : i+m-1]$, the HASHq algorithm shifts the pattern so that the suffix q-gram $T[i+m-q : i+m-1]$ of the text substring shall match the rightmost occurrence of the same q-gram in the pattern. For practical efficiency, we use a hash function, though it may result in aligning mismatching q-grams occasionally. The shift amount is given by $\mathit{shift}(h(T[i+m-q : i+m-1]))$ where

$$\mathit{shift}(c) = m - \max(\{j \mid h(P[j-q+1 : j]) = c,\ q \le j \le m\} \cup \{q-1\}),$$
$$h(x) = (2^{q-1} \cdot x[1] + 2^{q-2} \cdot x[2] + \cdots + 2 \cdot x[q-1] + x[q]) \bmod 2^8.$$

We repeatedly shift the pattern till the suffix q-grams of the pattern and the considered text substring have a matching hash value, in which case the shift amount will be 0. We then compare the characters of the pattern and the text substring from left to right. If a character mismatch occurs during the comparison, the pattern is shifted by

$$\min(\{\, k \mid h(P[m'-k : m-k]) = h(P[m' : m]),\ 1 \le k \le m-q \,\} \cup \{m'\}) \tag{1}$$

where $m' = m-q+1$, since the q-gram suffixes of the pattern and the text substring have the same hash values. The time complexity of the preprocessing phase for computing the shift function is $O(mq)$. The searching phase has $O(n(m+q))$ time complexity. The worst-case time complexity is worse than that of the naive method, but it works fast in practice.

▶ **Fact 2.** *If* $\mathit{shift}(h(T[i+m-q : i+m-1])) = j \neq m-q+1$, *then* $h(P[m-j-q+1 : m-j]) = h(T[i+m-q : i+m-1])$. *There is no positive integer* $k < j$ *such that* $P = T[i+k : i+k+m-1]$.

3 Proposed algorithms

3.1 DISTq algorithm

Our proposed algorithm uses three kinds of shifting functions. The first one *HQ_Shift* is essentially the same as *shift*, the one used in the HASHq algorithm, except for the hashing function. The second one *dist* is based on the distance of the closest occurrences of the q-grams of the same hash value in the pattern. We involve *KMP_Shift* as the third one to guarantee the linear-time behavior.

Formally, the first shifting function is given as

$$\mathit{HQ\_Shift}[c] = m - \max(\{j \mid h(P[j-q+1 : j]) = c,\ q \le j \le m\} \cup \{q-1\}),$$
$$\text{where}\ \ h(x) = (4^{q-1} \cdot x[1] + 4^{q-2} \cdot x[2] + \cdots + 4 \cdot x[q-1] + x[q]) \bmod 2^{16}.$$

Fact 2 holds for *HQ_Shift*.

The second shifting function is defined for $j = q, \ldots, m$ by

$$\mathit{dist}[j] = \min(\{\, k \mid h(P[j'-k : j-k]) = h(P[j' : j]),\ 1 \le k \le j-q \,\} \cup \{j'\})$$

where $j' = j-q+1$. This function *dist* is a generalization of the shift (Eq. 1) used in the HASHq algorithm. We have $\mathit{dist}[j] = k < j'$ if the q-gram ending at j and the one ending at $j-k$ have the same hash value, while no q-grams occurring between those have the same value. If no q-gram ending before j has the same hash value, then $\mathit{dist}[j] = j'$. By using this, in the situation where $h(P[j-q+1 : j]) = h(T[i+j-q : i+j-1])$, when a mismatch occurs anywhere between $T[i : i+m-1]$ and P, the pattern can be shifted by $\mathit{dist}[j]$.

Algorithm 2 Computing *HQ_Shift*.

```
1  Function PreHqShift(P, q)
2      m ← |P|;
3      for i ← 0 to 2^16 − 1 do
4          HQ_Shift[i] ← m − q + 1;
5      for i ← q to m do
6          hash ← h(P[i − q + 1 : i]);
7          HQ_Shift[hash] ← m − i;
8      return HQ_Shift;
```

Algorithm 3 Computing *dist*.

```
1  Function PreDistArray(P, q)
2      for j ← 1 to q − 1 do
3          dist[j] ← 1;
4      for j ← 0 to 2^16 − 1 do
5          prevpos[j] ← 0;
6      for j ← q to |P| do
7          hash ← h(P[j − q + 1 : j]);
8          if prevpos[hash] = 0 then d ← j − q + 1;
9          else d ← j − prevpos[hash];
10         dist[j] ← d; prevpos[hash] ← j;
11     return dist;
```

▶ **Fact 3.** *Suppose that* $h(P[j-q+1:j]) = h(T[i+j-q:i+j-1])$. *Then* $h(P[j-q+1-dist[j]:j-dist[j]]) = h(T[i+j-q:i+j-1])$, *unless* $dist[j] = j-q+1$. *Moreover, there is no positive integer* $k < dist[j]$ *such that* $P = T[i+k:i+k+m-1]$.

Those functions *HQ_Shift*, *dist* and *KMP_Shift* are computed in the preprocessing phase. Algorithms 2 and 3 compute the arrays *HQ_Shift* and *dist*, respectively.

Figure 1 shows examples of shifting the pattern using *HQ_Shift* and *dist*. Both functions *HQ_Shift* and *dist* shift the pattern using q-gram hash values based on Facts 2 and 3, respectively. The latter can be used only when we know that the pattern and the text substring have aligned q-grams ending at j with the same hash value and it may shift the pattern at most $j-q+1$, while the former can be used anytime and the maximum possible shift is $m-q+1$. The advantage of the function *dist* is in the computational cost. If we know that the premise of Fact 3 is satisfied, we can immediately perform the shift based on *dist*, while computing $HQ\_Shift(h(w))$ for the concerned q-gram w in the text is not as cheap as $dist[j]$. Our algorithm exploits this advantage of the new shifting function *dist*.

Next, we explain our searching algorithm shown in Algorithm 4. The searching phase is divided into three: *Alignment-phase*, *Comparison-phase*, and *KMP-phase*. The goal of the Alignment-phase is to shift the pattern as far as possible without comparing each single character of the pattern and the text. The Alignment-phase ends when we align the pattern and a text substring that have (a) aligned q-grams of the same hash value and (b) the same first character. Suppose P and $T[k-m+1:k]$ are aligned at the beginning of the Alignment-phase. If $s = HQ\_Shift[h(T[k-q+1:k])] \leq m-q$, by shifting the pattern by s, we find the

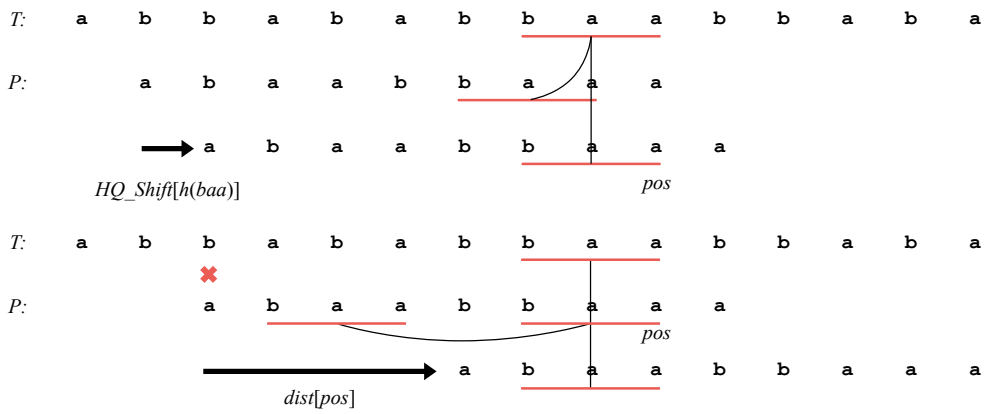


Figure 1 Shifting a pattern using *HQ_Shift* and *dist*.

aligned q-grams of the same hash value. Namely, $h(P[m-q-s+1:m-s]) = h(T[k-q+1:k])$. Otherwise, we shift the pattern by $m-q+1$ repeatedly until we find aligned q-grams of the same hash value. When finding a position *pos* satisfying (a) by aligning P and $T[k'-m+1:k']$ for some k', i.e., $h(P[pos-q+1:pos]) = h(T[k'-m+pos-q+1:k'-m+pos])$, we simply check the condition (b). If $P[1]$ and the corresponding text character match, we move to the Comparison-phase. Otherwise, we safely shift the pattern using $dist[pos]$. Note that although it is possible to use the function $HQ\_Shift(h(T[k'-q+1:k']))$ rather than $dist[pos]$, the computation would be more expensive. Shifting the pattern by $dist[pos]$, unless $dist[pos] = pos-q+1$, still the pattern and the aligned text substring satisfy (a). However, we do not repeat *dist*-shift any more, since the smaller *pos* becomes, the smaller the expected shift amount will be. We simply restart the Alignment-phase. Once the conditions (a) and (b) are satisfied, we move on to the Comparison-phase.

In the Comparison-phase, we check the characters from $P[2]$ to $P[m]$. If a character mismatch occurs during the comparison, either of the shift by *KMP_Shift* or by *dist* is possible. Therefore, we select the one where the resumption position of the character comparison goes further to the right after shifting the pattern. If the resumption position of the comparison is the same, we select the one with the larger shift amount. Recall that when the KMP algorithm finds that $P[1:j-1] = T[i:i+j-2]$ and $P[j] \neq T[i+j-1]$, it resumes comparison from checking the match between $T[i+j-1]$ and $P[j-KMP\_Shift[j]]$ if $KMP\_Shift[j] < j$, and $T[i+j]$ and $P[1]$ if $KMP\_Shift[j] = j$. On the other hand, if we shift the pattern by $dist[pos]$, we simply resume matching $T[i+dist[pos]]$ and $P[1]$. Therefore, we should use $KMP\_Shift[j]$ rather than $dist[pos]$ when either $KMP\_Shift[j] < j$ and $dist[pos] < j-1$ or $KMP\_Shift[j] = j > dist[pos]$. Summarizing the discussion, we shift the pattern by $dist[pos]$ if $dist[pos] \geq j-1$ and $dist[pos] \geq KMP\_Shift[j]$ hold. Otherwise, we shift the pattern by $KMP\_Shift[j]$. At this moment, we may have a "*partial match*" between the pattern and the aligned text substring. If we have performed the KMP-shift with $KMP\_Shift[j] < j-1$, then we have a match between the nonempty prefixes of the pattern and the aligned text substring of length $j - KMP\_Shift[j] - 1$. In this case, we go to the KMP-phase, where we simply perform the KMP algorithm. The KMP-phase prevents the character comparison position from returning to the left and guarantees the linear time behavior of our algorithm. If we have no partial match, we return to the Alignment-phase.

▶ **Theorem 1.** *The worst-case time complexity of the DISTq algorithm is* $O((n+m)q)$.

Algorithm 4 DISTq algorihm.

```
1  Function DISTq(P, T, q)
2      KMP_Shift ← PreKMPShift(P);
3      HQ_Shift ← PreHqShift(P, q);
4      dist ← PreDistArray(P, q);
5      n ← |T|; m ← |P|; i ← 1; j ← 1; k ← m;
6      while k ≤ n do
7          if j ≤ 1 then
8              while True do                                   // Alignment-phase
9                  sh ← HQ_Shift[h(T[k − m + 1 : k])]; k ← k + sh;
10                 if sh ≠ m − q + 1 then
11                     pos ← m − 1 − sh;
12                     if P[1] = T[k − m + 1] then break;
13                     k ← k + dist[pos];
14                 if k > n then halt;
15             j ← 2; i ← k − m + 2;                           // Comparison-phase
16             while j ≤ m and P[j] = T[i] do
17                 i ← i + 1; j ← j + 1;
18             if j = m + 1 then
19                 output i − m;
20             if dist[pos] ≥ j − 1 and dist[pos] ≥ KMP_Shift[j] then
21                 j ← j − dist[pos];
22             else
23                 j ← j − KMP_Shift[j];
24         else
25             while j ≤ m and P[j] = T[i] do                  // KMP-phase
26                 i ← i + 1; j ← j + 1;
27             if j = m + 1 then
28                 output i − m;
29             j ← j − KMP_Shift[j];
30         k ← i + m − j;
```

Proof. Since the proposed algorithm uses Fact 1 on the KMP algorithm to prevent the character comparison position from going back to the left, the number of character comparisons is at most $2n - m$ times like the KMP algorithm. In addition, the hash value of q-gram is calculated to perform the shift using *HQ_Shift*. Since the hash value calculation requires $O(q)$ time and it is calculated at the maximum of $n - q + 1$ places in the text, the hash value calculation takes $O(nq)$ time in total. Therefore, the worst-case time complexity of the searching phase is $O(nq)$. In the preprocessing, $O(mq)$ time is required to calculate the hash value of q-gram at $m - q + 1$ locations. ◀

▶ **Example 2.** Let $P =$ `abaabbaaa`. The shifting functions *dist*, *KMP_Shift*, and *HQ_Shift* are shown below. The hash values are calculated by treating each character as its ASCII value, e.g. `a` is calculated as 97.

j	1	2	3	4	5	6	7	8	9	10
P	a	b	a	a	b	b	a	a	a	
dist	-	-	1	2	3	4	5	4	7	
KMP_Shift	1	1	3	2	4	3	7	6	7	8

x	aba	baa	aab	abb	bba	aaa	others
$h(x)$	2041	2053	2038	2042	2057	2037	
HQ_Shift$[h(x)]$	6	1	4	3	2	0	7

Figure 2 illustrates an example run of the DISTq algorithm ($q = 3$) for finding $P =$ abaabbaaa in $T =$ abbaabbaababbabbaaabaabaabbaaa.

Attempt 1 We shift the pattern by one character for *HQ_Shift*$[h(T[7 : 9])] =$ *HQ_Shift*$[h($baa$)] =$ *HQ_Shift*$[2053] = 1$. Since the position of the q-gram aligned by this shift is 8, *pos* is updated to 8.

Attempt 2 We check whether the first character of the pattern matches the corresponding character of the text. Finding $P[1] \neq T[2]$, the pattern is shifted by *dist*$[pos] =$ *dist*$[8] = 4$.

Attempt 3 We shift the pattern by *HQ_Shift*$[h(T[12 : 14])] =$ *HQ_Shift*$[h($bba$)] = 2$ and update *pos* to 7.

Attempt 4 We check whether the first character of the pattern matches the corresponding character of the text. From $P[1] = T[8]$, we compare the characters of $P[2 : 9]$ and $T[9 : 16]$ from left to right. Since $P[2] \neq T[9]$, the pattern is shifted by *KMP_Shift*$[2]$ or *dist*$[pos] =$ *dist*$[7]$. From *KMP_Shift*$[2] = 1$, *dist*$[7] = 5$, *dist*$[pos] \geq 2 - 1$ and *dist*$[pos] \geq$ *KMP_Shift*$[2]$ are satisfied. Therefore, we shift the pattern by *dist*$[pos] =$ *dist*$[7] = 5$.

Attempt 5 We shift the pattern by *HQ_Shift*$[h(T[19 : 21])] =$ *HQ_Shift*$[h($aba$)] =$ *HQ_Shift*$[2041] = 6$ and update *pos* to 3.

Attempt 6 We check whether the first character of the pattern matches the corresponding character of the text. By $P[1] = T[19]$, the characters of $P[2 : 9]$ and $T[20 : 27]$ are compared from left to right. Since $P[6] \neq T[24]$, the pattern is shifted by *KMP_Shift*$[6]$ or *dist*$[pos] =$ *dist*$[3]$. By *KMP_Shift*$[6] = 3 >$ *dist*$[3] = 1$, the pattern is shifted by *KMP_Shift*$[6] = 3$.

Attempt 7 Attempt 6 shows that $P[1 : 2] = T[22 : 23]$, that is, there is a partial match, so we continue the comparison of $T[24 : 30]$ and $P[3 : 9]$. Since $T[24 : 30] = P[3 : 9]$, the pattern occurrence position 22 is reported.

3.2 LDISTq algorithm

The LDISTq algorithm modifies the DISTq algorithm so that the worst-case time complexity is independent of q. In the DISTq algorithm, if strings such as $T =$ a^n and $P =$ ba^{m-1} are given, $O(nq)$ time is required for searching phase because the hash values of each q-gram are calculated in the text. Since the hash function h defined in Section 3.1 is a rolling hash, if the hash value of $w[i : i + q - 1]$ has already been obtained for a string w, the hash value of $w[i + 1 : i + q]$ can be computed in constant time by $h(w[i + 1 : i + q]) = (4 \cdot (h(w[i : i + q - 1]) - 4^{q-1} \cdot w[i]) + w[i + q]) \bmod 2^{16}$. The LDIST$q$ algorithm modifies Line 9 of Algorithm 4 so that we calculate the hash value of the q-gram using the previously calculated value of the other q-gram in the incremental way, if they overlap. Similarly, the time complexity of the preprocessing phase can be reduced.

▶ **Theorem 3.** *The worst-case time complexity of the LDISTq algorithm is* $O(n + m)$.

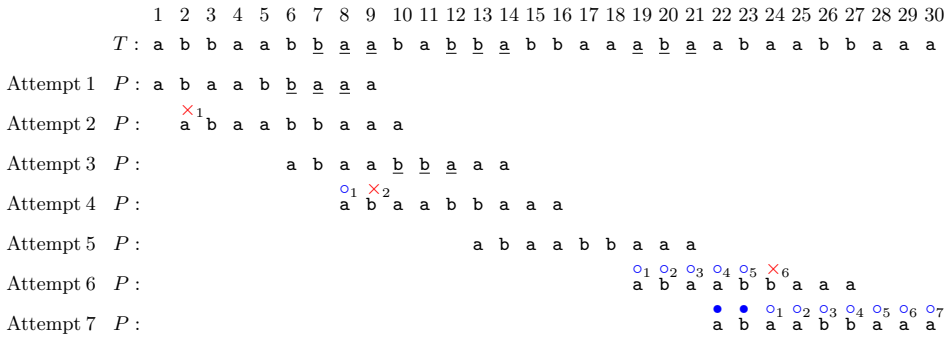


Figure 2 An example run of the DISTq algorithm for a pattern $P =$ `abaabbaaa` and a text $T =$ `abbaabbaababbabbaaabaabaabbaaa`. For each alignment of the pattern, ○ and × indicate a match and a mismatch between the text and the pattern, respectively. The character with ● is known to match the character at the corresponding position in the text without comparison. Subscript numbers show the order of character comparisons in each attempt.

Proof. Like the DISTq algorithm, we compare characters at most $2n - m$ times. To calculate the hash value of a q-gram, if it is overlapped with the q-gram for which the hash value has been calculated one step before, the incremental update is performed using the rolling hash. Therefore, the calculation of the hash value of q-grams takes $O(n)$ time in total. Thus, the worst-case time complexity of matching is $O(n)$. Calculating the hash values of q-grams in the preprocess is performed in the same way, so it is done in $O(m)$ time. ◀

4 Experiments

In this section, we compare the execution times of the proposed algorithms with the existing algorithms listed below, where algorithms that run in linear time in the input string size are marked with ⋆.

- BNDMq [20]: Backward Nondeterministic DAWG Matching algorithm using q-grams with $q = 2, 4$ and 6,
- SBNDMq [1]: Simplified version of the Backward Nondeterministic DAWG Matching algorithm using q-grams with $q = 2, 4, 6$ and 8,
- KBNDM [7]: Factorized variant of the BNDM algorithm,
- BSDMq [10]: Backward SNR DAWG Matching algorithm using condensed alphabets with groups of q characters with $1 \leq q \leq 8$,
- ⋆FJS [14]: Franek-Jennings-Smyth algorithm,
- ⋆FJS+ [18]: Modification of the FJS algorithm,
- HASHq [19]: Hashing algorithm using q-grams with $2 \leq q \leq 8$ (see Section 2.4),
- FS-w [11]: Multiple Windows version of the Fast Search algorithm [5] implemented using w sliding windows with $w = 1, 2, 4, 6$ and 8,
- IOM [6]: Improved Occurrence Matcher,
- WOM [6]: Worst Occurrence Matcher,
- SKIPq [9]: Skip-Search algorithm using q-grams with $2 \leq q \leq 8$,
- WFRq [8]: Weak-Factors-Recognition algorithm implemented with a q-chained loop with $2 \leq q \leq 8$,
- ⋆LWFRq [8]: Linear-Weak-Factors-Recognition algorithm implemented with a q-chained loop with $2 \leq q \leq 8$,
- ⋆DISTq: Our algorithm proposed in Section 3.1 (Algorithm 4) with $1 \leq q \leq 8$,
- ⋆LDISTq: Our algorithm proposed in Section 3.2 with $1 \leq q \leq 8$.

All algorithms are implemented in C language, compiled by GCC 9.2.0 with the optimization option `-O3`. We used the implementations in SMART [13] for all algorithms except for the FJS, FJS+ and our algorithms. The implementations of our algorithms are available at `https://github.com/ushitora/distq`. We experimented with the following strings:

1. Genome sequence (Table 1): the genome sequence of *E. coli* of length $n = 4641652$ with $\sigma = 4$, from NCBI[1]. The patterns are randomly extracted from T of length $m = 2, 4, 8, 16, 32, 64, 128, 256, 512$ and 1024. We measured the total running time of 25 executions.
2. English text (Table 2): the King James version of the Bible of length $n = 4017009$ with $\sigma = 62$, from the Large Canterbury Corpus[2] [2]. We removed the line breaks from the text. The patterns are randomly extracted from T of length $m = 2, 4, 8, 16, 32, 64, 128, 256, 512$ and 1024. We measured the total running time of 25 executions.
3. Fibonacci string (Table 3): generated by the following recurrence

   $$Fib_1 = \mathtt{b},\ Fib_2 = \mathtt{a} \text{ and } Fib_k = Fib_{k-1} \cdot Fib_{k-2} \text{ for } k > 2.$$

   The text is fixed to $T = Fib_{32}$ of length $n = 2178309$. The patterns are randomly extracted from T of length $m = 2, 4, 8, 16, 32, 64, 128, 256, 512$ and 1024. We measured the total running time of 100 executions.
4. Texts with frequent pattern occurrences (Tables 4, 5): generated by intentionally embedding a lot of patterns. We embedded $occ = 0, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536$, and 131072 occurrences of a pattern of length $m = 8$ into a text of length $n = 4000000$ over an alphabet of size $\sigma = 4$ and 95. More specifically, we first randomly generate a pattern and a provisional text, which may contain the pattern. Then we randomly change characters of the text until the pattern does not occur in the text. Finally we embed the pattern occ times at random positions without overlapping. We measured the total running time of 25 executions.

The best performance among three trials is recorded for each experiment. For the algorithms using parameter q or w, we report only the best results. The value of q or w giving the best performance is shown in round brackets.

Experimental results show that when the pattern is short, the SBNDMq and BSDMq algorithms have good performance in general. For the genome sequence text, WFRq, LWFRq and our algorithms are the fastest algorithms except when the pattern is very short. On the English text, SBNDMq and LWFRq run fastest for short and long patterns, respectively. On the other hand, DISTq runs almost as fast as the best algorithm on both short and long patterns. In fact, it runs faster than SBDDMq and LWFRq for long and short patterns, respectively. In the experiments on the Fibonacci string, the FJS algorithm and our algorithms have shown good results as the pattern length increases. Differently from the previous two sorts of texts, our algorithms clearly outperformed the LWFRq algorithm. Since the Fibonacci strings have many repeating structures and patterns are randomly extracted from the text, the number of occurrences of the pattern is very large in this experiment. Therefore, we hypothesize that the efficiency of DISTq algorithms does not decrease when the number of pattern occurrences is large. We fixed the pattern length and alphabet size and prepared data with the number of pattern occurrences intentionally changed. From the experimental result, it is found that our algorithms become more advantageous as the number of pattern occurrences increases. The results show that the LDISTq algorithm is generally slower than the DISTq algorithm. This should be due to the overhead of the process of determining whether to update the hash value difference by the rolling hash in the LDISTq algorithm.

[1] `https://www.ncbi.nlm.nih.gov/genome/167?genome_assembly_id=161521`
[2] `http://corpus.canterbury.ac.nz/`

Table 1 Genome sequence ($\sigma = 4$, $n = 4641652$).

m	2	4	8	16	32	64	128	256	512	1024
BNDMq	218.34$^{(2)}$	174.55$^{(2)}$	84.14$^{(4)}$	64.89$^{(6)}$	60.39$^{(6)}$	61.07$^{(6)}$	60.85$^{(6)}$	62.17$^{(6)}$	61.20$^{(6)}$	60.16$^{(6)}$
SBNDMq	**179.90$^{(2)}$**	154.20$^{(2)}$	80.94$^{(4)}$	73.29$^{(6)}$	57.10$^{(8)}$	61.01$^{(6)}$	61.74$^{(6)}$	61.20$^{(6)}$	61.67$^{(6)}$	60.80$^{(6)}$
KBNDM	311.78	201.99	150.15	113.84	83.23	67.83	75.65	75.39	76.58	74.72
BSDMq	195.38$^{(2)}$	**118.86$^{(3)}$**	84.23$^{(5)}$	63.03$^{(6)}$	61.26$^{(6)}$	58.75$^{(6)}$	57.50$^{(7)}$	56.99$^{(6)}$	57.01$^{(6)}$	56.58$^{(6)}$
⋆FJS	407.02	353.60	311.96	279.13	308.42	297.00	266.12	317.79	317.34	296.19
⋆FJS+	388.44	296.70	203.03	171.17	149.52	136.59	128.55	130.39	122.51	112.99
HASHq	571.44$^{(2)}$	272.86$^{(3)}$	126.00$^{(3)}$	88.12$^{(3)}$	68.22$^{(3)}$	58.84$^{(6)}$	55.09$^{(6)}$	59.48$^{(7)}$	57.34$^{(7)}$	57.96$^{(7)}$
FS-w	332.32$^{(4)}$	245.99$^{(4)}$	184.72$^{(4)}$	158.72$^{(4)}$	143.79$^{(6)}$	125.05$^{(4)}$	123.52$^{(6)}$	117.90$^{(6)}$	108.03$^{(6)}$	100.72$^{(6)}$
IOM	377.25	275.36	215.72	220.97	219.86	218.12	210.61	221.31	230.15	211.69
WOM	381.54	301.46	220.34	182.30	166.27	143.24	136.20	133.75	127.40	114.74
SKIPq	250.89$^{(2)}$	136.18$^{(3)}$	91.51$^{(4)}$	63.96$^{(6)}$	56.79$^{(7)}$	53.09$^{(7)}$	52.10$^{(7)}$	57.12$^{(7)}$	56.97$^{(8)}$	58.00$^{(6)}$
WFRq	219.50$^{(2)}$	168.39$^{(2)}$	88.86$^{(4)}$	65.82$^{(4)}$	57.19$^{(8)}$	55.12$^{(2)}$	51.77$^{(3)}$	50.04$^{(3)}$	55.02$^{(6)}$	**54.64$^{(8)}$**
⋆LWFRq	216.10$^{(2)}$	173.48$^{(3)}$	88.71$^{(4)}$	60.75$^{(5)}$	**53.84$^{(5)}$**	**50.48$^{(6)}$**	**49.65$^{(8)}$**	48.71$^{(6)}$	54.90$^{(6)}$	54.97$^{(7)}$
⋆DISTq	186.10$^{(2)}$	125.44$^{(3)}$	**78.56$^{(4)}$**	**60.48$^{(5)}$**	55.21$^{(6)}$	52.05$^{(7)}$	51.26$^{(8)}$	50.44$^{(8)}$	54.81$^{(7)}$	55.58$^{(8)}$
⋆LDISTq	295.55$^{(2)}$	181.99$^{(3)}$	86.58$^{(4)}$	65.29$^{(6)}$	56.74$^{(6)}$	52.31$^{(6)}$	50.39$^{(6)}$	**48.70$^{(7)}$**	**54.33$^{(4)}$**	55.22$^{(7)}$

Table 2 English text ($\sigma = 62$, $n = 4017009$).

m	2	4	8	16	32	64	128	256	512	1024
BNDMq	140.48$^{(2)}$	93.39$^{(2)}$	70.84$^{(4)}$	54.10$^{(4)}$	**45.88$^{(4)}$**	47.61$^{(4)}$	47.64$^{(4)}$	46.88$^{(4)}$	47.40$^{(6)}$	45.58$^{(4)}$
SBNDMq	**108.32$^{(2)}$**	**73.50$^{(2)}$**	**64.87$^{(2)}$**	**50.73$^{(4)}$**	47.85$^{(4)}$	48.45$^{(4)}$	48.55$^{(4)}$	47.56$^{(4)}$	47.08$^{(4)}$	45.98$^{(4)}$
KBNDM	192.76	126.56	92.03	71.04	64.17	56.79	49.56	49.09	50.24	47.68
BSDMq	117.22$^{(2)}$	79.96$^{(2)}$	70.36$^{(3)}$	57.72$^{(4)}$	49.91$^{(8)}$	48.00$^{(8)}$	44.99$^{(8)}$	43.62$^{(8)}$	43.06$^{(8)}$	42.70$^{(8)}$
⋆FJS	192.52	158.61	106.40	89.04	78.43	68.14	64.80	61.15	60.48	55.38
⋆FJS+	196.88	155.86	101.77	77.74	67.40	60.31	54.77	52.44	51.62	47.74
HASHq	312.91$^{(2)}$	218.95$^{(2)}$	107.38$^{(2)}$	70.85$^{(2)}$	56.90$^{(3)}$	49.33$^{(6)}$	46.63$^{(5)}$	45.27$^{(8)}$	44.77$^{(3)}$	42.98$^{(7)}$
FS-w	129.50$^{(6)}$	104.45$^{(6)}$	71.57$^{(6)}$	61.08$^{(4)}$	54.19$^{(6)}$	49.53$^{(4)}$	48.95$^{(6)}$	47.02$^{(4)}$	46.96$^{(6)}$	43.00$^{(4)}$
IOM	193.37	148.51	104.36	86.22	75.09	69.08	63.63	60.62	58.83	51.76
WOM	199.23	153.81	112.45	86.61	75.26	62.80	60.54	62.74	58.91	55.65
SKIPq	161.38$^{(2)}$	100.52$^{(2)}$	72.33$^{(3)}$	55.71$^{(4)}$	49.06$^{(4)}$	48.73$^{(4)}$	49.87$^{(4)}$	48.81$^{(8)}$	45.75$^{(2)}$	45.32$^{(2)}$
WFRq	137.40$^{(2)}$	85.22$^{(2)}$	68.88$^{(2)}$	52.85$^{(5)}$	46.67$^{(5)}$	**44.39$^{(8)}$**	**42.09$^{(8)}$**	41.66$^{(5)}$	41.65$^{(6)}$	40.53$^{(2)}$
⋆LWFRq	121.08$^{(2)}$	85.47$^{(2)}$	70.38$^{(2)}$	53.83$^{(3)}$	47.89$^{(5)}$	44.49$^{(5)}$	42.38$^{(6)}$	**41.09$^{(8)}$**	**41.23$^{(8)}$**	**40.30$^{(8)}$**
⋆DISTq	115.61$^{(2)}$	80.14$^{(2)}$	65.84$^{(3)}$	52.25$^{(4)}$	48.13$^{(4)}$	46.26$^{(4)}$	43.32$^{(4)}$	42.84$^{(8)}$	42.98$^{(4)}$	41.47$^{(7)}$
⋆LDISTq	229.62$^{(2)}$	102.15$^{(2)}$	69.70$^{(3)}$	55.34$^{(4)}$	49.46$^{(5)}$	45.93$^{(5)}$	44.37$^{(3)}$	43.15$^{(4)}$	42.21$^{(5)}$	41.11$^{(7)}$

Table 3 Fibonacci string ($\sigma = 2$, $n = 2178309$).

m	2	4	8	16	32	64	128	256	512	1024
BNDMq	343.25$^{(2)}$	308.44$^{(2)}$	283.26$^{(4)}$	257.64$^{(6)}$	233.94$^{(6)}$	285.63$^{(4)}$	284.37$^{(4)}$	293.00$^{(4)}$	307.82$^{(4)}$	315.47$^{(4)}$
SBNDMq	**286.02$^{(2)}$**	**292.15$^{(2)}$**	272.98$^{(4)}$	276.35$^{(6)}$	306.42$^{(6)}$	372.03$^{(6)}$	432.53$^{(6)}$	493.20$^{(6)}$	546.94$^{(6)}$	602.09$^{(6)}$
KBNDM	541.70	405.78	411.08	422.85	382.25	402.45	425.60	437.67	461.07	451.12
BSDMq	482.47$^{(2)}$	500.43$^{(3)}$	397.29$^{(5)}$	362.52$^{(8)}$	330.76$^{(6)}$	736.89$^{(1)}$	766.57$^{(1)}$	782.98$^{(1)}$	790.26$^{(1)}$	508.80$^{(3)}$
⋆FJS	402.44	362.23	276.97	237.87	218.38	**206.07**	203.86	202.94	196.49	194.01
⋆FJS+	456.20	396.04	335.70	319.93	295.64	300.36	296.48	295.37	288.56	289.00
HASHq	645.48$^{(2)}$	406.70$^{(2)}$	**257.69$^{(4)}$**	251.91$^{(7)}$	279.81$^{(7)}$	344.99$^{(7)}$	415.16$^{(7)}$	470.71$^{(7)}$	514.05$^{(7)}$	579.57$^{(7)}$
FS-w	383.23$^{(1)}$	396.23$^{(1)}$	347.72$^{(1)}$	289.15$^{(1)}$	253.36$^{(1)}$	246.82$^{(1)}$	248.73$^{(1)}$	235.66$^{(1)}$	235.35$^{(1)}$	230.61$^{(1)}$
IOM	381.92	414.42	453.54	497.84	543.93	641.13	751.42	839.92	899.19	1019.59
WOM	552.38	555.43	564.67	617.93	664.47	732.05	852.06	926.35	1036.31	1126.17
SKIPq	470.93$^{(2)}$	394.09$^{(2)}$	332.66$^{(5)}$	336.16$^{(8)}$	374.91$^{(8)}$	464.23$^{(3)}$	460.54$^{(3)}$	450.18$^{(3)}$	451.05$^{(3)}$	464.13$^{(3)}$
WFRq	442.32$^{(2)}$	497.95$^{(3)}$	528.45$^{(3)}$	652.48$^{(6)}$	2132.38$^{(7)}$	3762.19$^{(8)}$	6762.67$^{(8)}$	12624.63$^{(8)}$	24416.65$^{(8)}$	48596.02$^{(8)}$
⋆LWFRq	552.64$^{(2)}$	504.77$^{(3)}$	428.34$^{(5)}$	342.80$^{(7)}$	297.07$^{(7)}$	304.57$^{(2)}$	274.47$^{(6)}$	265.28$^{(6)}$	258.06$^{(6)}$	254.92$^{(6)}$
⋆DISTq	438.96$^{(1)}$	359.56$^{(2)}$	293.80$^{(2)}$	**235.61$^{(2)}$**	**213.07$^{(7)}$**	206.92$^{(2)}$	201.18$^{(8)}$	196.56$^{(5)}$	194.86$^{(4)}$	193.62$^{(5)}$
⋆LDISTq	565.13$^{(2)}$	412.15$^{(3)}$	300.98$^{(4)}$	245.38$^{(7)}$	215.89$^{(8)}$	208.40$^{(7)}$	**200.45$^{(4)}$**	**193.74$^{(4)}$**	**190.85$^{(3)}$**	**193.48$^{(8)}$**

Table 4 Texts with frequent pattern occurrences ($\sigma = 4$, $n = 4000000$, $m = 8$).

occ	0	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
BNDMq	73.53$^{(4)}$	72.28$^{(4)}$	72.87$^{(4)}$	73.89$^{(4)}$	74.39$^{(4)}$	75.75$^{(4)}$	77.60$^{(4)}$	81.63$^{(4)}$	82.67$^{(4)}$	88.98$^{(4)}$	108.56$^{(4)}$	146.13$^{(6)}$
SBNDMq	**68.58**$^{(4)}$	**68.92**$^{(4)}$	71.17$^{(4)}$	77.26$^{(4)}$	81.04$^{(4)}$	80.76$^{(4)}$	78.33$^{(4)}$	82.82$^{(4)}$	91.01$^{(4)}$	96.83$^{(4)}$	111.75$^{(4)}$	140.21$^{(4)}$
KBNDM	127.73	128.41	128.41	126.92	128.74	131.17	129.74	135.70	140.99	152.46	164.59	186.32
BSDMq	69.85$^{(4)}$	69.04$^{(4)}$	**68.85**$^{(4)}$	**69.19**$^{(4)}$	**70.63**$^{(4)}$	**72.00**$^{(4)}$	**72.48**$^{(4)}$	**76.25**$^{(4)}$	**79.95**$^{(4)}$	89.91$^{(4)}$	110.22$^{(4)}$	143.91$^{(4)}$
⋆FJS	246.26	253.35	263.94	247.73	255.94	276.44	262.83	256.36	251.33	269.39	259.93	251.06
⋆FJS+	174.38	173.86	180.46	173.05	178.65	182.98	177.56	181.17	182.48	190.71	197.69	199.60
HASHq	121.09$^{(3)}$	121.36$^{(3)}$	122.64$^{(3)}$	122.50$^{(3)}$	119.49$^{(3)}$	123.98$^{(3)}$	124.31$^{(3)}$	124.93$^{(3)}$	126.81$^{(3)}$	128.94$^{(3)}$	143.75$^{(3)}$	153.08$^{(4)}$
FS-w	154.89$^{(4)}$	155.56$^{(4)}$	165.18$^{(4)}$	156.67$^{(4)}$	159.03$^{(4)}$	166.44$^{(4)}$	165.62$^{(4)}$	160.46$^{(4)}$	167.46$^{(4)}$	178.53$^{(2)}$	185.86$^{(2)}$	191.38$^{(2)}$
IOM	185.31	181.07	195.53	187.98	194.18	200.99	195.98	195.89	197.67	202.61	214.90	209.52
WOM	192.59	192.28	207.57	189.21	196.02	203.86	196.98	199.79	203.59	215.79	219.94	230.59
SKIPq	79.32$^{(4)}$	77.14$^{(4)}$	79.19$^{(4)}$	82.08$^{(4)}$	81.82$^{(4)}$	82.55$^{(4)}$	83.83$^{(4)}$	87.08$^{(4)}$	89.82$^{(4)}$	93.09$^{(4)}$	106.41$^{(4)}$	126.22$^{(3)}$
WFRq	76.68$^{(4)}$	76.38$^{(4)}$	77.63$^{(4)}$	83.21$^{(4)}$	80.93$^{(4)}$	81.42$^{(4)}$	83.86$^{(4)}$	88.55$^{(4)}$	93.16$^{(4)}$	106.07$^{(4)}$	123.90$^{(4)}$	164.77$^{(4)}$
⋆LWFRq	76.99$^{(4)}$	76.33$^{(4)}$	78.83$^{(4)}$	77.57$^{(4)}$	78.74$^{(4)}$	76.46$^{(4)}$	84.65$^{(4)}$	89.87$^{(4)}$	96.17$^{(4)}$	108.67$^{(4)}$	129.35$^{(3)}$	168.70$^{(3)}$
⋆DISTq	69.67$^{(4)}$	73.38$^{(4)}$	74.35$^{(4)}$	74.31$^{(4)}$	75.12$^{(4)}$	74.96$^{(4)}$	77.21$^{(4)}$	77.56$^{(4)}$	80.09$^{(4)}$	**86.25**$^{(4)}$	**101.12**$^{(4)}$	**120.98**$^{(3)}$
⋆LDISTq	75.82$^{(4)}$	74.45$^{(4)}$	74.89$^{(4)}$	76.77$^{(4)}$	74.62$^{(4)}$	75.47$^{(4)}$	77.10$^{(4)}$	80.18$^{(4)}$	83.40$^{(4)}$	88.86$^{(4)}$	103.73$^{(4)}$	122.27$^{(4)}$

Table 5 Texts with frequent pattern occurrences ($\sigma = 95$, $n = 4000000$, $m = 8$).

occ	0	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
BNDMq	56.58$^{(2)}$	55.41$^{(2)}$	56.26$^{(2)}$	56.89$^{(2)}$	56.68$^{(2)}$	57.42$^{(2)}$	58.80$^{(2)}$	60.72$^{(2)}$	67.37$^{(2)}$	74.57$^{(2)}$	98.52$^{(2)}$	125.65$^{(2)}$
SBNDMq	**52.76**$^{(2)}$	**52.37**$^{(2)}$	**53.09**$^{(2)}$	**53.23**$^{(2)}$	55.56$^{(2)}$	**52.82**$^{(2)}$	56.41$^{(2)}$	**56.50**$^{(2)}$	61.58$^{(2)}$	69.96$^{(2)}$	91.72$^{(2)}$	112.41$^{(2)}$
KBNDM	81.98	80.22	77.61	79.42	79.55	81.88	82.61	82.25	89.19	98.65	111.59	142.65
BSDMq	54.89$^{(2)}$	56.07$^{(2)}$	54.55$^{(2)}$	55.99$^{(2)}$	55.78$^{(2)}$	56.78$^{(2)}$	58.38$^{(2)}$	61.48$^{(2)}$	66.76$^{(2)}$	76.66$^{(3)}$	96.79$^{(3)}$	131.76$^{(3)}$
⋆FJS	81.93	81.41	81.14	81.83	82.04	82.16	82.10	84.30	86.02	90.72	100.41	111.55
⋆FJS+	80.26	84.27	81.05	80.58	80.21	80.84	83.52	85.66	87.40	94.03	102.16	112.23
HASHq	101.34$^{(2)}$	103.34$^{(2)}$	102.15$^{(2)}$	105.02$^{(2)}$	101.55$^{(2)}$	104.07$^{(2)}$	105.84$^{(2)}$	107.46$^{(2)}$	107.79$^{(2)}$	112.15$^{(2)}$	118.52$^{(2)}$	126.53$^{(2)}$
FS-w	52.77$^{(8)}$	52.42$^{(6)}$	53.51$^{(6)}$	53.74$^{(6)}$	**53.07**$^{(6)}$	53.95$^{(6)}$	**55.33**$^{(6)}$	57.61$^{(6)}$	**61.36**$^{(6)}$	68.86$^{(6)}$	81.66$^{(4)}$	102.11$^{(2)}$
IOM	82.07	83.86	84.89	85.58	82.64	82.45	83.90	86.98	87.28	91.63	99.30	106.37
WOM	84.37	84.42	86.39	85.05	85.36	85.35	86.20	86.74	90.22	93.31	105.36	114.68
SKIPq	60.93$^{(2)}$	59.34$^{(2)}$	63.66$^{(3)}$	62.97$^{(3)}$	62.11$^{(2)}$	62.42$^{(2)}$	64.08$^{(2)}$	65.12$^{(2)}$	69.11$^{(2)}$	76.18$^{(3)}$	83.05$^{(3)}$	101.43$^{(3)}$
WFRq	59.39$^{(2)}$	59.57$^{(2)}$	59.24$^{(2)}$	60.12$^{(2)}$	61.16$^{(2)}$	55.51$^{(2)}$	58.24$^{(2)}$	62.38$^{(2)}$	68.35$^{(2)}$	81.99$^{(2)}$	104.92$^{(2)}$	139.15$^{(2)}$
⋆LWFRq	55.99$^{(2)}$	58.63$^{(2)}$	54.16$^{(2)}$	53.75$^{(2)}$	56.99$^{(2)}$	60.87$^{(2)}$	58.45$^{(2)}$	63.39$^{(2)}$	72.48$^{(2)}$	85.61$^{(2)}$	112.47$^{(3)}$	152.54$^{(3)}$
⋆DISTq	58.60$^{(2)}$	59.30$^{(2)}$	58.94$^{(2)}$	58.46$^{(2)}$	58.47$^{(3)}$	59.91$^{(2)}$	61.55$^{(2)}$	62.29$^{(2)}$	65.21$^{(2)}$	**65.99**$^{(2)}$	**77.36**$^{(2)}$	**95.05**$^{(2)}$
⋆LDISTq	62.56$^{(3)}$	61.52$^{(2)}$	66.62$^{(3)}$	66.90$^{(2)}$	67.28$^{(3)}$	63.66$^{(2)}$	65.22$^{(2)}$	67.15$^{(2)}$	67.59$^{(2)}$	73.93$^{(2)}$	85.10$^{(2)}$	100.33$^{(2)}$

5 Conclusion

We proposed two new algorithms for the exact string matching problem: the DISTq algorithm and the LDISTq algorithm. We confirmed that our algorithms are as efficient as the state-of-the-art algorithms in many cases. Particularly when a pattern frequently appears in a text, our algorithms outperformed existing algorithms. The DISTq algorithm runs in $O(q(n+m))$ time and the LDISTq algorithm runs in $O(n+m)$ time. Their performances were not significantly different in our experiments and rather the former ran faster than the latter in most cases, where the optimal value of q was relatively small.

References

1 Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Factor oracle: A new structure for pattern matching. In Jan Pavelka, Gerard Tel, and Miroslav Bartošek, editors, *SOFSEM'99: Theory and Practice of Informatics*, pages 295–310. Springer Berlin Heidelberg, 1999.

2 Ross Arnold and Tim Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of DCC '97. Data Compression Conference*, pages 201–210, 1997.

3 Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977. doi:10.1145/359842.359859.

4 Domenico Cantone and Simone Faro. Searching for a substring with constant extra-space complexity. In *Proceedings of Third International Conference on Fun with algorithms*, pages 118–131, 2004.

5 Domenico Cantone and Simone Faro. Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm. *Journal of Automata, Languages and Combinatorics*, 10:589–608, 2005. doi:10.1007/3-540-44867-5_4.

6 Domenico Cantone and Simone Faro. Improved and self-tuned occurrence heuristics. *Journal of Discrete Algorithms*, 28:73–84, 2014. doi:10.1016/j.jda.2014.07.006.

7 Domenico Cantone, Simone Faro, and Emanuele Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. *Information and Computation*, 213:3–12, 2012. Special Issue: Combinatorial Pattern Matching (CPM 2010). doi:10.1016/j.ic.2011.03.006.

8 Domenico Cantone, Simone Faro, and Arianna Pavone. Linear and Efficient String Matching Algorithms Based on Weak Factor Recognition. *Journal of Experimental Algorithmics*, 24(1):1–20, 2019. doi:10.1145/3301295.

9 Simone Faro. A very fast string matching algorithm based on condensed alphabets. In Riccardo Dondi, Guillaume Fertin, and Giancarlo Mauri, editors, *Algorithmic Aspects in Information and Management - 11th International Conference, AAIM 2016, Bergamo, Italy, July 18-20, 2016, Proceedings*, volume 9778 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2016. doi:10.1007/978-3-319-41168-2_6.

10 Simone Faro and Thierry Lecroq. A fast suffix automata based algorithm for exact online string matching. In Nelma Moreira and Rogério Reis, editors, *Implementation and Application of Automata*, pages 149–158. Springer Berlin Heidelberg, 2012.

11 Simone Faro and Thierry Lecroq. A multiple sliding windows approach to speed up string matching algorithms. In Ralf Klasing, editor, *Experimental Algorithms*, pages 172–183. Springer Berlin Heidelberg, 2012.

12 Simone Faro and Thierry Lecroq. The exact online string matching problem. *ACM Computing Surveys*, 45(2):1–42, 2013. doi:10.1145/2431211.2431212.

13 Simone Faro, Thierry Lecroq, Stefano Borzì, Simone Di Mauro, and Alessandro Maggio. The string matching algorithms research tool. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2016*, pages 99–113, Czech Technical University in Prague, Czech Republic, 2016.

14 Frantisek Franek, Christopher G. Jennings, and W.F. Smyth. A simple fast hybrid pattern-matching algorithm. *Journal of Discrete Algorithms*, 5(4):682–695, 2007. doi:10.1016/J.JDA.2006.11.004.

15 Saqib I. Hakak, Amirrudin Kamsin, Palaiahnakote Shivakumara, Gulshan A. Gilkar, Wazir Z. Khan, and Muhammad Imran. Exact string matching algorithms: Survey, issues, and future research directions. *IEEE Access*, 7:69614–69637, 2019.

16 R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980. doi:10.1002/spe.4380100608.

17 Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.

18 Satoshi Kobayashi, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. An improvement of the Franek-Jennings-Smyth pattern matching algorithm. In *Proceedings of the Prague Stringology Conference 2019*, pages 56–68, 2019.

19 Thierry Lecroq. Fast exact string matching algorithms. *Information Processing Letters*, 102(6):229–235, 2007. doi:10.1016/j.ipl.2007.01.002.

20 Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In Martin Farach-Colton, editor, *Combinatorial Pattern Matching*, pages 14–33. Springer Berlin Heidelberg, 1998.

21 Daniel M. Sunday and Daniel M. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990. doi:10.1145/79173.79184.

22 Sun Wu and Udi Manber. A fast algorithm for multi-pattern searching. Technical Report TR–94–17, Department of Computer Science, Chung-Cheng University, 1994.

Faster Fully Dynamic Transitive Closure in Practice

Kathrin Hanauer
University of Vienna, Faculty of Computer Science, Austria
kathrin.hanauer@univie.ac.at

Monika Henzinger
University of Vienna, Faculty of Computer Science, Austria
monika.henzinger@univie.ac.at

Christian Schulz
University of Vienna, Faculty of Computer Science, Austria
christian.schulz@univie.ac.at

Abstract

The fully dynamic transitive closure problem asks to maintain reachability information in a directed graph between arbitrary pairs of vertices, while the graph undergoes a sequence of edge insertions and deletions. The problem has been thoroughly investigated in theory and many specialized algorithms for solving it have been proposed in the last decades. In two large studies [Frigioni ea, 2001; Krommidas and Zaroliagis, 2008], a number of these algorithms have been evaluated experimentally against simple, static algorithms for graph traversal, showing the competitiveness and even superiority of the simple algorithms in practice, except for very dense random graphs or very high ratios of queries. A major drawback of those studies is that only small and mostly randomly generated graphs are considered.

In this paper, we engineer new algorithms to maintain all-pairs reachability information which are simple and space-efficient. Moreover, we perform an extensive experimental evaluation on both generated and real-world instances that are several orders of magnitude larger than those in the previous studies. Our results indicate that our new algorithms outperform all state-of-the-art algorithms on all types of input considerably in practice.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Dynamic Graph Algorithms, Reachability, Transitive Closure

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.14

Related Version A full version of the paper is available at [12], `https://arxiv.org/abs/2002.00813`.

Supplementary Material Source code and instances are available at `https://dyreach.taa.univie.ac.at/transitive-closure`.

Funding The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

1 Introduction

Complex graphs are useful in a wide range of applications from technological networks to biological systems like the human brain. These graphs can contain billions of vertices and edges. Analyzing these networks aids us in gaining new insights about our surroundings. One of the most basic questions that arises in this setting is whether one vertex can *reach* another vertex via a directed path. This simple problem has a wide range of applications such program analysis [28], protein-protein interaction networks [10], centrality measures, and is used as subproblem in a wide range of more complex (dynamic) algorithms such as

in the computation of (dynamic) maximum flows [8, 6, 11]. Often, the underlying graphs or input instances change over time, i.e., vertices or edges are inserted or deleted as time is passing. In a social network, for example, users sign up or leave, and relations between them may be created or removed over time. Terminology-wise, a problem is said to be *fully dynamic* if the update operations include both insertions *and* deletions of edges, and *partially dynamic* if only one type of update operations is allowed. In this context, a problem is called *incremental*, if only edge insertions occur, but no deletions, and *decremental* vice versa.

Recently, we studied an extensive set of algorithms for the *single-source* reachability problem in the fully dynamic setting [13]. The fully dynamic single-source reachability problem maintains the set of vertices that are reachable from a given *source vertex*, subject to edge deletions and insertions. In particular, we designed several fully dynamic variants of well-known approaches to obtain and maintain reachability information with respect to a distinguished source.

This yields the *starting point of this paper*: our goal was to transfer recently engineered algorithms for the fully dynamic *single-source* reachability problem [13] to the more general fully dynamic *transitive closure* problem (also known as fully dynamic all-pairs reachability problem). In contrast to the single-source problem, the *fully dynamic transitive closure* problem consists in maintaining reachability information between *arbitrary* pairs of vertices s and t in a directed graph, which in turn is subject to edge insertions and deletions. If the graph does not change, i.e., in the *static* setting, the question whether an arbitrary vertex s can reach another arbitrary vertex t can either be answered in linear time by starting a breadth-first or depth-first search from s, or it can be answered in constant time after the transitive closure of the graph, i.e., reachability information for all pairs of vertices, has been computed. The latter can be obtained in $\mathcal{O}(n^\omega)$, where ω is the exponent in the running time of the best known fast matrix multiplication algorithm (currently, $\omega < 2.38$ [24]), or combinatorially in $\mathcal{O}(n \cdot m)$ or $\mathcal{O}(n^3)$ time by either starting a breadth-first or depth-first search from each vertex or using the Floyd-Warshall algorithm [7, 38, 3].

In the dynamic setting, the aim is to avoid such costly recomputations from scratch after the graph has changed, especially if the update was small. Hence, the dynamic version of the problem has been thoroughly studied in theory and many specialized algorithms for solving it have been proposed in the last decades. However, even the insertion or deletion of a single edge may affect the reachability between $\Omega(n^2)$ vertex pairs, which is why one cannot hope for an algorithm with constant query time that processes updates in less than $\mathcal{O}(n^2)$ worst-case time if the transitive closure is maintained explicitly. Furthermore, conditional lower bounds [15] suggest that no faster solution than the naïve recomputation from scratch is possible after each change in the graph.

Whereas the static approach to compute the transitive closure beforehand via graph traversal can be readily adapted to the incremental setting, yielding an amortized update time of $\mathcal{O}(n)$ [17], a large number of randomized and deterministic algorithms [16, 18, 14, 20, 19, 21, 4, 5, 31, 29, 34, 27] has been devised over the last years for the decremental and the fully dynamic version of the problem. The currently fastest algorithm in the deletions-only case is deterministic, has a total update time of $\mathcal{O}(n \cdot m)$, and answers queries in constant time [27]. In the fully dynamic setting, updates can be processed deterministically in $\mathcal{O}(n^2)$ amortized time with constant query time [5], or, alternatively in $\mathcal{O}(m\sqrt{n})$ amortized update time with $\mathcal{O}(\sqrt{n})$ query time [31]. An almost exhaustive set of these algorithms has been evaluated experimentally against simple, static algorithms for graph traversal such as breadth-first or depth-first search in two large studies [9, 22]. Surprisingly, both have shown that the simple algorithms are competitive and even superior to the specialized

algorithms in practice, except for dense random graphs or, naturally, very high ratios of queries. Only two genuinely dynamic algorithms could challenge the simple ones: an algorithm developed by Frigioni et al. [9], which is based on Italiano's algorithms [17, 18] as well as an extension of a decremental Las Vegas algorithm proposed by Roditty and Zwick [31], developed by Krommidas and Zaroliagis [22]. Both rely on the computation and maintenance of strongly connected components, which evidently gives them an advantage on dense graphs. Nevertheless, they appeared to be unable to achieve a speedup of a factor greater than ten in comparison to breadth-first and depth-first search.

In this paper, we engineer a family of algorithms that build on recent experimental results for the single-source reachability problem [13]. Our algorithms are very easy to implement and benefit from strongly connected components likewise, although they do not (necessarily) compute them explicitly. In an extensive experimental evaluation on various types of input instances, we compare our algorithms to all simple algorithms from [9, 22] as well as a modified, bidirectional breadth-first search. The latter already achieves a speedup of multiple factors over the standard version, and our new algorithms outperform the simple algorithms on all types of input by several orders of magnitude in practice.

2 Preliminaries

Basic Concepts. Let $G = (V, E)$ be a directed graph with vertex set V and edge set E. Throughout this paper, let $n = |V|$ and $m = |E|$. The *density* of G is $d = \frac{m}{n}$. An edge $(u, v) \in E$ has *tail* u and *head* v and u and v are said to be *adjacent*. (u, v) is said to be an *outgoing* edge or *out-edge* of u and an *incoming* edge or *in-edge* of v. The *outdegree* $\deg^+(v)$/*indegree* $\deg^-(v)$/*degree* $\deg(v)$ of a vertex v is its number of (out-/in-) edges. The *out-neighborhood* (*in-neighborhood*) of a vertex u is the set of all vertices v such that $(u, v) \in E$ ($(v, u) \in E$). A sequence of vertices $s \to \cdots \to t$ such that each pair of consecutive vertices is connected by an edge is called an *s-t path* and s can *reach* t. A *strongly connected component* (*SCC*) is a maximal subset of vertices $X \subseteq V$ such that for each ordered pair of vertices $s, t \in X$, s can reach t. The *condensation* of a directed graph G is a directed graph G_C that is obtained by shrinking every SCC of G to a single vertex, thereby preserving edges between different SCCs. A graph is *strongly connected* if it has only one SCC. In case that each SCC is a singleton, i.e., G has n SCCs, G is said to be *acyclic* and also called a *DAG* (directed acyclic graph). The *reverse* of a graph G is a graph with the same vertex set as G, but contains for each edge $(u, v) \in E$ the reverse edge (v, u).

A *dynamic graph* is a directed graph G along with an ordered sequence of updates, which consist of edge insertions and deletions. In this paper, we consider the *fully dynamic transitive closure problem*: Given a directed graph, answer reachability queries between arbitrary pairs of vertices s and t, subject to edge insertions and deletions.

Related Work. Due to space limitations, we only give a brief overview over related work by listing the currently best known results for fully dynamic algorithms on general graphs in Table 1. For details and partially dynamic algorithms, see the full version [12].

In large studies, Frigioni et al. [9] as well as Krommidas and Zaroliagis [22] have implemented an extensive set of known algorithms for dynamic transitive closure and compared them to each other as well as to simple, static algorithms such as breadth-first and depth-first search. The set comprises the original algorithms in [17, 18, 39, 14, 19, 21, 29, 31, 33, 4, 5] as well as several modifications and improvements thereof. The experimental evaluations on random Erdős-Renyí graphs, instances constructed to be difficult on purpose, as well

Table 1 Currently best results for fully dynamic transitive closure. All running times are asymptotic ($\mathcal{O}$-notation).

Query Time	Update Time	
m	1	naïve
1	n^2	Demetrescu and Italiano [5], Roditty [29], Sankowski [34]
$\sqrt{n}$	$m\sqrt{n}$	Roditty and Zwick [31]
$m^{0.43}$	$m^{0.58}n$	Roditty and Zwick [31]
$n^{0.58}$	$n^{1.58}$,	Sankowski [34]
$n^{1.495}$	$n^{1.495}$	Sankowski [34]
n	$m + n \log n$	Roditty and Zwick [33]
$n^{1.407}$	$n^{1.407}$	van den Brand et al. [37]

as two instances based on real-world graphs, showed that on all instances except for dense random graphs or a query ratio of more than 65 %, the simple algorithms outperformed the dynamic ones distinctly and up to several factors. Their strongest competitors were the fully dynamic extension [9] of the algorithms by Italiano [17, 18], as well as the fully dynamic extension [22] of the decremental algorithm by Roditty and Zwick [31]. These two algorithms also were the only ones that were faster than static graph traversal on dense random graphs, by a factor of at most ten.

3 Algorithms

We propose a new and very simple approach to maintain the transitive closure in a fully dynamic setting. Inspired by a recent study on single-source reachability, it is based solely on single-source and single-sink reachability (SSR) information. Unlike most algorithms for dynamic transitive closure, it does not explicitly need to compute or maintain strongly connected components – which can be time-consuming – but, nevertheless, profits indirectly if the graph is strongly connected. Different variants and parameterizations of this approach lead to a family of new algorithms, all of which are easy to implement and – depending on the choice of parameters – extremely space-efficient.

In Section 4, we evaluate this approach experimentally against a set of algorithms that have been shown to be among the fastest algorithms in practice so far. This set includes the classic simple, static algorithms for graph traversal, breadth-first search and depth-first search. For the sake of completeness, we will start by describing the practical state-of-the-art algorithms, and then continue with our new approach. Each algorithm for fully dynamic transitive closure can be described by means of four subroutines: initialize(), insertEdge((u, v)), deleteEdge((u, v)), and query(s, t), which define the behavior during the initialization phase, in case that an edge (u, v) is added or removed, and how it answers a query of whether a vertex s can reach a vertex t, respectively.

Table 2 provides an overview of all algorithms in this section along with their abbreviations. All algorithms considered are combinatorial and either deterministic or Las Vegas-style randomized, i.e., their running time, but not their correctness, may depend on random variables.

3.1 Static Algorithms

In the static setting or in case that a graph is given without further (reachability) information besides its edges, breadth-first search (*BFS*) and depth-first search (*DFS*) are the two standard algorithms to determine whether there is a path between a pair of vertices or not. Despite their simplicity and the fact that they typically have no persistent memory (such as a cache, e.g.), experimental studies [9, 22] have shown them to be at least competitive with both partially and fully dynamic algorithms and even superior on various instances.

BFS, DFS. We consider both BFS and DFS in their pure versions: For each query(s, t), a new BFS or DFS, respectively, is initiated from s until either t is encountered or the graph is exhausted. The algorithms store or maintain no reachability information whatsoever and do not perform any work in initialize(), insertEdge$((u, v))$, or deleteEdge$((u, v))$. We refer to these algorithms simply as `BFS` and `DFS`, respectively.

In addition, we consider a hybridization of `BFS` and `DFS`, called `DBFS`, which was introduced originally by Frigioni et al. [9] and is also part of the later study [22]. In case of a query(s, t), the algorithm visits vertices in DFS order, starting from s, but additionally checks for each vertex that is encountered whether t is in its out-neighborhood.

Bidirectional BFS. To speed up static reachability queries even further, we adapted a well-established approach for the more general problem of finding shortest paths and perform two breadth-first searches alternatingly: Upon a query(s, t), the algorithm initiates a customary BFS starting from s, but pauses already after few steps, even if t has not been encountered yet. The algorithm then initiates a BFS on the reverse graph, starting from t, and also pauses after few steps, even if s has not been encountered yet. Afterwards, the first and the second BFS are resumed by turns, always for a few steps only, until either one of them encounters a vertex v that has already encountered by the other, or the graph is exhausted. In the former case, there is a path from s via v to t, hence the algorithm answers the query positively, and otherwise negatively. We refer to this algorithm as `BiBFS` (*Bidirectional BFS*) and use the respective out-degree of the current vertex in each BFS as step size, i.e., each BFS processes one vertex, examines all its out-neighbors, and then pauses execution. Note that the previous experimental studies [9, 22] do not consider this algorithm.

3.2 A New Approach

General Overview. Let v be an arbitrary vertex of the graph and let $R^+(v)$ and $R^-(v)$ be the sets of vertices reachable from v and that can reach v, respectively. To answer reachability queries between two vertices s and t, we use the following simple observations:

(O1) If $s \in R^-(v)$ and $t \in R^+(v)$, then s can reach t.
(O2) If v can reach s, but not t, i.e., $s \in R^+(v)$ and $t \notin R^+(v)$, then s cannot reach t.
(O3) If t can reach v, but s cannot, i.e., $s \notin R^-(v)$ and $t \in R^-(v)$, then s cannot reach t.

Whereas the first observation is widely used in several algorithms, we are not aware of any algorithms making direct use of the others. Our new family of algorithms keeps a list $\mathcal{L}_{SV}$ of length k of so-called *supportive vertices*, which work similarly to cluster centers in decremental shortest paths algorithms [32]. For each vertex v in $\mathcal{L}_{SV}$, there are two fully dynamic data structures maintaining the sets $R^+(v)$ and $R^-(v)$, respectively. In other words, these data structures maintain single-source as well as single-sink reachability for a vertex v. We give details on those data structures at the end of this section. All updates to the

graph, i.e., all notifications of insertEdge$(\cdot)$ and deleteEdge$(\cdot)$, are simply passed on to these data structures. In case of a query(s,t), the algorithms first check whether one of s or t is a supportive vertex itself. In this case, the query can be answered decisively using the corresponding data structure. Otherwise, the vertices in $\mathcal{L}_{SV}$ are considered one by one and the algorithms try to apply one of the above observations. Finally, if this also fails, a static algorithm serves as fallback to answer the reachability query.

Whereas this behavior is common to all algorithms of the family, they differ in their choice of supportive vertices and the subalgorithms used to maintain SSR information as well as the static fallback algorithm.

Note that it suffices if an algorithm has exactly one vertex v_i from each SCC C_i in $\mathcal{L}_{SV}$ to answer every reachability query in the same time as a query to a SSR data structure, e.g., $\mathcal{O}(1)$: If s and t belong to the same SCC C_i, then the supportive vertex v_i is reachable from s and can reach t, so the algorithm answers the query positively in accordance with observation **(O1)**. Otherwise, s belongs to an SCC C_i and t belongs to an SCC C_j, $j \neq i$. If C_i can reach C_j in the condensation of the graph, then also v_i can reach t and v_i is reachable from s, so the algorithm again answers the query positively in accordance with observation **(O1)**. If C_i cannot reach C_j in the condensation of the graph, then v_i can reach s, but not t so the algorithm answers the query negatively in accordance with observation **(O2)**. The supportive vertex representing the SCC that contains s or t, respectively, may be found in constant time using a map; however, updating it requires in turn to maintain the SCCs dynamically, which incurs additional costs during edge insertions and deletions.

Choosing Supportive Vertices. The simplest way to choose supportive vertices consists in picking them uniformly at random from the set of all vertices in the initial graph and never revisiting this decision. We refer to this algorithm as `SV(`k`)` (*k-Supportive Vertices*). During initialize(), `SV(`k`)` creates $\mathcal{L}_{SV}$ by drawing k non-isolated vertices uniformly at random from V. For each $v \in \mathcal{L}_{SV}$, it initializes both a dynamic single-source as well as a dynamic single-sink reachability data structure, each rooted at v. If less than k vertices have been picked during initialization because of isolated vertices, $\mathcal{L}_{SV}$ is extended as soon as possible.

Naturally, the initial choice of supportive vertices may be unlucky, which is why we also consider a variation of the above algorithm that periodically clears the previously chosen list of supportive vertices after c update operations and re-runs the selection process. We refer to this algorithm as `SVA(`k, c`)` (*k-Supportive Vertices with c-periodic Adjustments*).

As shown above, the perfect choice of a set of supportive vertices consists of exactly one per SCC. This is implemented by the third variant of our algorithm, `SVC` (*Supportive Vertices with SCC Cover*). However, maintaining SCCs dynamically is itself a non-trivial task and has recently been subject to extensive research [27, 30]. Here, we resolve this problem by waiving exactness, or, more precisely, the reliability of the cover. Similar to above, the algorithm takes two parameters z and c. During initialize(), it computes the SCCs of the input graph and arbitrarily chooses a supportive vertex in each SCC as representative if the SCC's size is at least z. In case that all SCCs are smaller than z, an arbitrary vertex that is neither a source nor a sink, if existent, is made supportive. The algorithm additionally maps each vertex to the representative of its SCC, where possible. After c update operations, this process is re-run and the list of supportive vertices as well as the vertex-to-representative map is updated suitably. However, we do not de-select supportive vertices picked in a previous round if they represent an SCC of size less than z, which would mean to also destroy their associated SSR data structures. Recall that computing the SCCs of a graph can be accomplished in $\mathcal{O}(n+m)$ time [36]. For a query(s,t), the algorithm looks up the SCC representative of s in its map and checks whether this information, if present, is still up-to-date by querying their associated data structures. In case of success, the algorithm answers the query as described

Table 2 Algorithms and abbreviations overview.

Algorithm	Long name	Algorithm	Long name
`DFS` / `BFS`	static DFS / BFS	`SV`	Supportive Vertices
`DBFS`	static DFS-BFS hybrid	`SVA`	Supportive Vertices with Adjustments
`BiBFS`	static bidirectional BFS	`SVC`	Supportive Vertices with SCC Cover

in the ideal scenario by asking whether the representative of s can reach t. Otherwise, the algorithm analogously tries to use the SCC representative of t. Outdated SCC representative information for s or t is deleted to avoid further unsuccessful checks. In case that neither s nor t have valid SCC representatives, the algorithm falls back to the operation mode of `SV`.

Algorithms for Maintaining Single-Source/Single-Sink Reachability. To access fully dynamic SSR information, we consider the two single-source reachability algorithms that have been shown to perform best in an extensive experimental evaluation on various types of input instances [13]. In the following, we only provide a short description and refer the interested reader to the original paper [13] for details.

The first algorithm, `SI`, is a fully dynamic extension of a simple incremental algorithm and maintains a not necessarily height-minimal reachability tree. It starts initially with a BFS tree, which is also extended using BFS in insertEdge$((u, v))$ only if v and vertices reachable from v were unreachable before. In case of deleteEdge$((u, v))$, the algorithm tries to reconstruct the reachability tree, if necessary, by using a combination of backward and forward BFS. If the reconstruction is expected to be costly because more than a configurable ratio ρ of vertices may be affected, the algorithm instead recomputes the reachability tree entirely from scratch using BFS. The algorithm has a worst-case insertion time of $\mathcal{O}(n + m)$, and, unless $\rho = 0$, a worst-case deletion time of $\mathcal{O}(n \cdot m)$.

The second algorithm, `SES`, is a simplified, fully dynamic extension of Even-Shiloach trees [35] and maintains a (height-minimal) BFS tree throughout all updates. Initially, it computes a BFS tree for the input graph. In insertEdge$((u, v))$, the tree is updated where necessary using a BFS starting from v. To implement deleteEdge$((u, v))$, the algorithm employs a simplified procedure in comparison to Even-Shiloach trees, where the BFS level of affected vertices increases gradually until the tree has been fully adjusted to the new graph. Again, the algorithm may abort the reconstruction and recompute the BFS tree entirely from scratch if the update cost exceeds configurable thresholds ρ and β. For constant β, the worst-case time per update operation (edge insertion or deletion) is in $\mathcal{O}(n + m)$.

Both algorithms have $\mathcal{O}(n + m)$ initialization time, support reachability queries in $\mathcal{O}(1)$ time, and require $\mathcal{O}(n)$ space. We use the same algorithms to maintain single-sink reachability information by running the single-source reachability algorithms on the reverse graph.

4 Experiments

Setup. For the experimental evaluation of our approach and the effects of its parameters, we implemented[1] it together with all four static approaches mentioned in Section 3 in

[1] Source code and instances are available at `https://dyreach.taa.univie.ac.at/transitive-closure`.

C++17 and compiled the code with GCC 7.4 using full optimization (`-O3 -march=native -mtune=native`). We would have liked to include the two best non-static algorithms from the earlier study [22]; unfortunately, the released source code is based on a proprietary algorithm library. Nevertheless, we are able to compare our new algorithms indirectly to both by relating their performance to DFS and BFS, as happened in the earlier study. All experiments were run sequentially under Ubuntu 18.04 LTS with Linux kernel 4.15 on an Intel Xeon E5-2643 v4 processor clocked at 3.4 GHz, where each experiment had exclusive access to one core and could use solely local memory, i.e., in particular no swapping was allowed.

For each algorithm (variant) and instance, we separately measured the time spent *by the algorithm* on updates as well as on queries. We specifically point out that the measured times exclude the time spent on performing an edge insertion or deletion *by the underlying graph data structure*. This is especially of importance if an algorithm's update time itself is very small, but the graph data structure has to perform non-trivial work. We use the dynamic graph data structure from the open-source library Algora [1], which is able to perform edge insertions and deletions in constant time. Our implementation is such that the algorithms are unable to look ahead in time and have to process each operation individually. To keep the numbers easily readable, we use k and m as abbreviations for $\times 10^3$ and $\times 10^6$, respectively.

Instances. We evaluate the algorithms on a diverse set of random and real-world instances, which have also been used in [13].

ER Instances. The random dynamic instances generated according to the Erdős-Renyí model $G(n, m)$ consist of an initial graph with $n = 100$k or $n = 10$m vertices and $m = d \cdot n$ edges, where $d \in [1.25 \dots 50]$. In addition, they contain a random sequence of 100k operations σ consisting of edge insertions, edge deletions, as well as reachability queries: For an insertion or a query, an ordered pair of vertices was chosen uniformly at random from the set of all vertices. Likewise, an edge was chosen uniformly at random from the set of all edges for a deletion. The resulting instances may contain parallel edges as well as loops and each operation is contained in a batch of ten likewise operations.

Kronecker Instances. Our evaluation uses two sets of size 20 each: `kronecker-csize` contains instances with $n \approx 130$k, whereas those in `kronecker-growing` have $n \approx 30$ initially and grow to $n \approx 130$k in the course of updates. As no generator for dynamic stochastic Kronecker graph exists, the instances were obtained by computing the differences in edges in a series of so-called *snapshot graphs*, where the edge insertions and deletions between two subsequent snapshot graphs were shuffled randomly. The snapshot graphs where generated by the `krongen` tool that is part of the SNAP software library [26], using the estimated initiator matrices given in [25] that correspond to real-world networks. The instances in `kronecker-csize` originate from ten snapshot graphs with 17 iterations each, which results in update sequences between 1.6m and 702m. As they are constant in size, there are roughly equally many insertions and deletions. Their densities vary between 0.7 and 16.4. The instances in `kronecker-growing` were created from thirteen snapshot graphs with five up to 17 iterations, resulting in 282k to 82m update operations, 66 % to 75 % of which are insertions. Their densities are between 0.9 and 16.4.

Real-World Instances. Our set of instances comprises all six directed, dynamic instances available from the Koblenz Network Collection KONECT [23], which correspond to the hyperlink network of Wikipedia articles for six different languages. In case of dynamic graphs, the update sequence is part of the instance. However, the performance of algorithms may be affected greatly if an originally real-world update sequence is permuted randomly [13]. For this reason, we also consider five "shuffled" versions per real-world network, where the edge

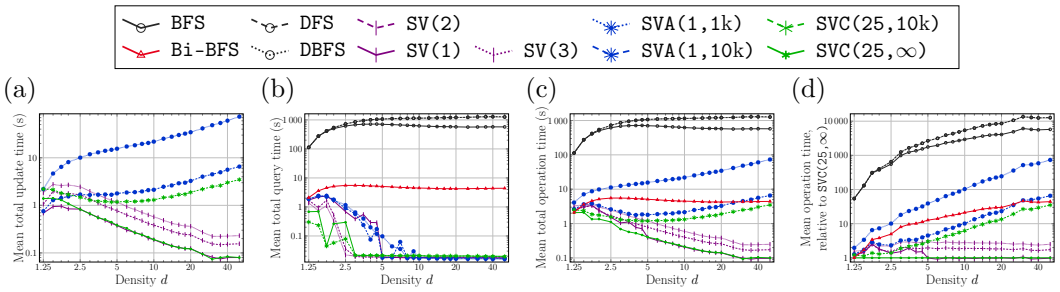


Figure 1 Random instances: $n = \sigma = 100$k and equally many insertions, deletions, and queries.

insertions and deletions have been permuted randomly. We refer to the set of original instances as `konect` and to the modified ones as `konect-shuffled`. Removals of non-existing edges have been ignored in all instances. In each case, the updates are dominated by insertions.

Experimental Results

We ran the algorithms `SV`, `SVA`, and `SVC` with different parameters: For `SV(`k`)`, we looked at $k = 1$, $k = 2$, and $k = 3$, which pick one, two, and three supportive vertices during initialization, respectively, and never reconsider this choice. We evaluate the variant that periodically picks new supportive vertices, `SVA(`k, c`)`, with $k = 1$ and $c = 1$k, $c = 10$k, and $c = 100$k. Preliminary tests for `SVC` revealed $z = 25$ as a good threshold for the minimum SCC size on smaller instances and $z = 50$ on larger. The values considered for c were again 10k and 100k. `BiBFS` served as fallback for all *Supportive Vertices* algorithms. Except for random ER instances with $n = 10$m, all experiments also included `BFS`, `DFS`, and `DBFS`; to save space, the bar plots only show the result for the best of these three. We used `SES` as subalgorithm on random instances, and `SI` on real-world instances, in accordance with the experimental study for single-source reachability [13]. More plots are available in the full version of this paper [12].

ER Instances. We start by assessing the average performance of all algorithms by looking at their running times on random ER instances. For $n = 100$k and equally many insertions, deletions, and queries, Figure 1 shows the mean running time needed to process all updates, all queries, and their sum, all operations, absolutely as well as the relative performances for all operations, where the mean is taken over 20 instances per density. As there are equally many insertions and deletions, the density remains constant. Note that all plots for random ER instances use logarithmic axes in both dimensions.

It comes as no surprise that `SV(2)` and `SV(3)` are two and three times slower on *updates*, respectively, than `SV(1)` (cf. **Figure 1a**). As their update time consists solely of the update times of their SSR data structures, they inherit their behavior and become faster, the denser the instance [13]. The additional work performed by `SVA(`$1, 1$k`)` and `SVA(`$1, 10$k`)`, which re-initialized their SSR data structures 66 and six times, respectively, is plainly visible and increases also relatively with growing number of edges, which fits the theoretical (re-)initialization time of $\mathcal{O}(n + m)$. Computing the SCCs only initially, as `SVC(`$25, \infty$`)` does, led to higher update times on very sparse instances due to an increased number of supportive vertices, but matched the performance of `SV(1)` for $d \geq 2.5$. As expected, re-running the SCC computation negatively affects the update time. In contrast to `SVA(`$1, 10$k`)`, however, `SVC(`$25, 10$k`)` keeps a supportive vertex as long as it still represents an SCC, and thereby saves

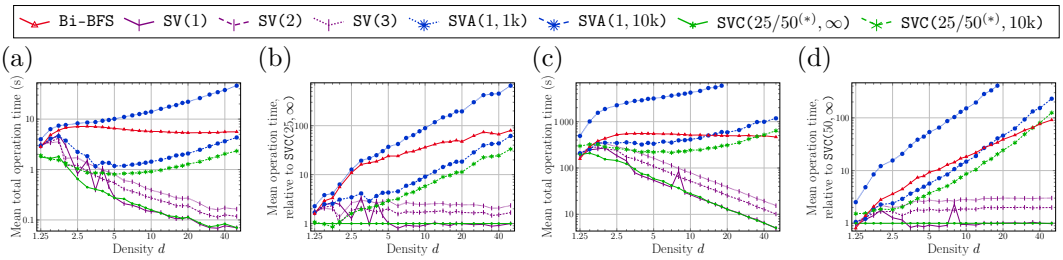


Figure 2 Random instances: $n = \sigma = 100\text{k}$ and 50 % queries (a, b); $n = 10\text{m}$, $\sigma = 100\text{k}$, and equally many insertions, deletions, and queries (c, d).
(*) `SVC` parameter: 25 for $n = 100\text{k}$, 50 for $n = 10\text{m}$.

the time to destroy the old SSR data structures and re-initialize the new ones. Evidently, both `SVA` algorithms used a single supportive vertex for $d \geq 2.5$.

Looking at *queries* (cf. **Figure 1b**), it becomes apparent that `SVC`$(25, 10\text{k})$ can make use of its well-updated SCC representatives as supportive vertices and speed up queries up to a factor of 54 in comparison to `SVA` and `SV`. Up to $d = 3$, it also outperforms `SVC`$(25, \infty)$. For larger densities, the query times among all dynamic algorithms level up progressively and reach at least equality already at $d = 2.5$ in case of `SV(2)` and `SV(3)`, at $d = 5$ in case of `SV(1)`, and at $d = 10$ at the latest for all others. This matches a well-known result from random graph theory that simple ER graphs with $m > n \ln n$ are strongly connected with high probability [2]. The running times also fit our investigations into the mean percentage of queries answered during the different stages in query$(\cdot)$ by the algorithms: For $d = 2$, `SV(1)` could answer 80 % of all queries without falling back to `BiBFS`, which grew to almost 100 % for $d = 5$ and above. `SV(2)` answered even more than 95 % queries without fallback for $d = 2$, and close to 100 % already for $d = 3$. The same applied to `SVC`$(25, \infty)$, which could use SCC representatives in the majority of these fast queries. `SV(1)` and `SV(2)` instead used mainly observation **(O1)**, but also **(O2)** and **(O3)** in up to 10 % of all queries. As all vertices are somewhat alike in ER graphs, periodically picking new supportive vertices does not affect the mean query performance. In fact, `SV` and `SVA` are up to 20 % faster than `SVC` on the medium and denser instances, which can be explained by the missing overhead for maintaining the map of representatives. All *Supportive Vertices* algorithms process queries considerably faster than `BiBFS`. The average speedup ranges between almost 7 on the sparsest graphs in relation to `SVC`$(25, 10\text{k})$ and more than 240 in relation to `SV(1)` on the densest ones. The traditional static algorithms `BFS`, `DFS`, as well as the hybrid `DBFS` were *distinctly slower* by a factor of up to 31k (`BFS`) and almost 70k (`DFS`, `DBFS`) in comparison to `SV(1)`, and even 53 to 130 and 290 times slower than `BiBFS`.

In sum over *all operations*, if there are equally many insertions, deletions, and queries (cf. **Figures 1c, d**), `SVC`$(25, \infty)$ and `SV(1)` were the *fastest algorithms* on all instances, where `SVC`$(25, \infty)$ won on the sparser and `SV(1)` won on the denser instances. For $d = 1.25$, `BiBFS` was almost as fast, but up to 45 times slower on all denser instances. `SV(2)` and `SV(3)` could not compensate their doubled and tripled update costs, respectively, by their speedup in query time, which also holds for `SVC`$(25, 10\text{k})$. `BFS`, `DFS`, and `DBFS` were between 54 and 13k times slower than `SVC`$(25, \infty)$ and `SV(1)`, despite the high proportion of updates, and are therefore *far from competitive*.

We repeated our experiments with $n = 100\text{k}$ and 50 % *queries* among the operations and equally many insertions and deletions, as well as with $n = 10\text{m}$ and equal ratios of

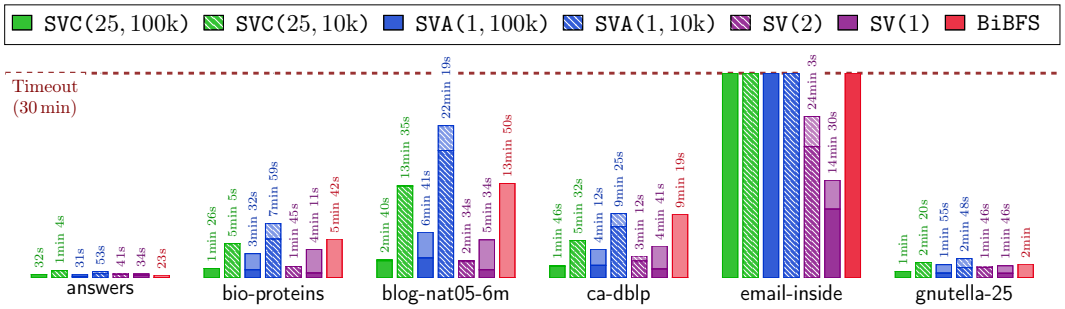


Figure 3 Total update (dark color) and query (light color) times on selected `kronecker-csize` instances.

insertions, deletions, and queries. The results, shown in **Figure 2**, *confirm* our findings above. In case of 50 % *queries*, a second supportive vertex as in `SV(2)` additionally stabilized the mean running time in comparison to `SV(1)`, up to $d = 5$ (cf. **Figures 2a, b**), but none of them could beat $\texttt{SVC}(25, \infty)$ on sparse instances. On denser graphs, `SV(1)` was again equally fast or even up to 20 % faster. As expected due to the higher ratio of queries, `BiBFS` lost in competitiveness in comparison to the above results and is between 1.6 and almost 80 times slower than $\texttt{SVC}(25, \infty)$ on dense instances. On the set of larger instances with $n = 10\text{m}$ (cf. **Figures 2c, d**), `SVA(1,1k)` reached the timeout set at 2 h on instances with $d \geq 20$. The *fastest* algorithms on average across all densities were again `SV(1)` and $\texttt{SVC}(50, \infty)$. `BiBFS` won for $d = 1.25$, where it was about 20 % faster than $\texttt{SVC}(50, \infty)$ and 10 % faster than `SV(1)`. Its relative performance then deteriorated with increasing density up to a slowdown factor of 91. Except for $d = 1.25$, $\texttt{SVC}(50, \infty)$ outperformed `SV(1)` on very sparse instances and was in general also more stable in performance, as can be observed for $d = 8$: Here, `SV(1)` picked a bad supportive vertex on one of the instances, which resulted in distinctly increased mean, median, and maximum query times. On instances with density around $\ln n$ and above, `SV(1)` was slightly faster due to its simpler procedure to answer queries and also more stable than on sparser graphs.

In *summary*, `SVC` with $c = \infty$ clearly showed the *best and most reliable performance* on average, closely followed by `SV(1)`, which was *slightly faster* if the graphs were *dense*.

Kronecker Instances. In contrast to ER instances, stochastic Kronecker graphs were designed to model real-world networks, where vertex degrees typically follow a power-law distribution and the neighborhoods are more diverse. For this reason, the choice of supportive vertices might have more effect on the algorithms' performances than on ER instances. **Figure 3** shows six selected results for all dynamic algorithms as well as `BiBFS` on `kronecker-csize` instances with a query ratio of 33 %: Each bar consists of two parts, the lower, darker one depicts the total update time, the lighter one on top the total query time, which is comparatively small for most dynamic algorithms and therefore sometimes hardly discernible. In case that an algorithm reached the timeout, we only use the darker color for the bar. The description next to each bar states the total operation time. By and large, the picture is very similar to that for ER instances. On 13 and 14 out of the 20 instances, `BFS`/`DBFS` and `DFS`, respectively, did not finish within six hours. As on the previous sets of instances, these algorithms are *far from competitive*. The performance of `BiBFS` was ambivalent: it was the fastest on two instances, but lagged far behind on others. `SV(1)` and `SV(2)` showed the *best performance* on the *majority* of instances and were the *only ones* to finish within six hours on the largest instance of the set, email-inside. On some graphs, the total operation time of `SV(1)` was dominated by the query time,

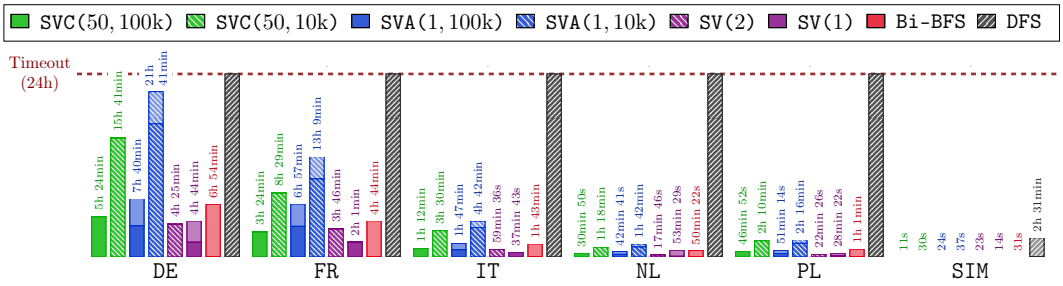


Figure 4 Total update (dark color) and query (light color) times on `konect` instances.

e.g., on bio-proteins, whereas `SV(2)` was able to reduce the total operation time by more than half by picking a second supportive vertex. However, `SVC(25, 100k)` was even able to outperform this slightly and was the *fastest* algorithm on half of the instances. As above, recomputing the SCCs more often (`SVC(25, 10k)`) or periodically picking new support vertices (`SVA(1, 1k)`, `SVA(1, 10k)`) led to a slowdown in general.

On `kronecker-growing`, `SV(1)` was the *fastest* algorithm on all but one instance. The overall picture is very similar.

Real-World Instances. In the same style as above, **Figure 4** shows the results on the six real-world instances with real-world update sequences, `konect`, again with 33 % queries among the operations. We set the timeout at 24 h, which was reached by `BFS`, `DFS`, and `DBFS` on all but the smallest instance. On the largest instance, `DE`, they were able to process only around 6 % of all updates and queries within this time. The *fastest algorithms* were `SV(1)` and `SV(2)`. If `SV(1)` chose the single support vertex well, as in case of `FR`, `IT`, and `SIM`, the query costs and the total operation times were low; on the other instances, the second support vertex, as chosen by `SV(2)`, could speed up the queries further and even compensate the cost for maintaining a second pair of SSR data structures. Even though the instances are growing and most vertices were isolated and therefore not eligible as supportive vertex during initialization, periodically picking new supportive vertices, as `SVA` does, did not improve the running time. `SVC(50, ∞)` performed well, but the extra effort to *compute the SCCs* and use their representatives as supportive vertices *did not pay off*; only on `SIM`, `SVC(50, ∞)` was able to outperform both `SV(1)` and `SV(2)` marginally.

Randomly permuting the sequence of update operations, as for the instance set `konect-shuffled`, did not change the overall picture.

5 Conclusion

Our extensive experiments on a diverse set of instances draw a somewhat surprisingly consistent picture: The most simple algorithm from our family, `SV(1)`, which picks a single supportive vertex, performed extremely well and was the fastest on a large portion of the instances. On those graphs where it was not the best, `SV(2)` could speed up the total running time by picking a second supportive vertex, i.e., the faster query time could compensate for the doubled update time. Additional statistical evaluations showed that already for sparse graphs, `SV(1)` and `SV(2)` answered a great majority of all queries in constant time using only its supportive vertices. Recomputing the strongly connected components of the graph in very large intervals and using them for the choice of supportive vertices yielded a comparatively good or marginally better algorithm on random instances, but not on real-world graphs.

The classic static algorithms BFS and DFS, which were competitive or even superior to the dynamic algorithms evaluated experimentally in previous studies, lagged far behind the new algorithms and were outperformed by several orders of magnitude.

References

1 Algora – a modular algorithms library. https://libalgora.gitlab.io.

2 B. Bollobás. *Random graphs*. Number 73 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2001.

3 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter Elementary Data Structures. MIT Press, 3rd edition, 2009.

4 C. Demetrescu and G. F. Italiano. Trade-offs for fully dynamic transitive closure on dags: Breaking through the $\mathcal{O}(n^2)$ barrier. *J. ACM*, 52(2):147–156, March 2005. doi:10.1145/1059513.1059514.

5 C. Demetrescu and G. F. Italiano. Mantaining dynamic matrices for fully dynamic transitive closure. *Algorithmica*, 51(4):387–427, 2008.

6 J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, April 1972. doi:10.1145/321694.321699.

7 R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, June 1962. doi:10.1145/367766.368168.

8 L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. doi:10.4153/CJM-1956-045-5.

9 D. Frigioni, T. Miller, U. Nanni, and C. Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *Journal of Experimental Algorithmics (JEA)*, 6:9, 2001.

10 A. Gitter, A. Gupta, J. Klein-Seetharaman, and Z. Bar-Joseph. Discovering pathways by orienting edges in protein interaction networks. *Nucleic Acids Research*, 39(4):e22–e22, November 2010.

11 A. V. Goldberg, S. Hed, H. Kaplan, R. E. Tarjan, and R. F. Werneck. Maximum flows by incremental breadth-first search. In *European Symposium on Algorithms*, pages 457–468. Springer, 2011.

12 K. Hanauer, M. Henzinger, and C. Schulz. Faster fully dynamic transitive closure in practice, 2020. arXiv:2002.00813.

13 K. Hanauer, M. Henzinger, and C. Schulz. Fully dynamic single-source reachability in practice: An experimental study. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 106–119, 2020. doi:10.1137/1.9781611976007.9.

14 M. Henzinger and V King. Fully dynamic biconnectivity and transitive closure. In *36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 664–672. IEEE, 1995.

15 M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *47th ACM Symposium on Theory of Computing*, STOC'15, pages 21–30. ACM, 2015.

16 T. Ibaraki and N. Katoh. On-line computation of transitive closures of graphs. *Information Processing Letters*, 16(2):95–97, 1983. doi:10.1016/0020-0190(83)90033-9.

17 G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986. doi:10.1016/0304-3975(86)90098-8.

18 G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28(1):5–11, 1988.

19 V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, page 81, USA, 1999. IEEE Computer Society.

20 V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *Journal of Computer and System Sciences*, 65(1):150–167, 2002. doi:10.1006/jcss.2002.1883.

21 V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proceedings of the 7th Annual International Conference on Computing and Combinatorics*, COCOON '01, page 268–277, Berlin, Heidelberg, 2001. Springer-Verlag.

22 I. Krommidas and C. D. Zaroliagis. An experimental study of algorithms for fully dynamic transitive closure. *ACM Journal of Experimental Algorithmics*, 12:1.6:1–1.6:22, 2008. doi:10.1145/1227161.1370597.

23 J. Kunegis. Konect: the Koblenz network collection. In *22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

24 F. Le Gall. Powers of tensors and fast matrix multiplication. In K. Nabeshima, K. Nagasaka, F. Winkler, and Á. Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303. ACM, 2014. doi:10.1145/2608628.2608664.

25 J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11:985–1042, March 2010. URL: http://dl.acm.org/citation.cfm?id=1756006.1756039.

26 J. Leskovec and R. Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

27 J. Łącki. Improved deterministic algorithms for decremental transitive closure and strongly connected components. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, page 1438–1445, USA, 2011. Society for Industrial and Applied Mathematics.

28 T. Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.

29 L. Roditty. A faster and simpler fully dynamic transitive closure. *ACM Trans. Algorithms*, 4(1), March 2008. doi:10.1145/1328911.1328917.

30 L. Roditty. Decremental maintenance of strongly connected components. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 1143–1150, USA, 2013. Society for Industrial and Applied Mathematics.

31 L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM Journal on Computing*, 37(5):1455–1471, 2008. doi:10.1137/060650271.

32 L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM J. Comput.*, 41(3):670–683, 2012. doi:10.1137/090776573.

33 L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM Journal on Computing*, 45(3):712–733, 2016.

34 P. Sankowski. Dynamic transitive closure via dynamic matrix inverse. In *45th Symposium on Foundations of Computer Science (FOCS)*, pages 509–517. IEEE, 2004.

35 Y. Shiloach and S. Even. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, 1981.

36 R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.

37 J. van den Brand, D. Nanongkai, and T. Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 456–480, 2019. doi:10.1109/FOCS.2019.00036.

38 S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, January 1962. doi:10.1145/321105.321107.

39 D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30(4):369–384, 1993.

Concurrent Expandable AMQs on the Basis of Quotient Filters

Tobias Maier
Karlsruhe Institute of Technology, Germany
t.maier@kit.edu

Peter Sanders
Karlsruhe Institute of Technology, Germany
sanders@kit.edu

Robert Williger
Karlsruhe Institute of Technology, Germany

Abstract

A quotient filter is a cache efficient Approximate Membership Query (AMQ) data structure. Depending on the fill degree of the filter most insertions and queries only need to access one or two consecutive cache lines. This makes quotient filters very fast compared to the more commonly used Bloom filters that incur multiple independent memory accesses depending on the false positive rate. However, concurrent Bloom filters are easy to implement and can be implemented lock-free while concurrent quotient filters are not as simple. Usually concurrent quotient filters work by using an external array of locks – each protecting a region of the table. Accessing this array incurs one additional memory access per operation. We propose a new locking scheme that has no memory overhead. Using this new locking scheme we achieve 1.6× times higher insertion performance and over 2.1× higher query performance than with the common external locking scheme.

Another advantage of quotient filters over Bloom filters is that a quotient filter can change its capacity when it is becoming full. We implement this growing technique for our concurrent quotient filters and adapt it in a way that allows unbounded growing while keeping a bounded false positive rate. We call the resulting data structure a fully expandable quotient filter. Its design is similar to scalable Bloom filters, but we exploit some concepts inherent to quotient filters to improve the space efficiency and the query speed.

Additionally, we propose several quotient filter variants that are aimed to reduce the number of status bits (2-status-bit variant) or to simplify concurrent implementations (linear probing quotient filter). The linear probing quotient filter even leads to a lock-free concurrent filter implementation. This is especially interesting, since we show that any lock-free implementation of other common quotient filter variants would incur significant overheads in the form of additional data fields or multiple passes over the accessed data. The code produced as part of this submission can be found at `https://www.github.com/Toobiased/lpqfilter`.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Shared memory algorithms

Keywords and phrases Quotient filter, Concurrent data structures, Locking

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.15

Supplementary Material The code produced as part of this submission can be found at `https://www.github.com/Toobiased/lpqfilter`.

1 Introduction

Approximate Membership Query (AMQ) data structures offer a simple interface to represent sets. Elements can be inserted, queried, and depending on the use case elements can also be removed. A query returns true if the element was previously inserted. Querying an element might return true even if it was not inserted. We call this a false positive. The probability that a non-inserted element leads to a false positive is called the false positive rate of the

filter. It can usually be chosen when constructing the data structure. AMQ data structures have two main advantages over other set representations they are fast and space efficient. Similar to hash tables most AMQ data structures have to be initialized with their capacity. This can be a problem for their space efficiency when the final size is unknown a priori.

AMQ data structures have become an integral part of many complex data structures or data base applications. Their small size and fast accesses can be used to sketch large, slow data sets. The AMQ filter is used before accessing the data base to check whether a slow lookup is actually necessary. Some recent uses of AMQs include network analysis [1] and bio informatics [15]. These are two of the areas with the largest data sets available today, but given the current big data revolution we expect more and more research areas will have a need for the space efficiency and speedup potential of AMQs. The most common AMQ data structure in practice is still a Bloom filter even though a quotient filter would usually be significantly faster. One reason for this is probably inertia, but another reason might be that concurrent Bloom filters are easy to implement – even lock-free and well scaling implementations. This is important because well scaling implementations have become critical in today's multi-processor scenarios since single core performance stagnates and efficient multi-core computation has become a necessity to handle growing data sets.

We present a technique for concurrent quotient filters that uses fine grained local locking inside the table – without an external array of locks. Instead of traditional locks we use the per-slot status bits that are inherent in quotient filters to lock local ranges of slots. As long as there is no contention on a single area of the table there is very little overhead to this locking technique because the locks are placed in slots that would have been accessed anyways. The capacity of a quotient filter can be increased without access to the original elements. However, because the false positive rate grows linearly with the number of inserted elements, this is only useful for a limited total growing factor. We implement this growing technique for our concurrent quotient filters and extend it to allow large growing factors at a strictly bounded false positive rate. These *fully expandable quotient filters* combine growing quotient filters with the multi level approach of scalable Bloom filters [2].

Related Work

Since quotient filters were first described in 2012 [3] there has been a steady stream of improvements. For example Pandey et al. [16] have shown how to reduce the memory overhead of quotient filters by using rank-select data structures. This also improves the performance when the table becomes full. Additionally, they show an idea that saves memory when insertions are skewed (some elements are inserted many times). They also mention the possibility for concurrent access using an external array of locks (see Section 5 for results). Recently, there was also a GPU-based implementation of quotient filters [9], further indicating that there is a lot of interest in concurrent AMQs even in these highly parallel scenarios.

Quotient filters are not the only AMQ data structures that have received attention recently. Cuckoo filters [8, 13] and very recently Morton filters [4] (based on cuckoo filters) are two other examples of AMQ data structures. Due to their similarity to cuckoo hash tables, they do not lend themselves to easy parallel solutions (insertions can have large effects on the overall data structure).

A scalable Bloom filter [2] allows unbounded growing by adding additional levels of Bloom filters once a level becomes full. Each new filter is initialized with a higher capacity and more hash functions than the last. The query time is dependent on the number of times the filter has grown both because more filters have to be checked and because later filters have more hash functions. In Section 4, we show a similar technique for fully expandable quotient filters that mitigates many of these problems.

Overview. In Section 2 we introduce the terminology, sequential quotient filters as well as our filter variations in a sequential setting. Section 3 describes our approach to concurrent quotient filters the lock-free linear probing quotient filter and the locally locked quotient filter. The dynamically growing quotient filter variant is described in Section 4. All presented data structures are evaluated in Section 5. Afterwards, we draw our conclusions in Section 6.

2 Sequential Quotient Filter

In this section we describe the basic sequential quotient filter as well as some variants to the main data structure. Throughout this paper, we use m for the number of slots in our data structure and n for the number of inserted elements. The fill degree of any given table is denoted by $\delta = n/m$. Additionally, we use p^+ to denote the probability of a false positive query. Quotient filters are approximate membership query data structures that were first described by Bender et al. [3] and build on an idea for space efficient hashing originally described by Cleary [6]. Quotient filters replace possibly large elements by *fingerprints*. The fingerprint $f(x)$ of an element x is a number in a predefined range $f: x \mapsto \{0, ..., 2^k - 1\}$ (binary representation with exactly k digits). We commonly get a fingerprint of x by taking the k bottommost bits of a hash function value $h(x)$ (i.e., xxHash [7]).

A quotient filter stores the fingerprints of all inserted elements. When executing a query for an element x, the filter returns `true` if the fingerprint $f(x)$ was previously inserted and `false` otherwise. Thus, a query looking for an element that was inserted always returns `true`. A false positive occurs when x was not inserted, but its fingerprint $f(x)$ matches that of a previously inserted element. Given a fully random fingerprint function, the probability of two fingerprints being the same is 2^{-k}. Therefore, the probability of a false positive is bounded by $n \cdot 2^{-k}$ where n is the number of stored fingerprints.

To achieve expected constant query times as well as to save memory, fingerprints are stored in a special data structure that is similar to a hash table with open addressing (specifically it is similar to robinhood hashing [5]). During this process, the fingerprint of an element x is split into two parts: the topmost q bits called the quotient $quot(x)$ and the bottommost r bits called the remainder $rem(x)$ ($q + r = k$). The quotient is used to address a table that consisting of $m = 2^q$ memory slots of $r + 3$ bits (a slot can store one remainder and three additional status bits). The quotient of each element is only stored implicitly by the position of the element in the table. The remainder is stored explicitly within one slot of the table. Similar to many hashing techniques, we try to store each element in one designated slot (index $quot(x)$) which we call its canonical slot. With the help of the status bits we can reconstruct quotient of each stored element even when it is shifted after its canonical slot.

Table 1 Meaning of different status bit combinations.

`1**`	this slot has a run
`000`	empty slot
`100`	cluster start
`*01`	run start
`*11`	continuation of a run
`*10`	– (not used see 3.2)

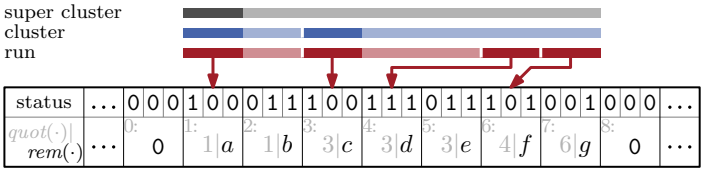


Figure 1 Section of the table with highlighted runs, clusters, and superclusters. Runs point to their canonical slot.

The main idea for resolving collisions is to find the next free slot – similar to linear probing hash tables – but to reorder the elements such that they are sorted by their fingerprints (see Figure 1). Elements with the same quotient (the same canonical slot) are stored in

consecutive slots, we call them a run. The canonical run of an element is the run associated with its canonical slot. The canonical run of an element does not necessarily start in its canonical slot. It can be shifted by other runs. If a run starts in its canonical slot we say that it starts a cluster that contains all shifted runs after it. Multiple clusters that have no free slots between them form a supercluster. We use the 3 status bits that are part of each slot to distinguish between runs, clusters, and empty slots. For this we store the following information about the contents of the slot (further described in Table 1): Were elements hashed to this slot (is its run non-empty)? Does the element in this slot belong to the same run as the previous entry (used as a run-delimiter signaling where a new run starts)? Does the element in this slot belong to the same cluster as the previous entry (is it shifted)?

During the query for an element x, all remainders stored in x's canonical run have to be compared to $rem(x)$. First we look at the first status bit of the canonical slot. If this status bit is unset there is no run for this slot and we can return false. If there is a canonical run, we use the status bits to find it by iterating to the left until the start of a cluster (third status bit $= 0$). From there, we move to the right counting the number of non-empty runs to the left of the canonical run (slots with first status bit $= 1$ left of $quot(x)$) and the number of run starts (slots with second status bit $= 0$). This way, we can easily find the correct run and compare the appropriate remainders.

An insert operation for element x proceeds similar to a query, until it either finds a free slot or a slot that contains a fingerprint that is $\geq f(x)$. The current slot is replaced with $rem(x)$ shifting the following slots to the right (updating the status bits appropriately).

When the table fills up, operations become slow because the average cluster length increases. When the table is full, no more insertions are possible. In this case, quotient filters can be migrated into a larger table – increasing the overall capacity. To do this a new table is allocated with twice the size of the original table and one less bit per remainder. Addressing the new table demands an additional quotient bit, since it is twice the size. This issue is solved by moving the uppermost bit from the remainder into the quotient ($q' = q + 1$ and $r' = r - 1$). The fingerprint size and number of elements remain the same, therefore, the capacity increase does not impact the false positive rate of the filter. But the false positive rate still increases linearly with the number of insertions ($p^+ = n \cdot 2^{-k}$). Therefore, the false positive rate doubles when the table is filled again. We call this migration technique *bounded growing*, because to guarantee reasonable false positive rates this method of growing should only be used a bounded number of times.

2.1 Previously Unpublished Variants

Two-Status-Bit Quotient Filter. Pandey et al. [16] already proposed a 2 status bit variant of their counting quotient filter. Their implementation however has average query and insertion times in $\Theta(n)$. The goal of our 2-status bit variant is to achieve running times close to $O(\textit{supercluster length})$. To achieve this, we change the definition of the fingerprint (only for this variant) such that no remainder can be zero $f' : x \mapsto \{0, ..., 2^q - 1\} \times \{1, ..., 2^r - 1\}$. Obtaining a non-zero remainder can easily be achieved by rehashing an element with different hash functions until the remainder is non-zero. This change to the fingerprint only has a minor impact on the false positive rate of the quotient filter ($n/m \cdot (2^r - 1)^{-1}$ instead of $n/m \cdot 2^{-r}$).

Given this change to the fingerprint we can easily distinguish empty slots from filled ones. Each slot uses two status bits the *occupied-bit* (first status bit in the description above) and the *new-run-bit* (run-delimiter). Using these status bits we can find a particular run by going to the left until we find a free slot and then counting the number of occupied- and new-run-bits while moving right from there (*Note*: a cluster start within a larger supercluster cannot be recognized when moving left).

Linear Probing Quotient Filter. This quotient filter is a hybrid between a linear probing hash table and a quotient filter. It uses no reordering of stored remainders and no status bits. To offset the removal of status bits we add three additional bits to the remainder (leading to a longer fingerprint). Similar to the two-status-bit quotient filter above we ensure no element x has $rem(x) = 0$ (by adapting the fingerprint function).

During the insertion of an element x the remainder $rem(x)$ is stored in the first empty slot after its canonical slot. Without status bits and reordering it is impossible to reconstruct the fingerprint of each inserted element. Therefore, a query looking for an element x compares its remainder $rem(x)$ with every remainder stored between x's canonical slot and the next empty slot. There are potentially more remainders that are compared than during the same operation on a normal quotient filter. The increased remainder length however reduces the chance of a single comparison to lead to a false positive by a factor of 8. Therefore, as long as the average number of compared remainders is less than 8 times more than before, the false positive remains the same or improves. In our tests, we found this to be true for fill degrees up to $\approx 70\%$. The number of compared remainders corresponds to the number of slots probed by a linear probing hash table, this gives the bound below. It should be noted that linear probing quotient filters cannot support deletions or the bounded growing technique.

▶ **Corollary 1** (p^+ linear probing q.f.). *The false positive rate of a linear probing quotient filter with $\delta = n/m$ (bound due to Knuth [10] chapter 6.4) is*

$$p^+ = \frac{E[\#comparisons]}{2^{r+3}-1} = \frac{1}{2}\left(1 + \frac{1}{(1-\delta)^2}\right) \cdot \frac{1}{2^{r+3}-1}$$

3 Concurrent Quotient Filter

Why does our concurrent quotient filter (not the Concurrent Linear Probing filter variant) still use locking? There are *Some Problems Inherent* to lock-free quotient filters. Every query has to read multiple data entries in a consistent state to succeed. All commonly known variants (except the linear probing quotient filter) use at least two status bits per slot, i.e., the *occupied-bit* and some kind of *run-delimiter-bit* (the run-delimiter might take different forms). The remainders and the run-delimiter-bit that belong to one slot cannot reliably be stored close to their slot and its occupied-bit. Therefore, the occupied-bit and the run-delimiter-bit cannot be updated with one atomic operation. For this reason, implementing a lock-free quotient filter that uses status bits would cause significant overheads (i.e. additional memory or multiple passes over the accessed data). The problem is that reading parts of multiple different but individually consistent states of the quotient filter leads to an inconsistent perceived state. To show this we assume that insertions happen atomically and transfer the table from one consistent state directly into the new final consistent state. During a query we have to find the canonical run and scan the remainders within this run. To find the canonical run we have to compute its rank among non-empty runs using the occupied-bits (either locally by iterating over slots or globally using a rank-select-data structure) and then find the run with the same rank using the run-delimiter-bits. There is no way to guarantee that the overall state has not changed between finding the rank of the occupied-bit bit and finding the appropriate run-delimiter. Specifically we might find a new run that was created by an insertion that shifted the actual canonical run. Similar things can happen when accessing the remainders especially when remainders are not stored interleaved with their corresponding status bits. Some of these issues can be fixed using multiple passes over the data but ABA problems might arise in particular when deletions are possible. To avoid these problems while

maintaining performance we have two options: either comparing remainders from different canonical slots and removing the need for status bits (i.e. the linear probing quotient filter) or by using locks that protect the relevant range of the table.

Besides the correctness there are two main goals for any concurrent data structure: scalability and overall performance. One major performance advantage of quotient filters over other AMQ data structures is their cache efficiency. Concurrent quotient filters should attempt to preserve this advantage. Especially insertions into short or empty clusters and (unsuccessful) queries with empty canonical runs lead to operations with only one or two accessed cache lines and at most one write operation. Any variant using external locks has an immediate disadvantage here.

Concurrent Slot Storage. Whenever data is accessed by multiple threads concurrently, we have to think about the atomicity of operations. The slots of a quotient filter can have arbitrary sizes. To reduce the number of updated cache lines, and to store all data connected to one slot together in one atomic data element we alternate between status bits and remainders. We have to take into account the memory that is wasted by using atomic data types inefficiently. It would be common to just use the smallest data type that can hold both the remainder and status bits and waste any excess memory. But quotient filters are designed to be space efficient. Therefore, we use the largest common atomic data type (64 bit) and pack as many slots as possible into one data element – we call this packed construct a group (of slots). This way, the waste of multiple slots accumulates to encompass additional elements. Furthermore, using this technique we can atomically read or write multiple slots at once – allowing us to update in bulk and even avoid some locking.

3.1 Concurrent Linear Probing Quotient Filter

Operations on a linear probing quotient filter (as described in Section 2.1) can be executed concurrently similar to operations of a linear probing hash table [11]. The table is constructed out of grouped atomic slots as described above. Each insertion changes the table atomically using a single compare and swap instruction. This implementation is even lock free, because at least one competing write operation on any given cell is successful. Queries find all remainders that were inserted into their canonical slot, because they are stored between their canonical slot and the next empty slot and the contents of a slot never change once something is stored within it.

3.2 Concurrent Quotient Filter with Local Locking

In this section we introduce an easy way to implement concurrent quotient filters without any memory overhead and without increasing the number of cache lines that are accessed during operations. This concurrent implementation is based on the basic (3-status-bit) quotient filter. The same ideas can also be implemented with the 2-status-bit variant, but there are some problems discussed later in this section.

Using status bits for local locking. To implement our locking scheme we use Two Combinations of Status Bits that are impossible to occur naturally (see Table 1) – `010` and `110`. We use `110` to implement a write lock. It is written into the first free slot after the canonical supercluster at the beginning of each write operation (using a CAS-operation see Block B Algorithm 1.a). Insertions wait when encountering a write lock. This ensures that only one insert operation can be active per supercluster. The combination `010` is used as a read lock.

Algorithm 1 Concurrent Operations.

1.a Insertion

```
(quot, rem) ← f(key)
// Block A: try a trivial insertion
group ← atomically load data around quot
if insertion into group is trivial then
    finish insertion with a CAS and return
// Block B: write lock the supercluster
scan right from it ← quot
    if it is write locked then
        wait until released and continue
    if it is empty then
        lock it with write lock and break
            if CAS unsuccessful re-examine it
// Block C: read lock the cluster
scan left from it ← quot
    if it is read locked then
        wait until released and retry this slot
    if it is cluster start then
        lock it with read lock and break
            if CAS unsuccessful re-examine it
// Block D: find the correct run
occ = 0;   run = 0
scan right from it
    if it is occupied and it < quot then occ++
    if it is run start then run++
    if occ = run and it ≥ quot then break
// Block E: insert into the run and shift
scan right from it
    if it is read locked then
        wait until released
    store rem in correct slot
    shift content of following slots
    (keep groups consistent)
    break after overwriting the write lock
unlock the read lock
```

1.b Query

```
(quot, rem) ← f(key)
// Block F: try trivial query
group ← atomically load data around slot quot
if answer can be determined from group then
    return this answer
// Block G: read lock the cluster
scan left from it ← quot
    if it is read locked then
        wait until released and retry this slot
    if it is cluster start then
        lock it with read lock and break
            if CAS unsuccessful re-examine it
// Block H: find the correct run
occ = 0;   run = 0
scan right from it
    if it is occupied and it < quot then occ++
    if it is run start then run++
    if occ = run and it ≥ quot then break
// Block I: search remainder within the run
scan right from it
    if it = rem
        unlock the read lock
        return contained
    if it is not continuation of this run
        unlock the read lock
        return not contained
```

All operations (both insertions Block C Algorithm 1.a and queries Block G Algorithm 1.b) replace the status bits of the first element of their canonical cluster with `010`. Additionally, inserting threads wait for each encountered read lock while moving elements in the table, see Block E Algorithm 1.a. When moving elements during the insertion each encountered cluster start is shifted and becomes part of the canonical cluster that is protected by the insertion's original read lock. This ensures, that no operation can change the contents of a cluster while it is protected with a read lock. It would also be possible to implement deletions in a similar way to insertions (first acquiring a write lock, then a read lock). But both queries and insertions are impacted by the possibility of holes being created within clusters while acquiring a lock requiring some extra work. The data structure used in our tests does not have deletions enabled.

Avoiding locks. Many instances of locking can be avoided, e.g., when the canonical slot for an insertion is empty (write the elements with a single compare and swap operation), when the canonical slot of a query either has no run (first status bit), or stores the wanted fingerprint. In addition to these trivial instances of lock elision, where the whole operation happens in one slot, we can also profit from our grouped atomic storage scheme. Since we store multiple slots together in one atomic data member, multiple slots can be changed at once. Each operation can act without acquiring a lock, if the whole operation can be completed without loading another data element. The correctness of the algorithm is still guaranteed, because the slots within one data element are always in a consistent state.

2-Status-Bit Concurrent Quotient Filter. In the 2-status-bit variant of the quotient filter there are no unused status bit combinations that can be used as read or write locks. But zero can never be a remainder when using this variant, therefore, a slot with an empty remainder but non-zero status bits cannot occur. We can use such a cell to represent a lock. Using this method, cluster starts within a larger supercluster cannot be recognized when scanning to the left ,i.e., in Blocks C and G (of Algorithms 1.a and 1.b). Instead we can only recognize supercluster starts (a non-empty cell to the right of an empty cell).

To write lock a supercluster, we store `01` in the status bits of an otherwise empty slot after the supercluster. To read lock a supercluster we remove the remainder from the table and store it locally until the lock is released (status bits `11`). However, this concurrent variant is not practical because we have to ensure that the read-locked slot is still a supercluster start (the slot to its left remains empty). To do this we can atomically compare and swap both slots. However this is a problem since both slots might be stored in different atomic data types or even cache lines. The variant is still interesting from a theoretical perspective where a compare and swap operation changing two neighboring slots is completely reasonable (usually below 64 bits). However, we have implemented this variant when we did some preliminary testing with non-aligned compare and swap operations and it is non-competitive with the other implementations.

Growing concurrently. The bounded growing technique (described in Section 2) can be used to increase the capacity of a concurrent quotient filter similar to that of a sequential quotient filter. In the concurrent setting we have to consider two things: distributing the work of the migration between threads and ensuring that no new elements are inserted into parts of the old table that were already migrated (otherwise they might be lost).

To distribute the work of migrating elements, we use methods similar to those in Maier et al. [11]. After the migration is triggered, every thread that starts an operation first helps with the migration before executing the operation on the new table. Reducing interactions between threads during the migration is important for performance. Therefore, we migrate the table in blocks. Every thread acquires a block by incrementing a shared atomic variable. The migration of each block happens one supercluster at a time. Each thread migrates all superclusters that begin in its block. This means that a thread does not migrate the first supercluster in its block, if it starts in the previous block. It also means that the thread migrates elements from the next block, if its last supercluster crosses the border to that block. The order of elements does not change during the migration, because they remain ordered by their fingerprint. In general this means that most elements within one block of the original table are moved into one of two blocks in the target table (block i is moved to $2i$ and $2i + 1$). By assigning clusters depending on the starting slot of their supercluster, we enforce that there are no two threads accessing the same slot of the target table. Hence, no atomic operations or locks are necessary in the target table.

As described before, we have to ensure that ongoing insert operations either finish correctly or help with the migration before inserting into the new table. Ongoing queries also have to finish to prevent deadlocks. To prevent other threads from inserting elements during the migration, we write lock each empty slot and each supercluster before it is migrated. These *migration-write-locks* are never released. To differentiate migration-write-locks from the ones used in a normal insertions, we write lock a slot and store a non-zero remainder (write locks are usually only stored in empty slots). This way, an ongoing insertion recognizes that the encountered write lock belongs to a migration. The inserting thread first helps with the migration before restarting the insertion after the table is fully migrated. Queries can happen concurrently with the migration, because the migration does not need read locks.

4 Fully Expandable QFs

The goal of this fully expandable quotient filter is to offer a resizable quotient filter variant with a bounded false positive rate that works well even if there is no known bound to the number of elements inserted. Adding new fingerprint bits to existing entries is impossible without access to the inserted elements. We adapt a technique that was originally introduced for scalable Bloom filters [2]. Once a quotient filter is filled, we allocate a new *additional* quotient filter. Each subsequent quotient filter increases the fingerprint size. Overall, this ensures a bounded false positive rate. This old idea offers new and interesting possibilities when applied to concurrent quotient filters, i.e., avoiding locks on lower levels, growing each level using the bounded growing technique, higher fill degree through cascading inserts, and early rejection of queries also through cascading inserts.

This new data structure starts out with one quotient filter, but over time, it may contain a set of quotient filters we call levels. At any point in time, only the newest (highest) level is active. Insertions operate on the active level. The data structure is initialized with two user-defined parameters the *expected capacity* c and the upper *bound for the false positive rate* $\overline{p}^+$. The first level table is initialized with m_0 slots where $m_0 = 2^{q_0}$ is the first power of 2 where $\delta_{grow} \cdot m_0$ is larger than c, here δ_{grow} is the fill ratio where growing is triggered and $\overline{n_i} = \delta_{grow} \cdot m_i$ is the maximum number of elements level i can hold. The number of remainder bits r_0 is chosen such that $\overline{p}^+ > 2\delta_{grow} \cdot 2^{-r_0}$ ($k_0 = q_0 + r_0$ fingerprint bits).

Queries have to check each level. Within the lower levels queries do not need any locks, because the elements there are finalized. The query performance depends on the number of levels. To keep the number of levels small, we have to increase the capacity of each subsequent level. To also bound the false positive rate, we have to reduce the false positive rate of each subsequent level. To achieve both of these goals, we increase the size of the fingerprint k_i by two for each subsequent level ($k_i = 2 + k_{i-1}$). Using the longer fingerprint we can ensure that once the new table holds twice as many elements as the old one ($\overline{n_i} = \overline{n_{i+1}}/2$), it still has half the false positive rate ($\overline{p}_i^+ = \overline{n_i} \cdot 2^{-k_i} = 2\overline{p}_{i+1}^+ = 2 \cdot \overline{n_{i+1}} \cdot 2^{-k_{i+1}}$).

When one level reaches its maximum capacity $\overline{n_i}$ we allocate a new level. Instead of allocating the new level to immediately have twice the number of slots as the old level, we allocate it with one 8th of the final size, and use the bounded growing algorithm (described in Section 2) to grow it to its final size (three growing steps). This way, the table has a higher average fill rate (at least $2/3 \cdot \delta_{grow}$ instead of $1/3 \cdot \delta_{grow}$).

▶ **Theorem 2** (Bounded p^+ in expandable QF)**.** *The fully expandable quotient filter holds the false positive probability $\overline{p}^+$ set by the user independently of the number of inserted elements.*

Proof. For the following analysis, we assume that fingerprints can potentially have an arbitrary length. The analysis of the overall false positive rate p^+ is very similar to that of the scalable Bloom filter. A false positive occurs if one of the ℓ levels has a false positive $p^+ = 1 - \prod_i (1 - p_i^+)$. This can be approximated with the Weierstrass inequality $p^+ \leq \sum_{i=1}^{\ell} p_i^+$. When we insert the shrinking false positive rates per level ($p_{i+1}^+ = p_i^+/2$) we obtain a geometric sum which is bounded by $2p_1^+$: $\sum_{i=0}^{\ell-1} p_i^+ 2^{-i} \leq 2p_1^+ < \overline{p}^+$. ◀

Using this growing scheme the number of filters is in $O(\log n/T)$, therefore, the bounds for queries are similar to those in a broad tree data structure. But due to the necessary pointers tree data structures take significantly more memory. Additionally, they are difficult to implement concurrently without creating contention on the root node.

Cascading Inserts. Cascading Inserts can be used to improve the memory usage and the query performance of our growing quotient filters. The idea is to insert elements on the lowest possible level. If the canonical slot in a lower level is empty, we insert the element into that level. This can be done using a simple CAS-operation (without acquiring a write lock). Queries on lower levels can still proceed without locking, because insertions cannot move existing elements.

The main reason to grow the table before it is full is to improve the performance by shortening clusters. The trade-off for this is space utilization. For the space utilization it would be optimal to fill each table 100%. Using cascading inserts this can be achieved while still having a good performance on each level. Queries on the lower level tables have no significant slow down due to cascading inserts, because the average cluster length remains small (cascading inserts lead to one-element clusters). Additionally, if we use cascading inserts, we can abort queries that encounter an empty canonical slot in one of the lower level tables, because this slot would have been filled by an insertion. Cascading inserts cause some overhead. Each insertion checks every level whether its canonical slot is empty. However, in some applications checking all levels is already necessary. For example in applications like element unification all lower levels are checked to prevent repeated insertions of one element. Applications like this often use combined query and insert operations that only insert if the element was not yet in the table. Here cascading inserts do not cost any overhead.

5 Experiments

All experiments were executed on a four socket Intel Xeon Gold 6138 machine with 20 cores per socket, each running at 2.0 GHz with 27.5MB cache size, and 768 GB main memory. The tests were compiled using gcc 9.2.0 and the operating system is Ubuntu 18.04.3. Each test was repeated 9 times using 3 different sequences of keys (same distribution) – 3 runs per sequence.

The first experiment we conducted is a speedup benchmark with strong work scaling between 1 to 160 threads (with hyperthreading) that can be seen in our technical report [12]. We compare our locally locked quotient filter (using status bit locking) and the lock-free linear probing filter with other AMQ data structures like a standard (externally) locked quotient filter, the counting quotient filter by Pandey et al. [14,16], and a specifically optimized bloom filter. The bloom filter was optimized by adapting its size to the size of the other tested data structures and in turn reducing the number of hash functions k such that a similar false positive rate was achieved ($k = 4$). We feel that this is the fairest comparison to the presented data structures (same memory; similar p^+). The results are fairly expected (linear scaling performance).

Fill Ratio Benchmark. In this benchmark, we test the same 5 filter implementations under a varying fill degree. To do this, we fill a table with $2^{30} \approx 1.1$G slots and $r = 10$. Every 10% of the fill ratio we execute a performance test the results are shown in Figure 2. During this test each table operation is executed 1M times. As one would expect, the throughput of our quotient filter variants decreases with increasing fill ratio. However, this seems not to be the case for the counting quotient filter and the Bloom filter. The optimized bloom filter that we used in these experiments is especially good at negative queries, because its density of 1s is actually very small, therefore, queries can often be aborted after one memory access. The insert performance of the counting quotient filter is at below 0.2MOps (independent of the fill degree). Our advanced quotient filter implementations – the linear probing quotient filter and

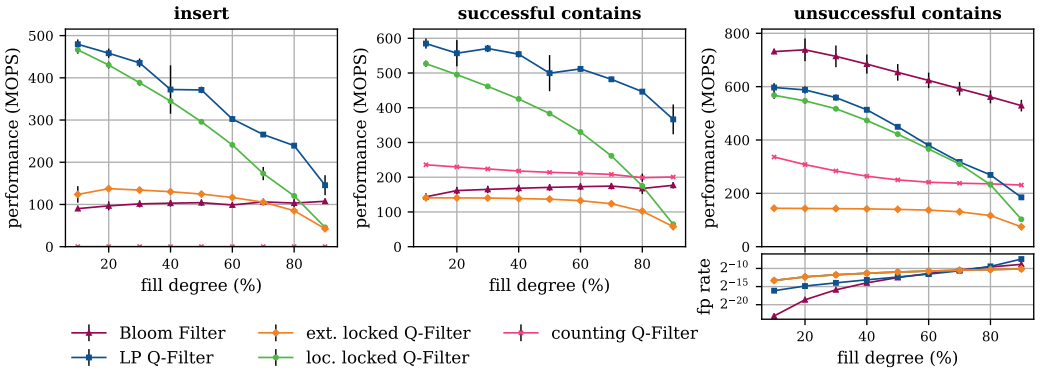


Figure 2 Throughput over fill degree. Using $p = 80$ threads and $2^{30} \approx 1.1$G slots. Each point of the measurement was tested using 1M operations (in parallel).

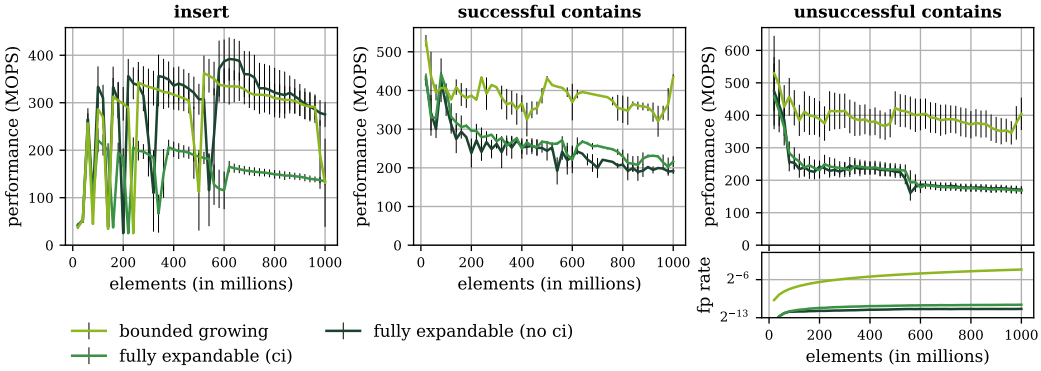


Figure 3 Throughput of the growing tables (all variants based on the local locking quotient filter). Using $p = 80$ threads 50 segments of 20M elements are inserted into a table initialized with $2^{24} \approx 16.8$M slots.

the locally locked quotient filter – display their strengths on sparser tables with about 3.25× and 2.89× higher insertion throughputs than the similar externally locked implementation at 30% fill degree. At 70% fill degree the advanced implementation are still 2.51× and 1.64× faster than the externally locked implementation.

Growing Benchmark. For this benchmark, shown in Figure 3, we insert 1G elements into each of our dynamically sized quotient filters bounded growing, fully expandable without cascading inserts (no ci), and fully expandable with cascading inserts (ci) (all based on the quotient filter using local locking). The filter was initialized with a capacity of only $2^{24} \approx 16.8$M (64× growing factor) slots and a target false positive rate of $\overline{p}^{+} = 2^{-10}$. The inserted elements are split into 50 segments of 20M elements each. We measure the running time of inserting each segment as well as the query performance after each segment.

As expected, the false positive rate of the quotient filter that uses bounded growing grows linearly with the number of elements, but each query still consists of only one table lookup – resulting in a query performance similar to that of the non-growing benchmark. Both fully expandable variants that combine bounded growing with the multi level technique stay below 2^{-10} false positive rate. But their query performance suffers due to the lower

level look ups and the additional memory accesses they incur. Cascading inserts can improve successful query times by over 10% (averaged over all segments), however, for unsuccessful queries we do not see an average speedup. As expected cascading inserts are slowing down insertion times significantly. AMQ-filter data structures are usually queried significantly more often than they are updated, therefore, many workloads in practice can still profit from this technique.

6 Conclusion

In this publication, we have shown a new technique for concurrent quotient filters that uses the status bits inherent to quotient filters for localized locking. Using this technique, no additional cache lines are accessed (compared to a sequential quotient filter). We were able to achieve a 1.6 times increase in insert performance over the external locking scheme ($p = 80$ and 70% fill degree; 2.1 and 2.4 on queries). Additionally, we proposed a simple linear probing based filter that does not use any status bits and is lock-free. Using the same amount of memory this filter achieves even better false positive rates up to a fill degree of 70% and also 1.5 times higher insertion speedups (than the local locking variant also at 70% fill degree).

We also propose to use the bounded growing technique which is supported by quotient filters to refine the growing scheme used in scalable Bloom filters. Using this combination of techniques guarantees that the overall data structure is always at least $2/3 \cdot \delta_{grow}$ filled (where δ_{grow} is the fill degree where growing is triggered). Using cascading inserts this can be improved by filling lower level tables even further, while also improving successful query times by up to 10%.

Our tests show that there is no optimal AMQ data structure. Which data structure performs best depends on the use case and the expected workload. The linear probing quotient filter is very good, if the table is not densely filled. The locally locked quotient filter is also efficient on tables below a fill degree of 70%. But, it is also more flexible for example when the table starts out empty and is filled to above 70% (i.e., constructing the filter). Our growing implementations work well if the number of inserted elements is not known prior to the table's construction. The counting quotient filter could perform well on very query heavy workloads that operate on densely filled tables.

There is some future work that could further improve the presented data structures. Two obvious improvements would be the implementation of deletions or the possibility to store element counters (similar to [16]). Additionally, it would be interesting to use hardware transactional memory to reduce the number of necessary locks even further. This has the possibility to speed up queries in particular.

References

1 Mohammad Al-hisnawi and Mahmood Ahmadi. Deep Packet Inspection using Quotient Filter. *IEEE Communications Letters*, 20(11):2217–2220, November 2016. doi:10.1109/LCOMM.2016.2601898.

2 Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable Bloom Filters. *Inf. Process. Lett.*, 101(6):255–261, March 2007. doi:10.1016/j.ipl.2006.10.007.

3 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, July 2012. doi:10.14778/2350229.2350275.

4 Alex D. Breslow and Nuwan S. Jayasena. Morton filters: fast, compressed sparse cuckoo filters. *The VLDB Journal*, August 2019. doi:10.1007/s00778-019-00561-0.

5 Pedro Celis, Per-Ake Larson, and J Ian Munro. Robin Hood Hashing. In *26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 281–288. IEEE, 1985.

6 J. G. Cleary. Compact Hash Tables Using Bidirectional Linear Probing. *IEEE Transactions on Computers*, C-33(9):828–834, 1984.

7 Yan Collet. xxHash. `https://github.com/Cyan4973/xxHash`. Accessed March 21, 2019.

8 Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo Filter: Practically better than Bloom. In *10th ACM International on Conference on Emerging Networking Experiments and Technologies*, pages 75–88, 2014. `doi:10.1145/2674005.2674994`.

9 Afton Geil, Martin Farach-Colton, and John D. Owens. Quotient Filters: Approximate membership queries on the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–462, May 2018. `doi:10.1109/IPDPS.2018.00055`.

10 Donald Ervin Knuth. *The Art of Computer Programming: sorting and searching*, volume 3. Pearson Education, 1997.

11 Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent Hash Tables: Fast and general(?)! *ACM Trans. Parallel Comput.*, 5(4):16:1–16:32, February 2019. `doi:10.1145/3309206`.

12 Tobias Maier, Peter Sanders, and Robert Williger. Concurrent Expandable AMQs on the basis of quotient filters. *CoRR*, 2019. `arXiv:1911.08374`.

13 Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive Cuckoo Filters. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 36–47. SIAM, 2018. `doi:10.1137/1.9781611975055.4`.

14 Prashant Pandey. Counting Quotient Filter. `https://github.com/splatlab/cqf`. Accessed August 07, 2019.

15 Prashant Pandey, Fatemeh Almodaresi, Michael A. Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell Systems*, 7(2):201–207.e4, 2018. `doi:10.1016/j.cels.2018.05.021`.

16 Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A General-Purpose Counting Filter: Making every bit count. In *ACM Conference on Management of Data (SIGMOD)*, pages 775–787, 2017. `doi:10.1145/3035918.3035963`.

Faster Multi-Modal Route Planning With Bike Sharing Using ULTRA

Jonas Sauer
Karlsruhe Institute of Technology, Germany
jonas.sauer2@kit.edu

Dorothea Wagner
Karlsruhe Institute of Technology, Germany
dorothea.wagner@kit.edu

Tobias Zündorf
Karlsruhe Institute of Technology, Germany
zuendorf@kit.edu

Abstract

We study multi-modal route planning in a network comprised of schedule-based public transportation, unrestricted walking, and cycling with bikes available from bike sharing stations. So far this problem has only been considered for scenarios with at most one bike sharing operator, for which MCR is the best known algorithm [6]. However, for practical applications, algorithms should be able to distinguish between bike sharing stations of multiple competing bike sharing operators. Furthermore, MCR has recently been outperformed by ULTRA for multi-modal route planning scenarios without bike sharing [3]. In this paper, we present two approaches for modeling multi-modal transportation networks with multiple bike sharing operators: The *operator-dependent* model requires explicit handling of bike sharing stations within the algorithm, which we demonstrate with an adapted version of MCR. In the *operator-expanded* model, all relevant information is encoded within an expanded network. This allows for applying any multi-modal public transit algorithm without modification, which we show for ULTRA. We proceed by describing an additional preprocessing step called *operator pruning*, which can be used to accelerate both approaches. We conclude our work with an extensive experimental evaluation on the networks of London, Switzerland, and Germany. Our experiments show that the new preprocessing technique accelerates both approaches significantly, with the fastest algorithm (ULTRA-RAPTOR with operator pruning) being more than an order of magnitude faster than the basic MCR approach. Moreover, the ULTRA preprocessing step also benefits from operator pruning, as its running time is reduced by a factor of 14 to 20.

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

Keywords and phrases Algorithms, Route Planning, Bike Sharing, Public Transportation

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.16

Supplementary Material Source code is available at `https://github.com/kit-algo/Bike-Sharing`.

Funding This research was funded by the DFG under grant number WA 654123-2.

1 Introduction

Research on efficient public transit route planning has seen remarkable advances in the past decade [2]. However, in practice, public transit is most often used in combination with other transportation modes, such as walking or cycling. This results in a multi-modal route planning scenario, which is still a challenge to solve efficiently [22]. In this work we present novel and efficient approaches for finding all Pareto-optimal journeys (regarding travel time and number of used public transit vehicles) in a multi-modal network featuring public transit, unrestricted walking, and bicycles available at bike sharing stations from multiple operators.

Related Work. Several algorithms have been proposed for scenarios where only one of the aforementioned modes of transportation is available. Walking (or cycling) on its own results in a classical shortest path problem, which can be solved using Dijkstra's algorithm [10]. The fastest known algorithms for this problem utilize an additional preprocessing phase to achieve fast query times. One such approach is Contraction Hierarchies (CH) [11], where the preprocessing computes shortcuts that allow the query to skip unimportant vertices. A comprehensive overview of speedup techniques for the shortest path problem is given in [2]. For the special case of route planning for bicycles, an extended scenario where travel time is optimized while constraining the altitude difference has been considered and accelerated with CH [21].

Algorithms for public transit networks can be divided into two groups: graph-based algorithms and algorithms operating directly on the timetable. Graph-based approaches can be further subdivided into *time-dependent* and *time-expanded* techniques, which both enable the usage of normal shortest path algorithms [20, 19]. These techniques have also been adapted for additional optimization criteria, such as number of used vehicles or price [18]. However, accelerating these graph-based approaches is quite difficult [4]. Many efficient algorithms therefore operate directly on the timetable and try to exploit its structure. Examples for such approaches are Transfer Patterns [1], Trip-Based Routing [23], CSA [8], and RAPTOR [7].

While route planning for walking, cycling, or public transit is quite well studied, the combined multi-modal problem still holds many challenges. As before, one possible approach is to model all aspects of the multi-modal network as a graph [13, 15]. In [16], a time-dependent approach was combined with ALT [14] in order to improve query times. Accelerating a graph-based search with CH yields even faster queries while also allowing for user-specific transfer mode preferences [9]. Another approach is to heuristically reduce the search space in order to obtain fast query times [5]. The only timetable-based technique that has been adapted for multi-modal scenarios is RAPTOR, resulting in the MCR algorithm [6]. Most recently, ULTRA (short for UnLimited Transfers) [3] has been presented, which uses a preprocessing step to compute shortcuts for non-timetable-based modes of transportation. These shortcuts can then be used to enable multi-modal queries for many timetable-based algorithms.

Most of the multi-modal algorithms also consider bike sharing as one of the available transportation modes. However, to the best of our knowledge, no algorithm so far has considered the more realistic scenario of multiple competing bike sharing operators.

Contribution. In this work we present two distinct approaches for solving the journey planning problem in multi-modal transportation networks with several bike sharing operators. For the first approach, which we call the operator-dependent model, we show how the well-known MCR algorithm can be adapted for our scenario. We then show that the query time of every label-propagating algorithm (such as MCR) is proportional to the number of bike sharing operators present in the network. This theoretical analysis is later confirmed by experiments. The second approach encodes the bike sharing information into an enlarged transfer graph and public transit schedule. Using this operator-expanded network eliminates the need for specialized algorithms. In particular, this enables us to use ULTRA, the fastest known multi-modal algorithm, for journey planning with bike sharing. For even faster queries we propose an additional preprocessing technique called operator pruning, which can be used with both models. This technique not only reduces the search space and running time of the query algorithms significantly, but also reduces the preprocessing time of ULTRA by

more than an order of magnitude. We conclude our work with an extensive experimental evaluation of our algorithms. We show that the combination of ULTRA-RAPTOR and operator pruning on the operator-expanded model yields the fastest algorithm on real-world networks.

2 Preliminaries

This section introduces the basic notation used throughout this work, as well as the problem statement and the algorithms on which our work is based.

Public Transit Network. A public transit network is a tuple $N = (\mathcal{S}, \mathcal{T}, \mathcal{R}, G)$ consisting of a set of *stops* $\mathcal{S}$, a set of *trips* $\mathcal{T}$, a set of *routes* $\mathcal{R}$ and a directed, weighted *transfer graph* $G = (\mathcal{V}, \mathcal{E})$. A stop $v \in \mathcal{S}$ is a location in the network where passengers can enter or exit a vehicle (e.g., a train, bus, ferry, etc.). Associated with each stop is a non-negative *departure buffer time* $\tau_{\text{buf}}(v)$, which is the minimum amount it takes for a passenger to board a vehicle after arriving at the stop. A trip $\mathrm{T} = \langle v_0, \ldots, v_k \rangle \in \mathcal{T}$ is a sequence of at least two stops which are served consecutively by the same vehicle. For each stop v in the trip, we denote the *arrival time* of the vehicle at v by $\tau_{\text{arr}}(\mathrm{T}, v)$ and the *departure time* by $\tau_{\text{dep}}(\mathrm{T}, v)$. The trips are partitioned into routes $r \in \mathcal{R}$, such that two trips are part of the same route if their stop sequence is identical and they do not overtake each other. A trip $\mathrm{T}_a \in \mathcal{T}$ overtakes a trip $\mathrm{T}_b \in \mathcal{T}$ if their stop sequence contains two stops $u, v \in \mathcal{S}$ such that T_a arrives at or departs from u before T_b, but arrives at or departs from v after T_b. The transfer graph $G = (\mathcal{V}, \mathcal{E})$ consists of a set of *vertices* $\mathcal{V}$ with $\mathcal{S} \subseteq \mathcal{V}$, and a set of *edges* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. Associated with each edge $e = (u, v) \in \mathcal{E}$ is a *walking time* $\tau_{\text{walk}}(e)$, which is the time required to walk from u to v.

Bike Sharing. In addition to walking, we consider bicycle sharing as a second transfer mode. For each edge $e = (u, v) \in \mathcal{E}$, we define the *biking time* $\tau_{\text{bike}}(e)$ as the time required to travel from u to v with a bicycle. Note that $\tau_{\text{walk}}(e)$ or $\tau_{\text{bike}}(e)$ may be ∞ to signify that e is not usable in the respective transfer mode. Some trips in the public transit network may allow passengers to carry along a bicycle with them, while others may not. The *bicycle transport function* $\mathrm{bt} \colon \mathcal{T} \to \{\texttt{true}, \texttt{false}\}$ maps to each trip $\mathrm{T} \in \mathcal{T}$ a boolean value $\mathrm{bt}(\mathrm{T})$ that indicates whether T allows bicycle transport or not.

Bikes can be rented from a number of different *bike sharing operators*. We denote the number of bike sharing operators by o and associate each operator with a number from $\{1, \ldots, o\}$. For simplicity, we will use the number 0 to denote that a passenger is currently not renting a bike. Each operator i operates a set $\mathcal{BS}_i \subseteq \mathcal{V}$ of *bike sharing stations* where passengers can pick up or drop off a bike. Note that a vertex may act as a bike sharing station for more than one operator. Each bike sharing station v has an associated *pickup time* $\tau_{\text{pick}}(v)$ that is required to pick up a bike, and a *dropoff time* $\tau_{\text{drop}}(v)$ that is required to drop off a bike. A passenger may only carry one bike at a time.

Journey. A *journey* J defines the movement of a passenger through the public transit network when traveling from a source vertex $s \in \mathcal{V}$ to a target vertex $t \in \mathcal{V}$. It is an alternating sequence of *trip legs* and *transfers*, where a trip leg is a subsequence of a trip that represents the passenger using that portion of the trip, and a transfer is a path in the transfer graph that connects the final stop of one trip leg (or s for the initial transfer) with the first stop of the following trip leg (or t for the final transfer). Note that some or all of the transfers may be empty.

To define which parts of the journey use bike sharing, the journey is augmented with a sequence $((u_1, v_1), \ldots, (u_n, v_n))$ that contains one tuple of bike sharing stations for each bike that is rented during the journey. For the i-th rented bike, u_i is the station where the bike is picked up and v_i is the station where it is dropped off. It is required that there is a bike sharing operator j for which $u_i, v_i \in \mathcal{BS}_j$ holds. This ensures that multiple bikes are not rented at the same time, and that the journey starts and ends with no bike rented.

The bike sharing stations must be visited by the transfers of the journey in the same order in which they appear in the sequence. If u_i and v_i are visited during different transfers, all trip legs that lie between these two transfers must allow bicycle transport. For every transfer edge $e \in \mathcal{E}$ used between u_i and v_i, the travel time for using the edge is $\tau_{\text{bike}}(e)$. For every edge $e \in \mathcal{E}$ that is traversed without a bike, the travel time is $\tau_{\text{walk}}(e)$.

Problem Statement. The criteria we use to evaluate a journey J are its arrival time at the target vertex t and the number of used public transit vehicles, i.e., the number of trip legs in J. A journey J *dominates* another journey J' if J arrives no later than J' and does not use more trips than J'. We call a journey J *Pareto-optimal* if it is not dominated by any other journey. A *Pareto set* is a set containing a minimal number of journeys such that every valid journey is dominated by a journey in the set. Naturally, all journeys in a Pareto set are Pareto-optimal. Given source and target vertices $s, t \in \mathcal{V}$ and an earliest departure time τ_{dep}, our objective is to find a Pareto set among all journeys from s to t that depart no later than τ_{dep}.

Algorithms. The algorithms we present in this work are based on RAPTOR [7], which is an algorithm designed to find journeys that optimize arrival time and number of trips on a public transit network with a transitively closed transfer graph. It operates in rounds, where round i finds journeys that use exactly i trips. To achieve this, each round extends journeys found in the previous round by performing a single scan over all routes that are incident to stops reached in the previous round. Transfers are explored between rounds by iterating across the outgoing edges of each reached stop.

Several extensions of RAPTOR have been proposed: McRAPTOR is able to optimize additional criteria by replacing the single label that RAPTOR stores per stop and round with a *bag* of labels that do not dominate each other. rRAPTOR answers *profile queries*, which ask for optimal journeys across an interval of possible departure times, rather than just a single departure time. This is done by performing one iteration of the basic RAPTOR algorithm for each possible departure time, in descending order. Journeys found in earlier iterations can be used to prune non-optimal journeys in later iterations, a technique called *self-pruning*. The MCR [6] algorithm extends (Mc)RAPTOR to work on unrestricted transfer graphs that are not transitively closed. This is done by replacing the edge scans between rounds with Dijkstra searches on a *core graph* created via a partial CH contraction of the transfer graph.

A faster alternative to MCR is the ULTRA [3] preprocessing technique, which precomputes a small number of *transfer shortcuts* that are provably sufficient for computing all Pareto-optimal journeys. Any public transit algorithm that normally requires a transitively closed transfer graph can then use these shortcuts to explore intermediate transfers of a journey. Initial and final transfers are handled separately via Bucket-CH [17, 11, 12], a fast technique for one-to-many queries on road networks. Overall, ULTRA extends public transit algorithms such as RAPTOR to unlimited transfers without a significant performance loss.

3 Algorithms

In this chapter we introduce new algorithms for solving the multi-modal route planning problem with multiple bike sharing operators. We propose two approaches for modeling bike sharing in a public transit network: First we show how MCR can be adapted to handle renting and returning of bicycles explicitly within the algorithm. We call this approach the *operator-dependent* model. We then introduce the *operator-expanded* model, where all relevant information regarding bike sharing is encoded directly in the network. This allows any existing multi-modal public transit algorithm to handle bike sharing without modifications. We demonstrate this on the example of ULTRA. Finally, we introduce a preprocessing technique called *operator pruning* to speed up queries in both models, and describe how it can be incorporated into MCR and ULTRA.

3.1 Operator-Dependent Model/MCR

Most public transit algorithms, including MCR, work by propagating vertex labels through the network. For the two criteria arrival time and number of trips, a label at a vertex v can be represented as a tuple (τ_{arr}, k), where τ_{arr} is the arrival time at v, and k is the number of trips used so far. Bike sharing can be incorporated by extending the labels to triples $(\tau_{\text{arr}}, k, i)$, where i is the operator of the currently rented bike (or 0 if no bike is rented). Since rented bikes have to be dropped off before the end of the journey, only labels at the target vertex t with operator 0 represent complete journeys. Whenever a label $(\tau_{\text{arr}}, k, 0)$ reaches a bike sharing station v, a new label $(\tau_{\text{arr}} + \tau_{\text{pick}}(v), k, i)$ must be created for each operator i with $v \in \mathcal{BS}_i$, to represent the passenger picking up a bike of operator i. Similarly, when a label $(\tau_{\text{arr}}, k, i)$ with $i \neq 0$ reaches a bike sharing station $v \in \mathcal{BS}_i$, a new label $(\tau_{\text{arr}} + \tau_{\text{drop}}(v), k, 0)$ must be created to represent the passenger dropping off the bike. The time required to traverse an edge $e \in \mathcal{E}$ is $\tau_{\text{bike}}(e)$ for labels with a rented bike and $\tau_{\text{walk}}(e)$ otherwise. When propagating a label with operator $i \neq 0$ through a route, any trip T that does not permit bicycle transport (i.e., bt(T) = `false`) must be ignored.

Without bike sharing, a label (τ_{arr}, k) may dominate another label (τ'_{arr}, k') if $\tau_{\text{arr}} \leq \tau'_{\text{arr}}$ and $k \leq k'$ hold. The same dominance rule still applies to labels with the same bike sharing operator: A label $(\tau_{\text{arr}}, k, i)$ dominates a label $(\tau'_{\text{arr}}, k', i)$ if $\tau_{\text{arr}} \leq \tau'_{\text{arr}}$ and $k \leq k'$ hold. However, as the following lemma shows, it is not possible to establish dominance rules for labels with different operators:

▶ **Lemma 1.** *Let $A = (\tau_{arr}, k, i)$ and $B = (\tau'_{arr}, k', j)$ be two labels at some vertex v with $\tau_{arr} \leq \tau'_{arr}$, $k \leq k'$, and $i \neq j$. Then A may not dominate B.*

Proof. If $i = 0$, label B has access to a bike and A does not. Using the bike may allow the journey represented by B to overtake the journey represented by A and reach the target faster. If $i \neq 0$, then the passenger represented by A is carrying a bike, which must be returned before reaching the target. This may require a detour that may not be required for B, possibly allowing the journey represented by B to overtake and reach the target faster. ◀

Thus, bike sharing can be incorporated into the Pareto optimization by treating the bike sharing operator as a third criterion whose values are all incomparable with each other.

In MCR, the number of trips is not stored directly in the labels but unrolled into the round data structure. For the MR-∞ variant of MCR, which only optimizes arrival time and number of trips, it is sufficient to store a single arrival time per vertex and round. Variants with additional criteria replace the single arrival time with a bag of non-dominated labels. When a new label is added, it must be compared to all other labels in the bag to eliminate

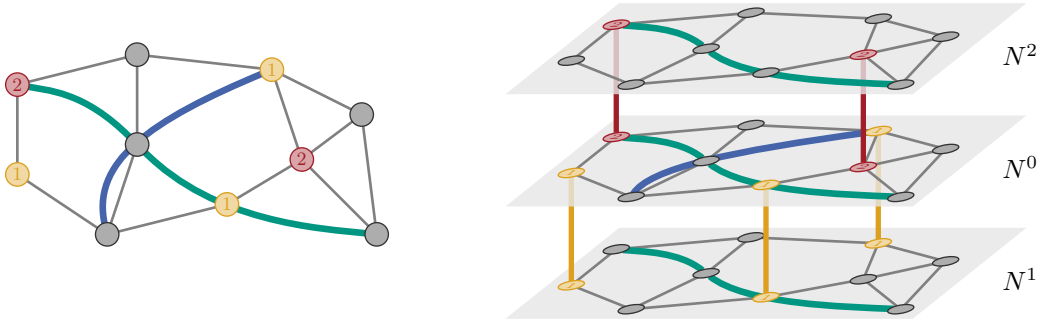


Figure 1 A public transit network (left) and the corresponding operator-expanded network (right). The network features two public transit routes, with the trips of the green route allowing bicycle transport and the trips of the blue route disallowing it. Bikes can be rented and returned at the three bike-sharing stations of operator 1 (yellow) or at the two stations of operator 2 (red).

dominated labels. A naive approach to incorporating bike sharing would be to store bags of labels with two criteria: arrival time and bike sharing operator. However, like the number of transfers, the operator criterion is discrete and only permits a few possible values. Thus, it is more efficient to unroll it into the data structure as well: For each vertex and round, we store an array $(\tau^0_{\text{arr}}, \dots, \tau^o_{\text{arr}})$, where τ^i_{arr} is the best arrival time achieved so far with operator i. As shown by Lemma 1, a new label with operator i only needs to be compared with τ^i_{arr}, since it is incomparable to the other entries. Thus, we can handle all operators independently of each other and do not need bags at all. In each round, we perform $o+1$ independent route scanning phases, where phase i only considers labels with operator i. The resulting algorithm is a variant of MR-∞ whose worst-case running time is proportional to that of the original MR-∞ and $o+1$.

3.2 Operator-Expanded Network/ULTRA

One drawback of the operator-dependent model is that it requires algorithms to be explicitly adapted for bike sharing. An alternative is to unroll the bike sharing information into an enlarged public transit network. Any existing multi-modal public transit algorithm can then handle bike sharing without modification by operating on this *operator-expanded* network.

Given an original public transit network N, the operator-expanded network N^{e} consists of $o+1$ copies $N^0, \dots, N^o$ of N. The idea is that N^i is used by passengers who are currently renting a bike from operator i, with N^0 representing walking. Accordingly, the weight of an edge $e \in \mathcal{E}$ is $\tau_{\text{walk}}(e)$ if it is part of N^0 and $\tau_{\text{bike}}(e)$ otherwise. In all copies N^i with $i \neq 0$, trips that do not allow bicycle transport are removed. The network copies are connected as follows: For a vertex $v \in \mathcal{V}$ in the original network, we denote its copy in network N^i by v^i. For each operator i and each bike sharing station $v \in \mathcal{BS}_i$, the expanded network includes the edges (v^0, v^i) with weight $\tau_{\text{pick}}(v)$ and (v^i, v^0) with weight $\tau_{\text{drop}}(v)$. These edges represent picking up and dropping off a bike, respectively.

Let $\mathcal{OP} = \{0, 1, \dots, o\}$ be the set of bike sharing operators. We then define the operator-expanded network formally as $N^{\text{e}} = (\mathcal{S}^{\text{e}}, \mathcal{T}^{\text{e}}, \mathcal{R}^{\text{e}}, G^{\text{e}} = (\mathcal{V}^{\text{e}}, \mathcal{E}^{\text{e}}))$, with

$$
\begin{aligned}
\mathcal{S}^{\text{e}} &= \{v^i \mid i \in \mathcal{OP} \wedge v \in \mathcal{S}\},\\
\mathcal{T}^{\text{e}} &= \{\mathrm{T}^i = \langle v^i_0, \dots, v^i_k \rangle \mid i \in \mathcal{OP} \wedge \mathrm{T} = \langle v_0, \dots v_k \rangle \in \mathcal{T} \wedge (\mathrm{bt}(\mathrm{T}) \vee i = 0)\},\\
\mathcal{R}^{\text{e}} &= \{\{\mathrm{T}^i \mid \mathrm{T} \in r \wedge (\mathrm{bt}(\mathrm{T}) \vee i = 0)\} \mid i \in \mathcal{OP} \wedge r \in \mathcal{R}\},\\
\mathcal{V}^{\text{e}} &= \{v^i \mid i \in \mathcal{OP} \wedge v \in \mathcal{V}\},\\
\mathcal{E}^{\text{e}} &= \{(v^i, w^i) \mid i \in \mathcal{OP} \wedge (v, w) \in \mathcal{E}\} \cup \{(v^0, v^i), (v^i, v^0) \mid i \in \mathcal{OP} \setminus \{0\} \wedge v \in \mathcal{BS}_i\}.
\end{aligned}
$$

An example of how the operator-expanded network is constructed is shown in Figure 1.

An s-t-query on the original network can be solved with an s^0-t^0-query on the operator-expanded network, using an unmodified multi-modal public transit algorithm (such as MCR) that no longer needs to handle bike sharing explicitly. For ULTRA, both the preprocessing and the query algorithm can be run on the operator-expanded network. The preprocessing, which normally performs profile searches between all vertices, only needs to perform profile searches between the vertices in N^0. As with MCR, ULTRA contracts the transfer graph before the preprocessing is performed. A naive approach would be to create the operator-expanded network first and then contract the expanded transfer graph. However, since the transfer graphs of all networks N^i with $i \neq 0$ are identical, this would lead to redundant work. Instead, it is more efficient to contract two copies of the original transfer graph: one with walking weights and one with biking weights. Bike sharing stations are left uncontracted at this point. These contracted copies can then be inserted into the operator-expanded network in place of the original graph. If desired, an additional contraction can then be performed on the resulting transfer graph of the operator-expanded network.

The ULTRA query consists of two phases: The Bucket-CH search for the initial and final transfers is done on the transfer graph of the operator-expanded network, taking care of bike renting automatically. The main public transit algorithm (e.g., RAPTOR) uses the operator-expanded network with the shortcuts computed by the preprocessing phase. A shortcut (u^i, v^j) corresponds to a shortcut (u, v) in the original network that requires a bike from operator i to access and ends with the passenger having rented a bike from operator j.

3.3 Operator Pruning

A crucial observation that can be used to speed up bike sharing queries is that bike sharing operators typically only serve a limited region (e.g., a single city). Taking a rented bike far outside that region is typically not useful, since the passenger would eventually need to travel back in order to return the bike. Consider a journey that involves taking a train from region A served by operator i to region B served by operator j. If the passenger has rented a bike from operator i, it is usually preferable to return it before taking the train, and then rent a bike from operator j in region B if necessary. However, because labels with different operators may not dominate each other, MCR will also continue exploring the option where the passenger takes the bike from operator i into region B, even though it cannot be returned there. Without additional information, the algorithm will not be aware that the only way to turn this into a valid journey is to travel back to region A, return the bike and then come back to region B, creating an unnecessary detour. To prevent this, we compute an *operator hull* for each operator i, which is a region of the network outside of which it is never useful to travel with a bike from operator i. This allows algorithms to prune journeys once they leave the hull for operator i while carrying a bike from operator i.

Preprocessing. For the hull computation, we define the *cycling network* as $N^{\mathrm{c}} = (\mathcal{S}, \mathcal{T}^{\mathrm{c}}, \mathcal{R}^{\mathrm{c}}, G)$ with $\mathcal{T}^{\mathrm{c}} = \{\mathrm{T} \in \mathcal{T} \mid \mathrm{bt}(\mathrm{T}) = \texttt{true}\}$, $\mathcal{R}^{\mathrm{c}} = \{r \in \mathcal{R} \mid \exists \mathrm{T} \in r\colon \mathrm{bt}(\mathrm{T}) = \texttt{true}\}$, and $\tau_{\mathrm{bike}}(e)$ as the weight for each edge $e \in \mathcal{E}$. This is the network as it appears to a passenger who is using a bike for the entirety of the journey. The *operator hull* $H^i = (\mathcal{V}^i, \mathcal{T}^i)$ for an operator i consists of a set of vertices $\mathcal{V}^i \subseteq \mathcal{V}$ and a set of trips $\mathcal{T}^i \subseteq \mathcal{T}$ such that every journey in N^{c} between bike sharing stations $s, t \in \mathcal{BS}_i$ is dominated by a journey in N^{c} that only uses vertices in $\mathcal{V}^i$ and trips in $\mathcal{T}^i$. It can be computed by running a profile variant of MCR (without bike sharing) on N^{c} from each station $s \in \mathcal{BS}_i$ and unpacking all found journeys ending at another station $t \in \mathcal{BS}_i$. The individual profile searches can be sped up with a simple pruning rule: A profile search from a source station s starts by exploring the initial transfers, computing for each vertex $v \in \mathcal{V}$ the biking time $\tau_{\mathrm{bike}}(s, v)$ from s to v.

From this we can compute $\tau_{\text{bike}}^{\max} := \max_{t \in \mathcal{BS}_i} \tau_{\text{bike}}(s,t)$, which is the maximum biking time to any other bike sharing station of the same operator. Since no optimal journey in N^{c} that ends at a station in $\mathcal{BS}_i$ may be longer than this, the profile search can prune labels whose travel time exceeds $\tau_{\text{bike}}^{\max}$.

Note that the operator hull is an overapproximation of the region outside of which it is never useful to travel with a bike: Journeys that are optimal in N^{c} and therefore contribute to the operator hull may be dominated by journeys that require switching between different bike operators or dropping off a bike to take a trip without bicycle transport. These journeys could be excluded from the hull by using MCR with bike sharing for the hull computation. While this might lead to smaller hulls, it would require significantly higher computation times. Moreover, because the hull computation for one operator would no longer be independent of the other operators, any change in the set of bike sharing stations for one operator would necessitate recomputing all hulls, rather than just the one for the changed operator.

Combination with MCR. Incorporating operator pruning into MCR is straightforward: During the Dijkstra searches, labels with operator i are not propagated to vertices that are not in $\mathcal{V}^i$. Similarly, during the route scanning phase for operator i, routes with no trips in $\mathcal{T}^i$ are ignored and arrival times at stops not in $\mathcal{V}^i$ are not updated. While it would be possible to skip trips that are not in $\mathcal{T}^i$ during the individual route scans, this has no performance benefit since RAPTOR always scans routes until the last stop. Thus, if a trip is skipped, the route scan will simply continue with the next reachable trip.

Combination with Operator-Expanded Network/ULTRA. Given an operator hull $H^i = (\mathcal{V}^i, \mathcal{T}^i)$, we define the *hull network* $N_H^i = (\mathcal{S}^i, \mathcal{T}^i, \mathcal{R}^i, G^i)$ with $\mathcal{S}^i = \mathcal{S} \cap \mathcal{V}^i$, $\mathcal{R}^i \subseteq \mathcal{R}$ the set of routes for which at least one trip is contained in $\mathcal{T}^i$, and G^i the subgraph of G that is induced by $\mathcal{V}^i$. In order to incorporate operator pruning into the operator-expanded network, we first compute operator hulls on the original, non-expanded network. For each operator $i \neq 0$, we then replace the network copy N^i with the hull network N_H^i. The network copy N^0 that represents walking is left unchanged. In the resulting network, leaving the operator hull for the currently rented bike is no longer possible because the corresponding parts of the network have been deleted. Accordingly, any algorithm that runs on this network (including ULTRA) will automatically benefit from operator pruning.

3.4 Extended Scenarios

Free-Floating Bike Sharing. Some bike sharing operators use a *free-floating* (or *dockless*) sharing system without fixed stations, where bikes can be picked up or dropped off at any location within the served region. This can be handled by considering every vertex in the region as a bike sharing station. Unlike with fixed bike sharing stations, this scenario is inherently dynamic: Bikes are not available at every vertex in the region, and it is not known in advance where bikes will be located. Therefore, precomputation techniques such as ULTRA are not applicable. MCR can handle this by checking explicitly whether a bike is available when arriving at a vertex. Operator pruning can also be adapted by running the profile variant of MCR from each boundary vertex of the region, in addition to including all vertices and trips within the region in the hull.

Fixed Pickup Stations with Free-Floating Dropoff. We also consider a hybrid system where pickup is restricted to fixed stations but dropoff is allowed at any location. As with the fully station-based system, we assume that a bike is always available at every station. This makes ULTRA feasible again. Under the reasonable assumption that $\tau_{\text{bike}}(e) \leq \tau_{\text{walk}}(e)$ holds

Table 1 Sizes of the used public transit networks and their bike sharing systems.

Network	Stops	Routes	Trips	Vertices	Edges	Stations	Operators
London	20 595	2 107	125 436	183 025	579 888	823	4
Switzerland	25 426	13 934	369 534	604 167	1 847 140	534	11
Germany	244 055	231 089	2 387 297	6 872 105	21 372 360	2 682	22

for every edge $e \in \mathcal{E}$, it only makes sense to drop off a bike at specific vertices: A bike from operator i may be dropped off at a stop (in order to enter a trip without bicycle transport), a pickup station from a different operator j (in order to switch operators), a boundary vertex of the served region (when leaving the region), or the target vertex. Dropping off the bike at any other vertex would incur unnecessary walking costs. For each such vertex v, the edge (v^i, v^0) is added to the operator-expanded network. Since the target vertex t changes with each query, the edge (t^i, t^0) is only inserted temporarily at query time.

Biking as Additional Trip. In the original MCR publication [6], which considered bike sharing with only one operator, each bike that was used in a journey was counted as an additional trip. Incorporating this into our adapted version of MCR is straightforward. The operator pruning technique can be applied without changes, since its preprocessing only considers journeys that use a single bike for the entire journey. Adapting ULTRA, however, would require fundamental changes because it is based on enumerating all journeys with exactly two trips. If bike usage is counted as an additional trip, two trips are no longer sufficient for finding all relevant transfers.

4 Experiments

All algorithms were implemented in C++17 compiled with GCC version 8.2.1 and optimization flag -O3. All experiments were conducted on a machine with two 8-core Intel Xeon Skylake SP Gold 6144 CPUs clocked at 3.5 GHz, with a turbo frequency of 4.2 GHz, 192 GiB of DDR4-2666 RAM, and 24.75 MiB of L3 cache.

Benchmark Data. We evaluated our algorithms on the transportation network of London, which was also used to evaluate the MCR algorithm [6], as well as the networks of Switzerland and Germany as introduced in [22]. For all three instances, we combined the public transit networks with transfer graphs representing walking and cycling that were extracted from OpenStreetMap[1]. In order to obtain travel times, we assumed an average walking speed of 4.5 km/h and an average cycling speed of 20 km/h. However, the cycling speed was reduced to the speed limit for streets where the speed limit is below 20 km/h. The locations of bike sharing stations were also extracted from OpenStreetMap. We assigned each bike sharing station to an operator based on the information available (e.g., identifying different spellings of the same company). For stations that were not annotated with any information, we chose an operator of other nearby stations at random. Operators with only one bike sharing station were dropped from the dataset, as they are irrelevant for routing. For reproducibility we make the resulting cleaned bike sharing station dataset available[2]. For our experiments we assume a pickup time of 20 seconds and a dropoff time of 10 seconds for all bike sharing stations. An overview of the networks is given in Table 1.

[1] https://download.geofabrik.de/
[2] https://i11www.iti.kit.edu/PublicTransitData/BikeSharing/

Table 2 Overview of all preprocessing steps. We report the sizes of the precomputed data structures as well as the required computation time. Columns labeled with OE contain results for the full operator-expanded network; columns labeled with OE-OP represent the operator-expanded network with operator pruning. The *Total* row corresponds to the preprocessing time of ULTRA-OE(-OP). Entries marked with ? are not reported as their computation would take several weeks. We report single core running times with the exception of two rows marked with (16), where 16 cores were used.

		London		Switzerland		Germany	
	Measured value	OE	OE-OP	OE	OE-OP	OE	OE-OP
Results	Cycling core vertices	–	26 742	–	37 779	–	435 751
	Operator hull vertices	–	13 735	–	19 018	–	351 224
	Expanded stops	102 975	31 216	301 500	36 892	5 613 265	411 980
	Expanded vertices	290 791	197 558	1 019 779	623 228	16 461 221	7 225 923
	Expanded edges	2 081 218	748 938	7 673 596	2 002 615	155 594 242	24 236 935
	ULTRA shortcuts	1 831 779	521 882	3 389 309	435 514	?	7 873 379
Running time [h:m:s]	Walking Core-CH	–	0:06	–	0:28	–	6:36
	Cycling Core-CH	0:06	0:06	0:28	0:28	6:43	6:43
	Operator hulls	–	3:01:21	–	50:20	–	83:38:15
	Operator hulls (16)	–	15:34	–	4:15	–	8:45:22
	Expanded Core-CH	0:08	0:07	0:34	0:29	8:38	7:07
	ULTRA shortcuts (16)	14:14:51	43:31	9:59:43	21:50	?	30:50:13
	Expanded Bucket-CH	0:14	0:09	1:09	0:33	?	17:47
	Total	14:15:19	59:33	10:01:54	28:03	?	40:13:48

4.1 Preprocessing

All algorithms discussed in this work require some form of preprocessing. The most elaborate preprocessing steps are the operator hull computation as well as the ULTRA shortcut computation. In addition to these two steps, several CH computations are required. An overview of the preprocessing steps and their results is given in Table 2.

Regarding the operator hull computation, we observe that the number of vertices contained in the union of all hulls is significantly smaller than the size of the corresponding core graph. This means that some parts of the network will never be visited with a rented bike. The small hulls also have a direct impact on the expanded networks, as their size is also significantly reduced when operator hulls are applied. Using 16 cores, hulls for the small networks can be computed in a few minutes, while Germany requires less than 9 hours. For the networks of London and Switzerland, this corresponds to a speedup factor of 12 compared to a sequential computation. For the Germany network, we only achieve a speedup of 9.5. In order to explain this effect, we measured the average number of instructions executed per CPU cycle. For the sequential hull computation on the Germany network, we recorded a value of 1.1, while it was only 0.9 for the parallel computation. These numbers suggest that the reduced speedup observed for the Germany network is due to an overloaded memory system.

The impact of the operator hulls on the ULTRA shortcuts is also quite strong. On the London and Switzerland networks, operator hulls reduce the preprocessing time by a factor of 14 and 20, and the number of shortcuts by a factor of about 4 and 8, respectively. For the Germany network, ULTRA is only viable with operator hulls. Without operator hulls, only 4.7% of the shortcut computation was finished after one week. We therefore estimate that the complete shortcut computation would take about 21 weeks.

Table 3 Performance overview of all approaches described in this work. All algorithms are evaluated with and without the use of operator pruning (OP). The MR-∞ algorithm is evaluated for both the operator-dependent (OD) and the operator-expanded (OE) model, while ULTRA can only be used with the operator-expanded model. The query results are averages over 10 000 random queries. The Vertices (Routes) column reports the average number of vertices (routes) settled (scanned) by the algorithm.

Network	Algorithm	Preprocessing	Query			
		Time [h:m:s]	Rounds	Vertices	Routes	Time [ms]
London	MR-∞-OD	0:12	9.59	342 361	25 037	112.2
	MR-∞-OD-OP	15:46	8.90	135 765	10 884	51.1
	MR-∞-OE	0:14	9.59	320 286	25 045	119.1
	MR-∞-OE-OP	15:53	8.64	117 188	9 152	34.2
	ULTRA-OE	14:15:19	9.70	78 486	25 922	52.8
	ULTRA-OE-OP	59:33	8.75	23 534	9 532	17.1
Switzerland	MR-∞-OD	0:56	9.55	840 396	171 361	286.8
	MR-∞-OD-OP	5:11	8.49	176 364	54 173	85.0
	MR-∞-OE	1:02	9.55	782 572	171 410	345.0
	MR-∞-OE-OP	5:40	8.35	144 522	43 980	52.8
	ULTRA-OE	10:01:54	9.70	107 627	180 064	117.2
	ULTRA-OE-OP	28:03	8.48	29 394	44 970	21.0
Germany	MR-∞-OD	13:19	11.99	17 421 659	2 888 893	9 830.1
	MR-∞-OD-OP	8:58:41	10.62	2 689 029	706 307	2 183.9
	MR-∞-OE	15:21	11.99	16 120 342	2 889 313	10 599.3
	MR-∞-OE-OP	9:05:48	10.24	2 091 814	679 898	1 322.7
	ULTRA-OE-OP	40:13:48	10.38	301 832	688 525	649.3

4.2 Queries

To evaluate the impact of operator pruning and the differences between the operator-dependent and operator-expanded models, we evaluated all algorithms presented in this work on 10 000 random queries. An overview of the results is given in Table 3.

We use MCR, or more specifically its MR-∞ variant, to compare the operator-dependent and operator-expanded model, as MR-∞ can be used with both models. Without operator pruning, both models perform similarly. This is to be expected, as the operator-dependent algorithm more or less simulates what the standard algorithm does on the operator-expanded model. The number of rounds is exactly the same for both approaches, and the small deviations in the number of settled vertices and scanned rounds can be explained by differences in the respective core graphs and a slightly better target pruning in the dependent model. Still, the operator-dependent model is slightly faster, as it has less memory usage.

Operator pruning improves query times significantly, with the speedup ranging from 2.2 on the operator-dependent London network to 8.0 on the operator-expanded Germany network. Naturally, the speedup is greater on larger networks with more bike sharing operators. Approximately one fewer round is performed on average as a result of reducing the search space. Moreover, the operator-expanded model benefits more strongly from operator pruning than the operator-dependent model, being faster by a factor of 1.5 to 1.7. This is because vertices and trips that are not part of the operator hull are removed entirely from the network instead of being skipped at query time.

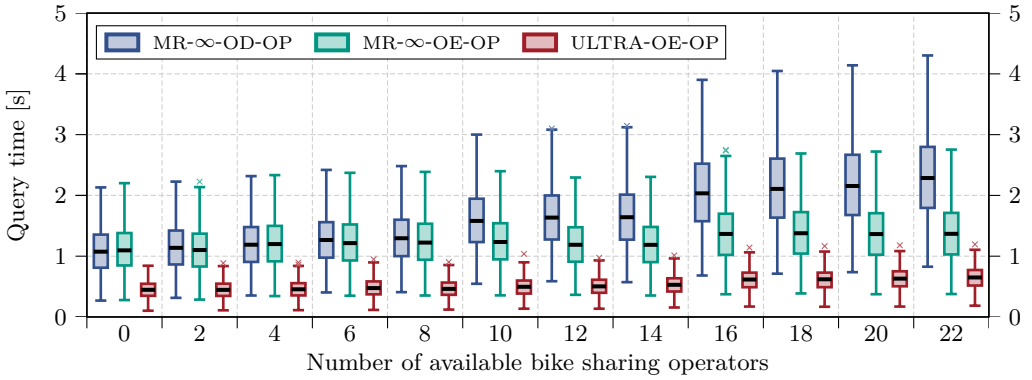


Figure 2 Running time of all query algorithms with operator pruning depending on the number of bike sharing operators available. We used the Germany network without bike sharing and gradually added the bike sharing operators in sets of two (in order to compensate for differences in the number of stations). We evaluated the same 1 000 random queries for each number of bike sharing operators.

Combining ULTRA-RAPTOR with the operator-expanded model reduces query times even further. Compared to the base approach of using MR-∞ on the operator-dependent network, we achieve speedup factors of 6.6, 13.7, and 15.1 for the networks of London, Switzerland, and Germany, respectively. Furthermore, ULTRA-RAPTOR is the first algorithm that enables query times below a second for all networks.

Impact of Additional Bike Sharing Operators. We evaluated how the number of available bike sharing operators influences the query time of the algorithms. For this we used the Germany network, as it is the largest network and has the most bike sharing operators. We started without any bike sharing operators and created partial instances by successively adding operators. To compensate for differences in the number of bike sharing stations, we added operators in pairs of two, pairing large operators with smaller ones. Finally, we evaluated the same set of randomly picked long-range queries for all networks. The results are depicted in Figure 2. We observe that for all algorithms the number of operators is correlated linearly to the query time. The impact on the query time is the strongest for MR-∞-OD-OP, further confirming that the operator-expanded model benefits more from operator pruning than the operator-dependent model.

5 Conclusion

We presented two novel approaches for modeling multi-modal transportation networks with various competing bike sharing operators: the operator-dependent and operator-expanded model. We showed that both models result in similar query performance, with the operator-dependent model being more memory-efficient and the operator-expanded model being compatible with existing query algorithms without modifications. Given its compatibility, we were able to combine the operator-expanded model with ULTRA, a known speedup technique for multi-modal networks, in order to reduce query times. Additionally, we developed a fast preprocessing step called operator pruning, which can be used to accelerate queries in both models. Our experimental evaluation shows that combining operator pruning with ULTRA-RAPTOR enables queries that are more than an order of magnitude faster than the operator-dependent variant of MCR. Beyond that, we showed that using operator pruning also reduces the preprocessing time of ULTRA by more than an order of magnitude.

References

1 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 *European Symposium on Algorithms*, pages 290–301. Springer, 2010.

2 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*, pages 19–80. Springer, 2016.

3 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.

4 Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller-Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder than Expected. In *OASIcs-OpenAccess Series in Informatics*, volume 12. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

5 Dominik Bucher, David Jonietz, and Martin Raubal. A Heuristic for Multi-Modal Route Planning. In *Progress in Location-Based Services 2016*, pages 211–229. Springer, 2017.

6 Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato Werneck. Computing Multimodal Journeys in Practice. In *International Symposium on Experimental Algorithms*, pages 260–271. Springer, 2013.

7 Daniel Delling, Thomas Pajor, and Renato F Werneck. Round-based Public Transit Routing. *Transportation Science*, 49(3):591–604, 2014.

8 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly Simple and Fast Transit Routing. In *International Symposium on Experimental Algorithms*, pages 43–54. Springer, 2013.

9 Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. User-Constrained Multimodal Route Planning. *ACM Journal of Experimental Algorithmics*, 19:3.2:1–3.2:19, April 2015.

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

11 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

12 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.

13 Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos Zaroliagis. Multimodal Dynamic Journey-Planning. *Algorithms*, 12(10):213, 2019.

14 Andrew V Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search meets Graph Theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

15 Jan Hrnčíř and Michal Jakob. Generalised Time-Dependent Graphs for Fully Multimodal Journey Planning. In *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, pages 2138–2145. IEEE, 2013.

16 Dominik Kirchler. *Efficient Routing on Multi-Modal Transportation Networks*. PhD thesis, Ecole Polytechnique X, 2013.

17 Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 36–45. SIAM, 2007.

18 Matthias Müller-Hannemann and Mathias Schnee. Finding all Attractive Train Connections by Multi-Criteria Pareto Search. In *Algorithmic Methods for Railway Optimization*, pages 246–263. Springer, 2007.

19 Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable Information: Models and Algorithms. In *Algorithmic Methods for Railway Optimization*, pages 67–90. Springer, 2007.

20 Stefano Pallottino and Maria Grazia Scutella. Shortest Path Algorithms in Transportation Models: Classical and Innovative Aspects. In *Equilibrium and Advanced Transportation Modelling*, pages 245–281. Springer, 1998.

21 Sabine Storandt. Route Planning for Bicycles – Exact Constrained Shortest Paths made Practical via Contraction Hierarchy. In *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

22 Dorothea Wagner and Tobias Zündorf. Public Transit Routing with Unrestricted Walking. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2017.

23 Sascha Witt. Trip-Based Public Transit Routing. In *Algorithms-ESA 2015*, pages 1025–1036. Springer, 2015.

Efficient Route Planning with Temporary Driving Bans, Road Closures, and Rated Parking Areas

Alexander Kleff
PTV Group, Karlsruhe, Germany
alexander.kleff@ptvgroup.com

Frank Schulz
PTV Group, Karlsruhe, Germany
frank.schulz@ptvgroup.com

Jakob Wagenblatt
Karlsruhe Institute of Technology, Germany
jakob.wagenblatt@student.kit.edu

Tim Zeitz
Karlsruhe Institute of Technology, Germany
tim.zeitz@kit.edu

Abstract

We study the problem of planning routes in road networks when certain streets or areas are closed at certain times. For heavy vehicles, such areas may be very large since many European countries impose temporary driving bans during the night or on weekends. In this setting, feasible routes may require waiting at parking areas, and several feasible routes with different trade-offs between waiting and driving detours around closed areas may exist. We propose a novel model in which driving and waiting are assigned abstract costs, and waiting costs are location-dependent to reflect the different quality of the parking areas. Our goal is to find Pareto-optimal routes with regards to arrival time at the destination and total cost. We investigate the complexity of the model and determine a necessary constraint on the cost parameters such that the problem is solvable in polynomial time. We present a thoroughly engineered implementation and perform experiments on a production-grade real world data set. The experiments show that our implementation can answer realistic queries in around a second or less which makes it feasible for practical application.

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

Keywords and phrases driving bans, realistic road networks, route planning, shortest paths

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.17

Related Version A full version of the paper is available at `https://arxiv.org/abs/2004.09163`.

1 Introduction

Many European countries impose temporary driving bans for heavy vehicles. Driving may be restricted during the night, on weekends, and on public holidays. Such bans may apply to the whole road network of a country or parts of it. When routing a heavy vehicle from a source to a destination, it is crucial to take these temporary driving bans into account. But it is not only about heavy vehicles. Temporary closures of bridges, tunnels, border crossings, mountain pass roads, or certain inner-city areas as well as closures due to roadworks may affect all road users alike. In case of road space rationing in cities, the driving restriction may depend on the license plate number. To sum up, temporary driving restrictions exist in different forms, and the closing and re-opening times of a road segment must be considered in the route planning.

As a consequence of temporary driving restrictions, waiting times may be inevitable and even last for hours. During such waiting hours, the vehicle must be parked properly, and thus a suitable parking area has to be found. The driving time of the detour from and to such a parking area should also be incorporated in the route planning. Unfortunately, the underlying shortest (here: quickest) path problem becomes *NP*-hard if waiting is only allowed at dedicated locations [14]. This is because in this case, the so-called *FIFO* (first in, first out) property is not satisfied, that is, the property that a driver cannot arrive earlier by departing later. Thus, our first research question is how we can consider dedicated waiting locations without making the underlying problem *NP*-hard. It is our aim to obtain a feasible running time even for long-distance routes.

In practice, we often find that small parking areas without any facilities like public toilets or restaurants cause the least detour. So an algorithm that looks for the shortest route, that is, a route with the shortest driving time, would select small parking areas in these cases, provided that waiting is necessary. But the longer the waiting time is, the more vital a secure and pleasant place for waiting becomes. So it may be important for the driver that nearby facilities of the parking area and their quality are somehow taken into account as well. How to do this is our second research question.

In our setting, a single-criterion objective is not practical. A driver may not always be in favor of the shortest route if that means to spend a very long time waiting and to arrive at the destination considerably later than on the quickest route, that is, a route with the earliest arrival at the destination. Conversely, a driver may not always be interested in a quickest route if that route means to take an unjustified long detour around temporarily closed road segments that could be avoided by waiting in a comfortable place. In other words, an early arrival at the destination (and thus low opportunity costs), little driving time (and thus low fuel costs), and pleasant waiting conditions (and thus high driver satisfaction) are competing criteria. Solutions can differ significantly with regards to these criteria. How to deal with this and find reasonable routes is the third research question.

In this paper, we answer these questions as follows:

1. We present a model in which waiting is allowed at any vertex and any edge at any time in the road graph but waiting on edges and waiting on those vertices that do not correspond to parking areas is penalized. This is done by assigning a cost to time spent waiting there. Since driving comes at a price, too, we also assign a cost per time unit spent driving. As we will show, we can find a route with least costs in polynomial time if both cost parameters are set to the same value.
2. We assume that the nearby facilities of a parking area and their quality can be expressed by some single rating number. To take account of this, we assign a waiting cost to every corresponding vertex as well. This cost is lower than the cost of waiting anywhere else in the road graph, and it is even lower the higher the rating of the parking area is.
3. We return routes that are Pareto-optimal with regards to arrival time at the destination on the one hand and total costs on the other. Despite the potentially larger output, our algorithm still runs in polynomial time under the same condition as before.

As our experiments reveal, many queries within Europe are answered within milliseconds. Except some pathological cases, even more complex queries with four or more Pareto-optimal solutions are solved in less than a second.

Related Work. Many route planning problems are modeled as shortest path problems. To this day, the theoretically fastest known algorithm to find shortest paths on graphs with static non-negative edge weights is the algorithm of Dijkstra [9]. However, for many practical

applications, it is not fast enough. One approach to speed up the computation is to reduce the search space of Dijkstra's algorithm by guiding the search towards the destination by means of estimates of the remaining distance to the destination. It is known as the A* algorithm [12]. Since the advent of routing services, a lot of research has been done on efficient algorithms for routing in road networks. Routing services have to answer many queries on the same network. This can be used to speed up shortest path queries through precomputed auxiliary data. Many approaches exploit certain characteristics of road networks, for example the hierarchical structure (freeways are more important than rural roads). For an extensive overview, we refer to [1]. One particularly popular speed-up technique are Contraction Hierarchies [11]. During preprocessing, additional shortcut edges are inserted into the graph, which skip over unimportant vertices. This preprocessing typically takes a few minutes. Then, shortest path queries can be answered in less than a millisecond.

A natural approach to handle driving restrictions is to model them as time-dependent travel times [10]. For the blocked time, the travel time of the edge can be set to infinity. Time-dependent route planning has also received some attention and effective speed-up techniques are known [2, 3, 6, 5, 13].

Variants of our problem have been studied in the literature. In [7] a related problem is discussed where nodes (not edges) have time windows and waiting is associated with a cost. In [15] an overview is given over different exact approaches to solving shortest path problems with resource constraints. Time windows on nodes are a specific kind of constraint in this framework. More specialized models for routing applications have been proposed. The authors of [17] study the problem of planning a single break, considering driving restrictions and provisions on driver breaks. They aim to find only the route with the earliest arrival.

Contribution. We present a novel model that helps answer our three research questions in the context of temporary driving restrictions and dedicated waiting locations. To the best of our knowledge, this is the first unifying approach that gives answers to all three research questions. Our theoretical analysis reveals that our model can be solved to optimality in polynomial time, given certain restrictions on the parameterization. The experimental evaluation of our implementation demonstrates a practical running time.

Outline. In Section 2, we give a formal definition of the routing problem at hand. In Section 3, we present an exact algorithm for this problem. In Section 4, we analyze the complexity of the problem and show that our algorithm runs in polynomial time if the costs for driving are the same as for waiting anywhere else than at a dedicated waiting location. In Section 5, we describe techniques to speed-up the computation. In Section 6, we present the main results of our experiments. Finally, we conclude in Section 7.

2 Problem

A problem instance comprises a *road graph with ban intervals on edges, driving costs and location-dependent waiting costs* (or *road graph with ban intervals and costs* for short) as well as a set of *queries*. The road graph is characterized by the following attributes:

- A set V of n vertices and a set E of m directed edges.
- A mapping Φ that maps each edge $e \in E$ to a sequence of disjoint time intervals, where the edge is considered to be *closed* during each interval. Precisely, for any *ban interval* $[t^{closed}, t^{open}) \in \Phi(e)$ of an edge e, t^{open} denotes the first point in time after t^{closed} where the edge is open again. Here and in the following, all points in time are integers and the

length of an interval is denoted by $|[t^{closed}, t^{open})|$ and equals $t^{open} - t^{closed} > 0$. During such a time span, a vehicle on the corresponding road segment must not move. We denote the total number of ban intervals as b.

- A mapping $\delta : E \to \mathbb{N}$ that maps each edge $e := (u, v) \in E$ to the time $\delta(e)$ that it takes to drive from u to v, provided the edge is open.
- A mapping ρ that maps each vertex to a rating in $\{0, 1, \ldots, r\}$ with $r \leq n$. Rating 0 means *unrated*, that is, it is assumed that it is highly difficult, dangerous, and not allowed to park the vehicle there. In contrast to an unrated location, we call a vertex v with $\rho(v) > 0$ a *parking location*.
- A parameter set of abstract costs, consisting of $d \in \mathbb{Q}_{\geq 0}$, the cost per unit of driving time, and $w_i \in \mathbb{Q}_{\geq 0}$ for all i from 0 to r, the cost per time unit of waiting on a vertex with rating i. Edges are always unrated so waiting there costs w_0 per time unit. W.l.o.g. $w_i < w_{i-1}$ holds for all i between 1 and r, that is, we assume that waiting on vertices with a higher rating costs less than waiting on those with a lower rating.

A *u-v-route* is a triple (R, A, D) of three sequences of the same length $\ell := |R| = |A| = |D|$. Here, R is the sequence of vertices along the route. It describes a (not necessarily simple) *path* in the graph that starts at u and ends in v, that is, $e_i := (R[i], R[i+1]) \in E$ for all $1 \leq i < \ell$ and $R[1] = u$ and $R[\ell] = v$. The other two sequences A and D denote the *arrival times* and the *departure times* from the respective vertices, where $A[i] \leq D[i]$ for all $1 \leq i \leq \ell$ and $A[i+1] - D[i] \geq \delta(e_i)$ for all $1 \leq i < \ell$ holds.

A query comprises a *source* $s \in V$ and a *destination* $z \in V$ as well as a *planning horizon* H. The latter is defined as the time interval between an *earliest departure time* t^{min} from s and a *latest arrival time* t^{max} at z. Waiting costs arise as soon as the planning horizon opens. For a given query, we look for *feasible* s-z-routes. A route is feasible with respect to the planning horizon if $A[1] = t^{min}$ and $D[\ell] \leq t^{max}$. In addition, ban intervals must be taken account of. Let $T_i := [D[i], A[i+1])$ be the time interval in which the edge $e_i := (R[i], R[i+1])$ of the route's path is traversed. A route is feasible with respect to the ban intervals if $\sum_{I \in \Phi(e_i)} |T_i \cap I| \leq |T_i| - \delta(e_i)$ for all $1 \leq i < \ell$. Here, $\sum_{I \in \Phi(e_i)} |T_i \cap I|$ is the time during which the edge between $R[i]$ and $R[i+1]$ is closed while the edge is being traversed.

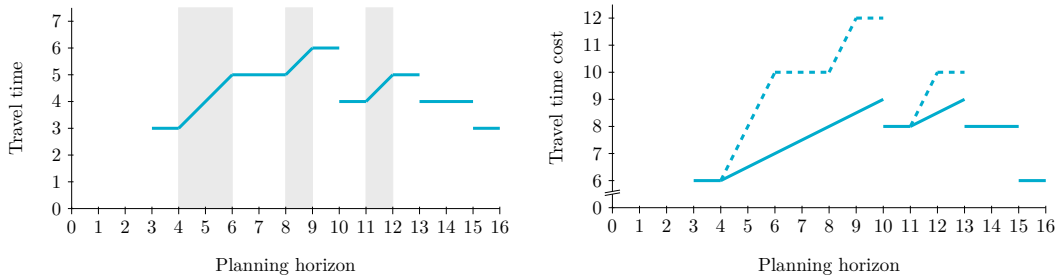
Let *travel time* include driving time and waiting time. The *travel time costs* of a route are the sum of the waiting time costs and the driving time costs. So given a route of length ℓ, the travel time costs are

$$\sum_{i=1}^{\ell} w_{\rho(R[i])} \cdot (D[i] - A[i]) + \sum_{i=1}^{\ell-1} w_0 \cdot (A[i+1] - D[i] - \delta(e_i)) + d \cdot \delta(e_i),$$

where we use $e_i := (R[i], R[i+1])$. We say an s-z-route is *Pareto-optimal* (or simply *optimal*) if it is feasible and if its travel time costs are less or its arrival time at z is earlier or equality holds in both cases compared to any other feasible s-z-route. For a query, the objective is to find a maximal set of (Pareto-)optimal s-z-routes such that no two routes in the set have both the same arrival time at z and the same travel time costs.

3 Algorithm

The algorithm maintains a priority queue. Each entry of the queue consists of a vertex and a point in time within the planning horizon as key. We say a vertex is *visited* at a certain point in time whenever we remove the top entry from the queue, that is, an entry with the earliest time among the entries in the queue. At every vertex $v \in V$, we store a time-dependent



(a) Travel time function $\mathcal{T}_e$ of an edge e with ban intervals (grey) and a driving time $\delta(e)$ of 3. The latest departure to be at v at time t is $t - \mathcal{T}_e(t)$.

(b) Cost profile of vertex v after linking, that is, after considering travel time (dashed) and waiting time at v (solid).

Figure 1 Computing the cost profile of a vertex v. Let v be adjacent to the source s via an edge $e := (s, v)$ with three ban intervals and a driving time $\delta(e)$ of 3. The corresponding travel time function is given in Figure 1a. It is infinite between $0 = t^{min}$ and $3 = \delta(e)$. In Figure 1b, we see the cost profile $\mathcal{C}_v$ after considering the travel time along the edge (dashed) and after considering waiting at v (solid). Here, the assumed cost parameters are $w_{\rho(s)} = 0$, $w_{\rho(v)} = 0.5$, and $d = w_0 = 2$, where $w_{\rho(s)} = 0$ implies that the cost profile $\mathcal{C}_s$ at the source is 0 over the whole planning horizon.

function $\mathcal{C}_v : H \to \mathbb{Q}_{\geq 0} \cup \{\infty\}$. It maps a point in time t within the planning horizon H to an upper bound on the minimum travel time cost over all s-v-routes that end in v at time t. We call this function *cost profile* of v or, more general, *label* of v. The algorithm works in a *label correcting* manner in the sense that a vertex may be visited multiple times, albeit at different times within the planning horizon.

Before we describe the phases of the algorithm in greater detail, we introduce an auxiliary time-dependent function $\mathcal{T}_e$ for every edge $e \in E$. It maps a time t at the head v of an edge $e := (u, v)$ to the *shortest travel time* that it takes to traverse the edge from u to v completely and be at v at time t, possibly including waiting time. That is, for a time t at v, $\mathcal{T}_e(t)$ is the minimum period p such that $p - \sum_{I \in \Phi(e)} |[t-p, t) \cap I| \geq \delta(e)$ holds if such a p exists, and ∞ otherwise. In other words, $t - \mathcal{T}_e(t)$ is the latest departure time from u in order not to arrive at v later than at time t. An example is given in Figure 1a.

In the initialization phase of the algorithm, we set $\mathcal{C}_s(t) := w_{\rho(s)} \cdot (t - t^{min})$ for all $t \in H$. For every other $v \in V \setminus \{s\}$, we set $\mathcal{C}_v(t) := \infty$ for all $t \in H$. Furthermore, we insert the source s with key t^{min} into the priority queue.

As long as the queue is not empty, we are in the main loop of the algorithm. In every iteration of the main loop, we remove the top entry from the queue. Let us suppose we visit a vertex u at time $t^{visit} \geq t^{min}$. Then, we check for every edge $e := (u, v)$ going out of u whether we can improve the cost profile $\mathcal{C}_v$ of v. We do so in three steps. In the first step, we consider the travel time along the edge and set

$$\mathcal{C}'_v(t) := \mathcal{C}_u(t - \mathcal{T}_e(t)) + d \cdot \delta(e) + w_0 \cdot (\mathcal{T}_e(t) - \delta(e)) \tag{1}$$

for all t with $t^{visit} + \mathcal{T}_e(t) \leq t \leq t^{max}$. For all other $t \in H$ we set $\mathcal{C}'_v(t) := \infty$. In the second step, we consider waiting at v at cost $w_{\rho(v)}$ per time unit and set

$$\mathcal{C}'_v(t) := \min\{\mathcal{C}'_v(t') + w_{\rho(v)} \cdot (t - t') \mid t^{min} \leq t' \leq t\} \tag{2}$$

for all $t \in H$. An example of the first two steps is illustrated in Figure 1b. Finally, in the third step, we compare $\mathcal{C}'_v$ and $\mathcal{C}_v$. Let t^* be the earliest point in time such that $\mathcal{C}'_v(t^*)$ is less than $\mathcal{C}_v(t^*)$ if such a time t^* exists. Only if it exists, we set $\mathcal{C}_v(t)$ to the minimum of $\mathcal{C}_v(t)$ and $\mathcal{C}'_v(t)$ for all $t^* \leq t \leq t^{max}$. Furthermore, we insert vertex v with key t^* into the priority queue or decrease the key if v is already contained.

When the priority queue is empty, we enter the finalization phase of the algorithm. We say a time-cost-pair $(t, \mathcal{C}_z(t))$ with $t \in H$ and $\mathcal{C}_z(t) < \infty$ is Pareto-optimal if there is no time t' with $t^{min} \leq t' < t$ and $\mathcal{C}_z(t') \leq \mathcal{C}_z(t)$. In the finalization phase, we extract an s-z-route for every Pareto-optimal time-cost-pair. So let such a time-cost-pair $(t, \mathcal{C}_z(t))$ be given. In order to find a corresponding route (R, A, D), we initially push z and t and t to the front of the (empty) sequences R and A and D, respectively. The following is done iteratively until we reach the source, that is, $R[1] = s$ holds. First, we look for an incoming edge $e := (u, R[1])$ of $R[1]$ and a departure time t from u with

$$\mathcal{C}_u(t) + d \cdot \delta(e) + w_0 \cdot (\mathcal{T}_e(A[1]) - \delta(e)) = \mathcal{C}_{R[1]}(A[1])$$

which must exist. We push u and t to the front of R and D, respectively. Then, we push the earliest time $t \leq D[1]$ such that

$$\mathcal{C}_{R[1]}(t) + w_{\rho(R[1])} \cdot (D[1] - t) = \mathcal{C}_{R[1]}(D[1])$$

holds to the front of the arrival time sequence A, and continue with the next iteration. This concludes the description of the finalization phase and thus the whole algorithm.

For the correctness of the algorithm it is important that the upper bound $\mathcal{C}_v(t)$ on the minimum travel time cost is tight for all $t \leq t^{visit}$ and all $v \in V$ whenever we visit a vertex at time t^{visit}. After the main loop, it is tight for every $t \in H$ and all $v \in V$, especially for z. This can be proven by induction on the time of visit. The time of visiting a vertex increases monotonically because whenever a vertex is inserted into the queue or its key is decreased, the (new) value of that key can only be later than the current time of visit.

4 Analysis

In this section, we first show the intractability of the general problem. Then, we restrict the problem by requiring the driving cost d to be equal to the unrated waiting cost w_0, and prove that our algorithm solves the restricted problem in polynomial time.

Intractability of the General Problem. The first two theorems show the intractability of the general problem if $d \neq w_0$. Parking locations are not used in the proofs, so already the simplified problem without parking locations is intractable if $d \neq w_0$.

▶ **Theorem 1.** *If $d < w_0$ then it is NP-complete to decide whether there is a feasible route with travel time costs less than or equal to a given threshold k.*

We prove this theorem in the full version of this paper by reduction from PARTITION.

▶ **Theorem 2.** *If $d > w_0$ then the number of Pareto-optimal routes can be exponential in the number of vertices.*

Given a number of vertices a graph with ban intervals can be constructed that has exponentially many routes which are all Pareto-optimal. The construction and the proof that all those routes are Pareto-optimal can be found in the full version of this paper.

Tractable Problem Variant. For the remaining analysis we assume $d = w_0$. In the setting without parking locations, there is only one optimal solution, since the quickest solution has also the least cost. Hence, this setting is a single-criterion shortest path problem with time-dependent edge weights that fulfill the *FIFO* property and can be solved in polynomial

time with a time-dependent variant of Dijkstra's algorithm [10], and also our algorithm reduces to such a time-dependent Dijkstra variant and has polynomial running time. Now we turn to the setting $d = w_0$ with parking locations and show that it is still tractable.

Cost profiles are piecewise linear functions. An important aspect of our polynomial time proof is to count the non-differentiable points of the profiles. The running time of each profile operation of our algorithm is linear in the number of non-differentiable points of the involved profiles. These points are either *convex*, *concave*, or *discontinuous*, meaning an environment around such a point exists in which the profile is convex or concave or discontinuous, respectively. In a discontinuous point, a profile is always jumping down.

The non-differentiable points in the cost profiles are induced by the travel time functions. In our example of Figure 1a, the convex points are $\{4, 8, 11\}$, the concave points are $\{6, 9, 12\}$, and the discontinuous points are $\{10, 13, 15\}$. For a travel time function $\mathcal{T}_e$ of an edge e, we can assign a convex point t to the beginning of a ban interval in t, a concave point t to the end of a ban interval in t, and a discontinuous point t to the end of a ban interval in $t - \mathcal{T}_e(t)$. From this initial assignment, we can derive a ban interval assignment of the convex or discontinuous points of cost profiles. We omit to count the number of concave points of a cost profile because every gradient of a piece must be in $\{w_0, \dots, w_r\}$, so the number of consecutive concave points in a cost profile is limited by r.

Initially, a profile $\mathcal{C}_v$ of a vertex v has no convex or discontinuous points. Such points may be introduced in the third step of an iteration of the algorithm when the auxiliary profile $\mathcal{C}'_v$ is merged into $\mathcal{C}_v$. In the second step of an iteration, no new convex or discontinuous points can arise in $\mathcal{C}'_v$, so all such points must be created in $\mathcal{C}'_v$ in the first step. Since $d = w_0$, $\mathcal{C}'_v(t)$ is set to $\mathcal{C}_u(t - \mathcal{T}_e(t)) + d \cdot \mathcal{T}_e(t)$ (compare Equation (1)) for some edge $e = (u, v)$ in this step. If t_v is a convex or discontinuous point of $\mathcal{C}'_v$, then $\mathcal{T}_e$ must be convex or discontinuous in the same point in time, or $\mathcal{C}_u$ must be convex or discontinuous in $t_u := t_v - \mathcal{T}_e(t_v)$. In the former case, t_v inherits the assignment of the same point in time in $\mathcal{T}_e$, whereas in the latter case, t_v inherits the assignment of t_u in $\mathcal{C}_u$. Since the cost profiles change during the algorithm, we do not only assign a ban interval to every convex or discontinuous point but also an iteration. Again, in the former case, t_v is assigned the current iteration, whereas in the latter case, t_v inherits the iteration assignment of t_u in $\mathcal{C}_u$.

▶ **Lemma 3.** *If $d = w_0$ then a cost profile after iteration i has at most ib convex and at most ib discontinuous points.*

Proof. In the following, we denote the state of the profile C_v after iteration i by C_v^i. Let t_v be a convex or discontinuous point of $\mathcal{C}_v^i$ that is assigned both to an iteration k and to a ban interval of some edge with head x. We can follow the inheritance relation until we finally reach a convex or discontinuous point t_x in $\mathcal{C}_x^k$. By induction, we have $\mathcal{C}_v^i(t_v) = \mathcal{C}_x^k(t_x) + d \cdot (t_v - t_x)$. Now suppose there are two convex or two discontinuous points $t_v^1 < t_v^2$ in the profile $\mathcal{C}_v^i$ that are assigned to the same ban interval and the same iteration k, so they can be traced back to the same point t_x in $\mathcal{C}_x^k$. Then the previous observation implies that $\mathcal{C}_v^i(t_v^2) - \mathcal{C}_v^i(t_v^1) = d \cdot (t_v^2 - t_v^1)$ holds, that is, the profile $\mathcal{C}_v^i$ must contain a piece with gradient d that contains both t_v^1 and t_v^2. But then t_v^2 can neither be convex nor discontinuous. Hence, two convex or two discontinuous points must differ in their assigned ban interval or their assigned iteration and there can only be ib discontinuous and convex points, respectively. ◀

▶ **Lemma 4.** *If $d = w_0$ then the total number of iterations is at most $2n(b(r + 1) + 1)$.*

As in the proof of Lemma 3 we use the ban interval assignment of convex and discontinuous points. Every visit of a vertex can either be assigned to the start or end of a ban interval, or it can be assigned to a concave point of the final cost profile of the vertex. The detailed proof is omitted due to space limitations and can be found in the full version of this paper.

▶ **Theorem 5.** *If $d = w_0$ then the running time of the algorithm is polynomial.*

Proof. From Lemma 3 with the bound from Lemma 4 it follows that the number of pieces of any profile that is constructed during the algorithm is polynomial.

We now estimate the overall running time of our algorithm: Lemma 4 states that the total number of iterations is polynomial. In every iteration of the algorithm one vertex is considered and for its outgoing edges the profiles are updated with a running time linear in the number of pieces of the profiles. The adjacent vertices are inserted into the priority queue or their keys are decreased. Since the size of the priority queue is at most the total number of vertices also the running time of the priority queue operations is polynomial. ◀

5 Implementation

The past decade has seen a lot of research effort on the engineering of efficient route planning algorithms. This section describes the speed-up techniques we employ in our implementation and some implementation details.

We store cost profiles as a sorted list of pieces. Each piece is represented as a triple: a point in time from which this piece is valid, the costs it takes to reach the vertex at the beginning of the piece and the incline of the piece. For each piece we also store a parent vertex. This allows us to efficiently reconstruct routes by traversing the parent pointers.

We employ A* to guide the search toward the destination vertex. The queue is ordered by the original key plus an estimate of the remaining distance (here: driving time) to the destination. The estimate for vertex u is denoted by $\pi_z(u)$. We use the exact shortest driving time to z without driving restrictions as the potential. This is the best possible potential in our case. We efficiently extract these exact distances from a Contraction Hierarchy [11], as described in [16]. Since our algorithm has to run until the queue is empty, we can not immediately terminate when we reach the destination. However, we get a tentative cost profile at the destination. This allows for effective pruning. Additionally, we do not need to insert a vertex u into the queue when $t^{visit} + \pi_z(u) > t^{max}$ holds, that is, we cannot reach the destination from u within the planning horizon.

We employ pruning to avoid linking and merging when possible using the following rules:

- Consider a vertex u that is visited at t^{visit}. Before relaxing any outgoing edges, we first check if u can actually contribute to any optimal route to z. If $\mathcal{C}_u(t) + \pi_z(u) \cdot d > \mathcal{C}_z(t + \pi_z(u))$ for all t with $t^{visit} \leq t < t^{max}$, u can not contribute to an optimal route to z and can thus be skipped.
- Let $\alpha(u) := \min\{t \mid \mathcal{C}_u(t) < \infty\}$ be the first point in time such that u can be reached with finite costs and ∞ if no such point exists. For each vertex u, we maintain a lower bound $\beta(u) := \min_t\{\mathcal{C}_u(t)\}$ and an upper bound $\gamma(u) := \max_{t>\alpha(u)}\{\mathcal{C}_u(t)\}$ or ∞, if there are no finite costs. They can be updated efficiently during the merge operation. An edge (u, v) only needs to be relaxed if $\beta(u) + \delta(u, v) \cdot d \leq \gamma(v)$ or $\alpha(u) + \delta(u, v) < \alpha(v)$.
- When all of the pieces of the cost profile of a vertex u share the same parent vertex v and $\rho(u) = 0$, the edge (u, v) back to the parent does not need to be relaxed as loops can never be part of an optimal route unless they include waiting at a parking location.

Table 1 Rating and default waiting cost by capacity of parking locations. The driving cost is the same as the cost for waiting at unrated vertices.

Capacity of parking locations	≥ 80	≥ 40	≥ 15	≥ 5	≥ 1	-
Rating	5	4	3	2	1	0
Default waiting costs	3	4	5	6	7	14
Number of parking locations	448	997	2 664	5 418	5 748	21.9 M

6 Experimental Evaluation

Our algorithm is implemented in C++14 and compiled with Visual C++. For the CH-potentials, we build upon the Contraction Hierarchy implementation of RoutingKit[1] [8]. All experiments were conducted on a Windows 10 Pro machine with an Intel i7-7600 CPU with a base frequency of 3.4 GHz and 32 GB of DDR4 RAM. The implementation is single-threaded.

Our experimental setup is taken from [4]. We perform experiments on a road network used in production by PTV[2]. The network is adapted from data by TomTom[3]. It covers Austria, France, Germany, Italy, Liechtenstein, Luxembourg, and Switzerland. It has 21.9 million vertices and 47.6 million edges. We use travel times, driving bans, and road closures for a truck with a gross combined weight of 40 tons. Driving bans were derived from the current legislation of the respective countries. This includes Sunday driving bans in all countries, a late Saturday driving ban in Austria and night driving bans in Austria, Liechtenstein and Switzerland. Additionally, there is a Saturday driving ban in Italy during the summer holidays. The dataset also includes several local road closures in city centers.

Parking locations were taken from data by Truck Parking Europe[4]. There is a total of 15 317 vertices classified as parking locations in our data set. The dataset also contains the capacity of each parking location. We assign each parking location a rating between 1 and 5 depending on its capacity. Table 1 shows the number of parking locations for each rating and our default waiting costs. We also evaluate different parameterizations. The waiting costs are calculated such that for an hour of waiting a detour of up to four minutes will be taken to get to a parking location rated better by one. For waiting at the source vertex of a query, we assign zero waiting costs regardless of the rating.

We generate two sets of source-destination pairs and combine them with different planning horizons. The first set is used to evaluate the practicality of our model. It is designed to make the algorithm cope with the night driving ban in Austria and Switzerland. We select 100 pairs of vertices. One vertex is randomly selected from the area around southern Germany. The other vertex is selected from the area around northern Italy. See the full version of this paper for coordinates and a visualization. We store each pair in both directions. Hence, we have 200 vertex pairs in this set. The planning horizon starts at Monday 2018/7/2, 18:00 with length one day (query set A1) and two days (A2). Figure 2 depicts an example from A1.

The second set is generated by selecting 100 source vertices uniformly at random. From each source vertex, we run Dijkstra's algorithm without a specific target ignoring any driving restrictions. Dijkstra's algorithm explores the graph by traversing vertices in increasing distance of the source vertex. We use the order in which vertices are settled to select

[1] https://github.com/RoutingKit/RoutingKit
[2] https://ptvgroup.com
[3] https://tomtom.com
[4] https://truckparkingeurope.com

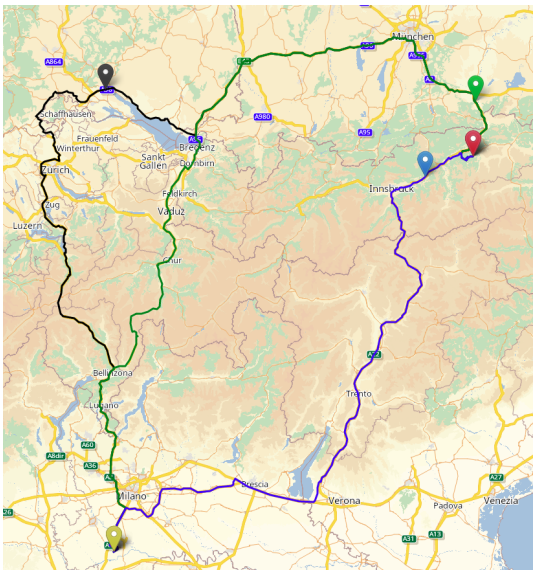


Figure 2 Optimal paths of an example query from northwestern Austria to northern Italy, slightly south of Milano. The source is indicated by a red, the destination by a yellow marker. The other markers indicate the parking locations along the respective routes. The blue route in the east has the shortest driving time, around 10.5 hours, but the latest arrival. It schedules a waiting time of seven hours during the night driving ban at a parking location of rating 4 and afterwards takes the fastest route to the destination. The green route in the middle arrives an hour earlier at the destination but the driving time is over two hours longer. This route includes three hours of waiting at a parking location of rating 5. The black route in the west takes 16 hours to drive, includes only a few minutes of waiting and arrives six minutes before the green one.

destination vertices with different distances from the source. Every 2^ith settled vertex with $i \in [12, 24]$ is stored. We denote i as the *rank* of the query. This results in 1 300 source-destination pairs. We combine these vertex pairs with four planning horizons: starting at Friday 2018/7/6, 06:00 for one day (denoted as query set B1), for two days (B2) and starting later that day at 18:00 for one day (B3) and for two days (B4).

We first investigate whether allowing waiting everywhere (albeit penalized) may lead to unwanted results in practice. On the one hand, routes with many stops are impractical. Our experiments indicate that this is not the case: Accross all routes for A1, there is at most one additional stop scheduled (0.2 on average). On the other hand, let us call a route *precarious* if waiting is scheduled at an unrated location (other than the source vertex). For 187 of the 200 queries of A1, there is no precarious route in the Pareto set. For the other 13 queries, the Pareto set always contains more than one route, and it is always only the quickest route in the Pareto set that is precarious. So filtering out such routes in a postprocessing step does not make a query infeasible. On average, the second quickest route in the Pareto set arrives 422 s later than the quickest but precarious route (minimum 38 s, maximum 877 s).

We also evaluate the influence of different waiting cost parameterizations on the performance and the results of our algorithm. Table 2 depicts the results. We observe that the parametrization has only limited influence on the results of the algorithm. The average number of optimal routes and the arrival time deviation change only very little even between the two most extreme configurations. Since waiting at the source vertex costs nothing, the majority of the waiting in all configurations is scheduled there. When waiting at parking

Table 2 Query statistics for different waiting cost parameters for query set A1. The first six columns show the waiting cost parameters. Waiting costs at the source are always set to zero. The waiting time columns depict the share of the time spent waiting at vertices with the respective rating summed up over all routes. The routes column gives the average number of optimal routes per query. The arrival time deviation column contains the average of the difference between earliest and latest arrival time among all optimal routes for all queries. Running times are also averaged.

						Waiting time by rating [%]							Optimal Routes	Arrival time deviation	Running time
w_5	w_4	w_3	w_2	w_1	$w_0 = d$	s	5	4	3	2	1	0	[#]	[h:mm]	[ms]
1	10	50	100	1000	10000	59.4	2.5	5.8	23.3	3.1	2.8	3.1	3.02	2:21	364.1
1	2	4	8	16	128	62.2	3.5	6.5	19.8	2.1	2.8	3.1	3.02	2:20	412.3
1	2	4	8	16	32	70.8	6.0	5.1	12.1	1.0	1.9	3.1	2.96	2:20	435.4
3	4	5	6	7	14	79.3	6.2	3.1	4.6	1.5	2.0	3.3	2.86	2:17	529.4
16	24	28	30	31	32	85.2	4.6	1.1	3.3	1.1	1.1	3.6	2.71	2:14	742.2

locations is much cheaper than driving, less waiting time will be scheduled at the source and more waiting at parking locations. Also, clear differences between the costs lead to a better running time, because cost profiles become less complex.

We next investigate the algorithm's performance for each of the different query sets. We report the same numbers limited to non-trivial queries. A query is denoted as *trivial* if there is exactly one optimal route which is also optimal when ignoring all driving restrictions. Table 3 depicts the results. Clearly, the query set has a strong influence on the running time of the algorithm. Average running times range from ten milliseconds to one second when looking at all queries. However, median query times are significantly smaller. The reason for this is that our algorithm can answer trivial queries in a few milliseconds or less. Due to the perfect potentials, the algorithm only traverses the optimal path. Once the destination is reached, because of the target pruning, all other vertices in the queue are skipped and the algorithm terminates. Excluding trivial queries, we get a clearer picture of the algorithm's performance when solving the harder part of the problem.

For the query sets B1 and B2, only 4% to 5% of the queries have to deal with driving restrictions. This is mostly due to closures for individual roads in certain cities and not country-wide driving bans. When the planning horizon begins later at 18:00 (B3 and B4), we get around twice as many non-trivial queries. These are primarily caused by the night driving bans in Austria and Switzerland. Road closures and country-wide driving bans lead to different optimal routes. When there is a road closure on the shortest path ignoring any driving restrictions, we often have two optimal routes. One which takes a (small) detour around the closure, and one waiting at the source until the closed road opens and then taking that slightly shorter path. Thus, we have two routes with very similar driving times but (often vastly) diverging arrival times. When dealing with night driving bans, we get more optimal results with different trade-offs as in the example of Figure 2.

Increasing the length of the planning horizon to two days leads to more non-trivial queries, more optimal routes per query, and a greater deviation in arrival time. The reason are routes with a travel time longer than 24 hours which were not valid for the shorter planning horizon.

Even when we restrict ourselves to queries with non-trivial results, running times still vary depending on the query set. Average and median deviate not as strong as when considering all queries, but the distribution of running times is still skewed by a few long running queries,

Table 3 Query statistics for all six query sets. First, for all queries. Second, only for non-trivial queries. A query is denoted as trivial if there is exactly one optimal route which is also optimal when ignoring all driving restrictions. All numbers are averages unless reported otherwise. The arrival time deviation column contains the average of the difference between earliest and latest arrival time among all optimal routes for all queries. The routes column contains the number of optimal routes.

	Set	Planning horizon	Query share [%]	Optimal Routes [#]	Arrival time deviation [h:mm]	Running time Avg. [ms]	Running time Median [ms]
	A1	Mon. 18:00, 1 day	100.0	2.86	2:17	529.4	266.3
	A2	Mon. 18:00, 2 days	100.0	3.54	3:19	648.1	405.6
	B1	Fri. 06:00, 1 day	100.0	1.04	0:10	10.0	0.6
	B2	Fri. 06:00, 2 days	100.0	1.08	0:16	79.5	0.7
	B3	Fri. 18:00, 1 day	100.0	1.13	0:08	205.8	0.6
	B4	Fri. 18:00, 2 days	100.0	1.32	0:20	1 028.1	0.7
Only non-trivial	A1	Mon. 18:00, 1 day	67.5	3.82	3:13	764.1	560.6
Only non-trivial	A2	Mon. 18:00, 2 days	72.0	4.53	4:37	899.2	655.0
Only non-trivial	B1	Fri. 06:00, 1 day	4.1	2.19	4:10	42.5	6.6
Only non-trivial	B2	Fri. 06:00, 2 days	4.8	2.76	5:43	1 105.6	35.8
Only non-trivial	B3	Fri. 18:00, 1 day	9.2	2.73	1:25	1 359.0	475.2
Only non-trivial	B4	Fri. 18:00, 2 days	11.6	3.79	2:51	5 819.4	1 947.2

especially on set B4. The reason for this is that the running time heavily depends on the types and lengths of driving restrictions in the search space. The Saturday driving ban in Italy causes heavy outliers in B4 (but also B2 and B3), when the destination lies in an area blocked for most of the planning horizon. This causes the algorithm to explore large parts of the graph, until the driving ban is over. The worst of these queries took 49 seconds to answer. Nevertheless, when looking at query sets A1 and A2, we clearly see that the algorithm can answer queries affected by country-wide night driving bans in less than a second.

7 Conclusion

We have introduced a variant of the shortest path problem where driving on edges may be forbidden at times, both driving and waiting entail costs, and the cost for waiting depends on the rating of the respective location. The objective is to find a Pareto set of both quickest paths and minimum cost paths in a road graph. We have presented an exact algorithm for this problem and shown that it runs in polynomial time if the cost for driving is the same as for waiting in an unrated location. With this algorithm, we can solve routing problems that arise in practice in the context of temporary driving bans for trucks as well as temporary closures of roads or even larger parts of the road network.

Our experiments demonstrate that our implementation can answer queries with realistic driving restrictions in less than a second on average. There are a few slow outlier queries when the destination vertex lies in a blocked area. A promising angle to improve this could be to study bidirectional variants of our algorithm. We exploit Contraction Hierarchies to efficiently obtain good A* potentials. The algorithm can also be used in a dynamic (or live or online) scenario when combined with Customizable Contraction Hierarchies [8]. A natural extension of our problem at hand is to consider time-dependent driving times or rules for truck drivers that enforce a break after a certain accumulated driving time.

References

1 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. URL: http://www.springer.com/gp/book/9783319494869.

2 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.

3 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. URL: http://link.springer.com/chapter/10.1007/978-3-319-38851-9_3.

4 Christian Bräuer. Route Planning with Temporary Road Closures. Master's thesis, Karlsruhe Institute of Technology, 2018.

5 Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, 60(1):60–94, May 2011. doi:10.1007/s00453-009-9341-0.

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

7 Guy Desaulniers and Daniel Villeneuve. The shortest path problem with time windows and linear waiting costs. *Transportation Science*, 34(3):312–319, 2000.

8 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.

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

10 Stuart E. Dreyfus. An Appraisal of Some Shortest-Path Algorithms. *Operations Research*, 17(3):395–412, 1969.

11 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.

12 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.

13 Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* Search on Time-Dependent Road Networks. *Networks*, 59:240–251, 2012. Best Paper Award.

14 Ariel Orda and Raphael Rom. Traveling without waiting in time-dependent networks is NP-hard. Technical report, Dept. Electrical Engineering, Technion-Israel Institute of Technology, 1989.

15 Luigi Di Puglia Pugliese and Francesca Guerriero. A survey of resource constrained shortest path problems: Exact solution approaches. *Networks*, 62(3):183–200, 2013.

16 Ben Strasser and Tim Zeitz. A* with perfect potentials, 2019. arXiv:1910.12526.

17 Marieke van der Tuin, Mathijs de Weerdt, and Gernot Veit Batz. Route Planning with Breaks and Truck Driving Bans Using Time-Dependent Contraction Hierarchies. In *Proceedings of the Twenty-Eigth International Conference on Automated Planning and Scheduling*. AAAI Press, 2018. URL: https://www.semanticscholar.org/paper/Route-Planning-with-Breaks-and-Truck-Driving-Bans-Tuin-Weerdt/85c067c0a033f11166d114fcfde093d3250bb8fd.

Space and Time Trade-Off for the k Shortest Simple Paths Problem

Ali Al Zoobi
Université Côte d'Azur, Inria, CNRS, I3S, Sophia Antipolis Cedex, France
http://www-sop.inria.fr/members/Ali.Al-Zoobi/
ali.al-zoobi@inria.fr

David Coudert
Université Côte d'Azur, Inria, CNRS, I3S, Sophia Antipolis Cedex, France
http://www-sop.inria.fr/members/David.Coudert/
david.coudert@inria.fr

Nicolas Nisse
Université Côte d'Azur, Inria, CNRS, I3S, Sophia Antipolis Cedex, France
http://www-sop.inria.fr/members/David.Coudert/
nicolas.nisse@inria.fr

Abstract

The k shortest simple path problem (kSSP) asks to compute a set of top-k shortest simple paths from a vertex s to a vertex t in a digraph. Yen (1971) proposed the first algorithm with the best known theoretical complexity of $O(kn(m + n \log n))$ for a digraph with n vertices and m arcs. Since then, the problem has been widely studied from an algorithm engineering perspective, and impressive improvements have been achieved.

In particular, Kurz and Mutzel (2016) proposed a *sidetracks-based* (SB) algorithm which is currently the fastest solution. In this work, we propose two improvements of this algorithm.

We first show how to speed up the SB algorithm using dynamic updates of shortest path trees. We did experiments on some road networks of the 9th DIMAC'S challenge with up to about half a million nodes and one million arcs. Our computational results show an average speed up by a factor of 1.5 to 2 with a similar working memory consumption as SB. We then propose a second algorithm enabling to significantly reduce the working memory at the cost of an increase of the running time (up to two times slower). Our experiments on the same data set show, on average, a reduction by a factor of 1.5 to 2 of the working memory.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Shortest paths; Theory of computation → Design and analysis of algorithms

Keywords and phrases k shortest simple paths, graph algorithm, space-time trade-off

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.18

Supplementary Material The code of our algorithms is publicly available at `https://gitlab.inria.fr/dcoudert/k-shortest-simple-paths`.

Funding This work has been supported by the French government, through the UCA^{JEDI} Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-15-IDEX-01, the ANR project MULTIMOD with the reference number ANR-17-CE22-0016, the ANR project Digraphs with the reference number ANR-19-CE48-0013, and by Région Sud PACA.

1 Introduction

The classical k shortest paths problem (kSP) returns the top-k shortest paths between a pair of source and destination nodes in a graph. This problem has numerous applications in various kinds of networks (road and transportation networks, communications networks,

social networks, etc.) and is also used as a building block for solving optimization problems. Let $D = (V, A)$ be a digraph, an s-t path is a sequence $(s = v_0, v_1, \cdots, v_l = t)$ of vertices starting with s and ending with t, such that $(v_i, v_{i+1}) \in A$ for all $1 \leq i < l$. It is called simple if it has no repeated vertices, i.e., $v_i \neq v_j$ for all $0 \leq i < j \leq l$. The length of a path is the sum of the weights of its arcs and the top-k shortest paths is therefore the set containing a shortest s-t path, a second shortest s-t paths, *etc.* until the k^{th} shortest s-t path.

Several algorithms for solving kSP have been proposed. In particular, Eppstein [5] proposed an exact algorithm that computes k shortest paths (not necessarily simple) with time complexity in $O(m + n \log n + k)$, where m is the number of arcs and n the number of vertices of the graph. An important variant of this problem is the k shortest *simple* paths problem (kSSP) introduced in 1963 by Clarke *et al.* [3] which adds the constraint that reported paths must be simple. This variant of the problem has various applications in transportation network when paths with repeated vertices are not desired by the user. It is also a subproblem of other important problems like constrained shortest path problem, vehicle and transportation routing [10, 12, 19]. It can be applied successfully in bio-informatics [1], especially in biological sequence alignment [17] and in natural language processing [2]. For more applications, see Eppstein's recent comprehensive survey on k-best enumeration [6].

The algorithm with the best known time complexity for solving the kSSP problem has been proposed by Yen [20], with time complexity in $O(kn(m + n \log n))$. Since, several works have been proposed to improve the efficiency of the algorithm in practice [10, 14, 11, 7, 16].

Recently, Kurz and Mutzel [16, 15] obtained a tremendous running time improvement, designing an algorithm with the same flavor as Eppstein's algorithm. The key idea was to postpone as much as possible the computation of shortest path trees. To do so, they define a path using a sequence of shortest path trees and *deviations*. With this new algorithm, they were able to compute hundreds of paths in graphs with million nodes in about one second, while previous algorithms required an order of tens of seconds on the same instances. For instance, Kurz and Mutzel's algorithm computed $k = 300$ hundred shortest paths in 1.15 second for the COL network [4] while it required about 80 seconds for the Yen's algorithm and about 30 seconds by its improvement by Feng [7].

Our contribution. We propose two variations of the algorithm proposed by Kurz and Mutzel. We first show how to speed up their algorithm using dynamic updates of shortest path trees resulting in an average speed up by a factor of 1.5 to 2 and with a similar working memory consumption (i.e., the total memory consumption excluding the memory allocated for the input and the output). We then propose a second algorithm enabling a significant reduction of the working memory at the cost of a small increase of the running time.

This paper is organized as follows. First, in Section 2.2, we describe Yen's algorithm and then show in Section 2.3, how Kurz and Mutzel's algorithm improves upon it. In Section 3, we present our algorithms by precisely describing how they differ from Kurz and Mutzel's algorithm. Finally, Section 4 presents our simulation settings and results.

2 Preliminaries

2.1 Definition and Notation

Let $D = (V, A)$ be a directed graph (digraph for short) with vertex set V and arc set A. Let $n = |V|$ be the number of vertices and $m = |A|$ be the number of arcs of D. Given a vertex $v \in V$, $N^+(v) = \{w \in V \mid vw \in A\}$ denotes the out-neighbors of v in D. Let $w_D : A \to \mathbb{R}^+$ be a length function over the arcs. For every $u, v \in V$, a *(directed) path* from

u to v in D is a sequence $P = (v_0 = u, v_1, \cdots, v_r = v)$ of vertices with $v_i, v_{i+1} \in A$ for all $0 \leq i < r$. Note that vertices may be repeated, i.e., paths are not necessarily simple. A path is *simple* if, moreover, $v_i \neq v_j$ for all $0 \leq i < j \leq r$. The length of the path P equals $w_D(P) = \sum_{0 \leqslant i < r} w_D(v_i, v_{i+1})$ (we will omit D when there is no ambiguity). The distance $d_D(u, v)$ between two vertices $u, v \in V$ is the shortest length of a u-v path in D (if any). Given two paths $(v_1, \cdots, v_r)$ and $Q = (w_1, \cdots, w_p)$ and $v_r w_1 \in A$, let us denote the v_1-w_p path obtained by the concatenation of the two paths by $(v_1, \cdots, v_r, Q)$.

Given $s, t \in V$, a *top-k set of shortest s-t paths* is any set S of (pairwise distinct) simple s-t paths such that $|S| = k$ and $w(P) \leq w(P')$ for every s-t path $P \in S$ and s-t path $P' \notin S$. The k shortest simple paths problem takes as input a weighted digraph $D = (V, A)$, $w_D : A \to \mathbb{R}^+$ and a pair of vertices $(s, t) \in V^2$ and asks to find a top-k set of shortest s-t paths (if they exist).

Let $t \in V$. An *in-branching* T rooted at t is any sub-digraph of D that induces a tree containing t, such that every $u \in V(T) \setminus \{t\}$ has exactly one out-neighbor (that is, all paths go toward t). An in-branching T is called a *shortest path (SP) in-branching* rooted at t if, for every $u \in V(T)$, the length of the (unique) u-t path P^T_{ut} in T equals $d_D(u, t)$. Note that an SP in-branching is sometimes called *reversed shortest path tree*.

In the forthcoming algorithms, the following procedure will often be used (and the key point when designing the algorithms is to limit the number of such calls and to optimize each of them). Given a sub-digraph H of D and $u, t \in V(H)$, we use Dijkstra's algorithm for computing an SP in-branching rooted in t that contains a shortest u-t path in H. Note that, the execution of the Dijkstra's algorithm may be stopped as soon as a shortest u-t path has been computed (when u is reached), i.e., the in-branching may only be partial (i.e., not spanning D). The key point will be that this way to proceed not only returns a shortest u-t path in H (if any) but an SP in-branching rooted in t, containing u. Note that any such call has worst-case time complexity $O(m + n \log n)$.

Let $P = (v_0, v_1, \cdots, v_r)$ be any path in D and $i < r$. Any arc $a = v_i v' \neq v_i v_{i+1}$ is called a *deviation* of P at v_i. Moreover, any path $Q = (v_0, \cdots, v_i, v', v'_1, \cdots, v'_\ell = v_r)$ is called an *extension* of P at a (or at v_i). Note that neither P nor Q is required to be simple. However, if Q is simple, it will be called a *simple extension* of P at a (or at v_i). In addition, Q is called a shortest (simple) extension at v_i if and only if Q is an extension with minimum length among all (simple) extensions of P at v_i. Furthermore, Q is called a shortest (simple) extension at a if and only if Q is an extension with minimum length among all (simple) extensions of P at a.

2.2 General framework: Yen's algorithm and Feng's improvements

We start by describing the general framework used by the kSSP algorithms in [10, 14, 11, 7, 16]. Precisely, let us describe Yen's algorithm [20] trying to give its main properties and drawbacks. Then, we explain how Kurz and Mutzel's algorithm improves upon it (Section 2.3). Finally, we will detail our adaptation of the latter method in Section 3.

All of the algorithms described below start by computing a shortest s-t path $P_0 = (s = v_0, v_1, \cdots, v_r = t)$ (in what follows we always assume that there is at least one such path). This is done by applying Dijkstra's algorithm from t (as described in previous section), so also computes an SP in-branching T_0 rooted at t and containing s. Note that P_0 is simple since weights are non-negative. Obviously, a second shortest s-t simple path must be a shortest simple extension of P_0 at one of its vertices. Yen's algorithm computes a shortest simple extension of P_0 at v_i for every vertex v_i in P_0 as follows. For every $0 \leq i < r$, let $D_i(P_0)$ be the graph obtained from D by removing the vertices $v_0, \cdots, v_{i-1}$ (this is to avoid

non simple extension) and the arc v_iv_{i+1} (to ensure that the computed path is a new one, i.e., different from P_0). For every $0 \leq i < r$, an SP in-branching in $D_i(P_0)$ rooted at t is computed using Dijkstra's algorithm until it reaches v_i and therefore returns a shortest path Q_i from v_i to t. Note that Yen's algorithm no longer uses the SP in-branchings and this will be one key improvement described further. For every $0 \leq i < r$, the extension $(v_0, \cdots, v_i, Q_i)$ of P_0 at v_i is added to a set *Candidate* (initially empty). Note that the index i (called below *deviation-index*) where the path $(v_0, \cdots, v_i, Q_i)$ deviates from P_0 is kept explicit[1]. Once $(v_0, \cdots, v_i, Q_i)$ has been added to *Candidate* for all $0 \leq i < r$, by remark above, a shortest path in *Candidate* is a second shortest s-t simple path.

More generally, by induction on $0 < k' < k$, let us assume that a top-k' set S of shortest s-t paths has been computed and the set *Candidate* contains a set of simple s-t paths such that there exists a shortest path $Q \in Candidate$ such that $S \cup \{Q\}$ is a top-$(k'+1)$ set of shortest s-t paths. Yen's algorithm pursues as follows. Let $Q = (v_0 = s, \cdots, v_r = t)$ be any shortest path in *Candidate*[2] and let $0 \leq j < r$ be its deviation-index. First, Q is extracted from *Candidate* and it is added to S (as the $(k'+1)^{th}$ shortest s-t path). Then, every shortest extension of Q is added to *Candidate* (since they are potentially a next shortest s-t path). For this purpose, for every $j \leq i < r$, let $D_i(Q)$ be the digraph obtained from D by first removing the vertices $v_0, \cdots, v_{i-1}$ (this is to avoid non simple extension). Then, here is one important bottleneck of Yen's algorithm, for every arc v_iv' such that *Candidate* already contains some path with prefix $(v_0, \cdots, v_i, v')$, then v_iv' is removed from $D_i(Q)$. This therefore ensures to compute only new paths. Indeed, the computed extensions are distinct from every path previously computed as they have different prefixes (this is the reason to keep explicitly the deviation-index). For every $j \leq i < r$, an SP in-branching rooted at t is computed using Dijkstra's algorithm until it reaches v_i and therefore returns a shortest path Q_i from v_i to t in $D_i(Q)$. For every $0 \leq i < r$, the extension $(v_0, \cdots, v_i, Q_i)$ of Q at v_i (together with its deviation index i) is added to the set *Candidate*. This process is repeated until k paths have been found (when $k' = k$).

Therefore, for each path Q that is extracted from *Candidate*, $O(|V(Q)|)$ applications of Dijkstra's algorithm are done. This gives an overall time-complexity of $O(kn(m + n \log n))$ which is the best theoretical (worst-case) time-complexity currently known (and of all algorithms described in this paper) to solve the k-shortest simple paths problem.

One expensive part in the pre-described framework of Yen's algorithm are the multiple calls of Dijkstra's algorithm. Feng [7] proposed a practical improvement by trying to avoid some calls. Essentially, when a path $Q = (v_0, \cdots, v_r)$ with deviation-index j is extracted, its extensions are computed from $i = j$ to $r - 1$. Roughly, for every $j < i \leq r$, the computation of the extension at v_i is actually done with the help of the initial SP in-branching T_0.

In practice, this process significantly accelerates the executions of Dijkstra's algorithm. At the price of a larger memory consumption, Kurz and Mutzel improved Yen's framework which leads to the fastest algorithm currently known (Section 2.3) for the k shortest simple paths problem.

---

[1] The deviation-index is not kept explicitly in Yen's algorithm but, since it is a trivial improvement already existing in the literature, we mention it here.

[2] Actually *Candidate* is implemented, using a pairing heap, in such a way that extracting a shortest path in it takes logarithmic time and insertions are done in constant time.

2.3 Kurz and Mutzel's algorithm

All of Yen's improvements aim at minimizing the time consumed by Dijktra's algorithm calls. Instead, Kurz and Mutzel [16] chose to use a smaller number of such calls. This can be done by memorizing the SP in-branchings previously computed by the algorithm. More precisely, instead of keeping the paths in the set $Candidate$, the algorithm keeps only a representation of it using a sequence of SP in-branchings and deviations. These representations will allow to extract any shortest path P in time $O(|P|)$ and the length of P in constant time. As a result, for each shortest s-t path P, the memorized SP in-branching can be used to extract a shortest extension of P at a vertex v_i in a pivot step. Unfortunately, there is no guarantee that the extracted shortest extension will be simple. If it is not simple, a new Dijsktra's algorithm call has to be done. However, in many cases the extracted extension is simple and a Dijsktra's algorithm call can be avoided.

Precisely, Kurz and Mutzel's algorithm mainly relies on two key improvements. First, following a principle of Eppstein's algorithm [5], it explicitly keeps the SP in-branchings computed during the execution of the algorithm (this is achieved at some non-negligible cost of working memory consumption, but leads to an improvement of the practical running-time). Moreover, instead of computing the extensions of the extracted path in each iteration, the algorithm adds to $Candidate$ a representation of each extension (together with a lower bound of its length). Then, only when such a representation is extracted from $Candidate$, the corresponding extension is explicitly computed. This way of postponing the computations allows to avoid the actual computation of many extensions (which are not used any further), which leads to a drastic improvement of the running time.

Let us describe the Kurz and Mutzel's algorithm whose pseudo code is given in Algorithm 1. As usual, the algorithm starts with the computation of a shortest s-t (simple) path $P_0 = (v_0 = s, v_1, \cdots, v_r = t)$ together with an SP in-branching tree T_0 rooted at t and containing s. T_0 is added to a set $\mathcal{T}$ initially empty (this set $\mathcal{T}$ will contain all computed SP in-branchings). Then, for every $0 \leq i < r$, and for every deviation e at v_i (i.e., arcs $e = v_iv'$ are considered for every $v' \in N^+(v_i) \setminus \{v_0, \cdots, v_{i+1}\}$), let $P_{v't}(T_0)$ be the shortest path from v' to t in T_0. Note that the path $Q(i, e) = (v_0, \cdots, v_i, v', P_{v't}(T_0))$ is a shortest extension of P_0 at e, but it is not necessarily simple (it is not simple if $P_{v't}(T_0)$ intersects $\{v_0, \cdots, v_i\}$). Hence, $lb(e) = w((v_0, \cdots, v_i)) + w(v_iv') + w(P_{v't}(T_0))$ is a lower bound on the length of any shortest simple extension of P_0 at e (and it is its actual length if the path $Q(i, e)$ is simple). The algorithm proceeds as follows. First, by categorizing the vertices of T_0 (using a trick due to Feng [7] that we do not detail here), it is possible to decide in constant time, for each $i < r$ and each deviation e at v_i, whether $Q(e, i)$ is simple or not. Then, for every $0 \leq i < r$, and for every deviation e at v_i, the algorithm adds $((T_0, e, T_0), lb(e))$ in a heap (ordered using lb) $Candidate_{simple}$ if $Q(e, i)$ is simple, and it adds $((T_0, e, T_i), lb(e))$ in a heap $Candidate_{not-simple}$ otherwise, where T_i is the name of a new SP in-branching rooted at t in $D \setminus \{v_0, \cdots, v_i\}$ whose actual computation is postponed. Hence, T_0 is the only SP in-branching that has been computed (using Dijkstra's algorithm) so far.

More generally, by induction on $0 < k' < k$, let us assume that a top-k' set S of shortest s-t paths and two heaps $Candidate_{simple}$ and $Candidate_{not-simple}$ have been computed. Following Eppstein's idea, each element of these heaps is of the form $((T_0, e_0, \cdots, T_h, e_h, T_{h+1}), lb)$ (describing a path as explained below) such that, for every $0 \leq i \leq h$, T_i is an SP in-branching that has already been computed and stored in $\mathcal{T}$, while (only if the element comes from $Candidate_{not-simple}$) T_{h+1} may not have already been computed but has a pointer associated to it stored in $\mathcal{T}$. That is, even if T_{h+1} has not yet been computed, it is defined and can be referred to. Observe that we may have $T_j = T_{j+1}$ for some $0 \leq j \leq h + 1$.

Algorithm 1 Kurz-Mutzel Sidetrack Based (SB) algorithm for the kSSP [16].

Require: A digraph $D = (V, A)$, a source $s \in V$, a sink $t \in V$, and an integer k
Ensure: k shortest simple s-t paths

Let $Candidate_{simple} \leftarrow \emptyset$, $Candidate_{not-simple} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$ and $Output \leftarrow \emptyset$
$T_0 \leftarrow$ an SP in-branching of D rooted at t containing s
Add $((T_0), w(P_{st}(T_0)))$ to $Candidate_{simple}$
while $Candidate_{simple} \cup Candidate_{not-simple} \neq \emptyset$ and $|Output| < k$ **do**
    $\varepsilon = ((T_0, e_0, \cdots, T_h, e_h = (u_h, v_h), T_{h+1}), lb) \leftarrow$ a shortest element in $Candidate_{simple}$ and $Candidate_{not-simple}$ // *with priority to* $Candidate_{simple}$
    if $\varepsilon \in Candidate_{simple}$ **then**
        Extract ε from $Candidate_{simple}$ and add it to the $Output$
        for every deviation $e = v_j v'$ with $v_j \in P_{v_h t}(T_{h+1})$ **do**
            $ext \leftarrow (T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T_{h+1})$
            $lb' \leftarrow lb - w(P_{v_j t}(T_{h+1})) + w(e) + w(P_{v' t}(T_{h+1}))$
            if ext represents a simple path **then**
                Add (ext, lb') to $Candidate_{simple}$
            else
                $T' \leftarrow$ the name of an SP in-branching of $D_h(P)$ // T' *is not computed yet*
                Add T' to $\mathcal{T}$
                Add $((T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T'), lb')$ to $Candidate_{not-simple}$
    else
        if T_{h+1} has not been computed yet **then**
            Compute T_{h+1}, an SP in-branching of $D_h(P)$ and add it to $\mathcal{T}$
        $\varepsilon' = ((T_0, e_0, \cdots, T_h, e_h, T_{h+1}), lb + w(P_{v' t}(T_{h+1})) - w(P_{v' t}(T_h)))$
        Add ε' to $Candidate_{simple}$
return $Output$

Let P_1 be the simple path that starts in s and, for every $0 \leq j \leq h$, follows the (already computed) tree T_j from the current vertex till the tail of the deviation e_j and then follows deviation e_j to its head. Hence, P_1 ends in the head z of e_h. Now, if the element is in $Candidate_{simple}$, we know by induction that the SP in-branching T_{h+1} has already been computed and that the path P obtained by concatenating P_1 and the shortest z-t path $P_{zt}(T_{h+1})$ is guaranteed to be simple and has length $w(P) = w(P_1) + w(P_{zt}(T_{h+1})) = lb(e_h)$. If the element is in $Candidate_{not-simple}$ (the shortest z-t path $P_{zt}(T_h)$ intersects P_1) the algorithm actually computes the SP in-branching T_{h+1} rooted at t (if not already done). Observe that the digraph in which T_{h+1} is computed is a subdigraph of the digraph in which T_h has been computed. Furthermore, $w(P_{zt}(T_{h+1})) \geq w(P_{zt}(T_h))$ (by setting $w(P_{zt}(T_{h+1})) = +\infty$ if there is no z-t path in T_{h+1}) and z is the only common vertex of P_1 and $P_{zt}(T_{h+1})$. Hence, the path P obtained by concatenating P_1 and the shortest z-t path $P_{zt}(T_{h+1})$ (if it exists) is guaranteed to be simple and has length $w(P) = w(P_1) + w(P_{zt}(T_{h+1})) \geq w(P_1) + w(P_{zt}(T_h)) = lb(e_h)$.

An iteration of Kurz and Mutzel's algorithm proceeds as follows. First an element $\varepsilon = ((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}), lb)$ with smallest lb is extracted from $Candidate_{simple}$ and $Candidate_{not-simple}$ (with priority for $Candidate_{simple}$ in case of equal lb). If ε was in $Candidate_{simple}$, the path P as defined above is the next shortest simple s-t path and it is added to the output. Then, all possible deviations of P along the path $P_{zt}(T_{h+1})$ are determined and added to $Candidate_{simple}$ or $Candidate_{not-simple}$ depending on whether they are simple or not (note that only a representation of them is build and not the actual path).

Otherwise, the algorithm actually computes the SP in-branching T_{h+1} (if not already done) to determine the shortest z-t path in T_{h+1} (if it exists), and adds T_{h+1} to $\mathcal{T}$. If such path exists, the algorithm adds to $Candidate_{simple}$ a new element $((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}), lb')$ describing a simple s-t path with length $lb' = w(P_1) + w(P_{zt}(T_{h+1})) = lb - w(P_{zt}(T_h)) + w(P_{zt}(T_{h+1}))$.

Actually, the same SP in-branching can be used for all deviations at the same vertex v_i of a given path P. So, for each vertex v_i in P, a single call of Dijkstra's algorithm is needed. As a result, finding all of the extensions of a given path P can be done in $O(|P|(m + n \log n))$ time. Therefore, the time complexity of Kurz and Mutzel's algorithm (in the worst case) is bounded by $O(kn(m + n \log n))$ as the algorithm extends no more than k paths and the number of vertices of each path is bounded by n.

Overall, Kurz and Mutzel's algorithm computes k shortest simple s-t paths with a much lower number of applications of Dijkstra's algorithm and so its running time is in general much better than the algorithms proposed by Yen or Feng. On the other hand, it requires to store many SP in-branchings previously computed which implies a larger working memory consumption.

3 Our contributions

We propose two independent variants of the SB algorithm (Algorithm 1). The first one, called SB*, gives, with respect to our experimental results, an average speed up by a factor of 1.5 to 2 compared with SB algorithm. The second one, called PSB (Parsimonious SB), is based on a different manner to handle non simple candidates in order to reduce the number of computed and stored SP in-branchings. This leads to a significant reduction of the working memory at the price of a slight increase in running time compared with SB algorithm.

3.1 The SB* algorithm

Here, we propose a variant of the SB algorithm, strongly based on Kurz and Mutzel's framework, that is a tiny modification of SB algorithm allowing to speed it up.

More precisely, each time a representation $(T_0, e_0, T_1 \cdots, e_{h-1} = (u_{h-1}, v_{h-1}), T_h, e_h = (u_h, v_h), T_{h+1})$ is extracted from $Candidate_{not-simple}$ and that T_{h+1} has not been computed yet (i.e., it is only a pointer), our algorithm does not compute T_{h+1} from scratch as SB algorithm does. Instead, the SB* algorithm creates a copy T of T_h, discards vertices of the path from v_{h-1} to u_h in T_h, and updates the SP in-branching T using standard methods for updating a shortest path tree [9]. Then, the pointer T_{h+1} is associated with the new in-branching T.

It is clear that the SB* algorithm computes (and stores) exactly the same number of in-branchings as the SB algorithm. The computational results presented in Section 4.2 show that this update procedure gives an average speed up by a factor of 1.5 to 2.

3.2 The PSB algorithm

Our main contribution is the Parsimonious SB algorithm (PSB) presented in this section whose main goal is to solve the k shortest simple paths problem with a good tradeoff between the running time and the working memory consumption. Indeed, a weak point of the SB algorithm comes from the fact that it keeps all the SP in-branchings that it computes throughout its execution in the memory . In order to reduce the working memory consumption, the main difference between the SB algorithm and the one presented here consists of the types of the elements that the PSB algorithm stores in the heap $Candidate_{not-simple}$ and

the way they are used. We now describe the PSB algorithm by detailing how its differs from the SB algorithm. Let us mention that the PSB algorithm uses a heap $Candidate_{simple}$ similar (i.e., containing exactly the same type of elements) to the one used by SB algorithm.

Let us start considering a step of PSB algorithm when an element $\varepsilon = ((T_0, e_0, T_1, e_1, \cdots, T_h, e_h = (u_h, v_h), T_{h+1}), lb)$ is extracted from $Candidate_{simple}$. The first difference between the SB algorithm is that T_{h+1} may have not yet been computed, in which case it must be computed at that step and stored in $\mathcal{T}$. Next, as the SB algorithm, the PSB algorithm first adds the (simple) path P corresponding to ε to the output. Then, it considers all deviations of P at the vertices between v_h and t, i.e., all arcs (not in P) with tail in $P_{v_h t}(T_{h+1})$. For every such deviation $e = uv$ with $u \in V(P_{v_h t}(T_{h+1}))$, by using Feng's "trick" (already mentioned without details), it can be decided in constant time whether it admits a simple extension, i.e., whether the path P_e that "follows" the path P_1 corresponding to ε from s to v_h, then follows the path $P_{v_h u}(T_{h+1})$, the arc e and finally the path $P_{vt}(T_{h+1})$ is simple or not. In the case when P_e is simple, then the element $((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, e, T_{h+1}), lb_e)$ is added to $Candidate_{simple}$, where $lb_e = w(P_1) + w(P_{v_h u}(T_{h+1})) + w(e) + w(P_{vt}(T_{h+1}))$ (exactly as it is done by the SB algorithm). The second difference with the SB algorithm relies on the set $X = \{f_1, \cdots, f_r\}$ of deviations for which the extension using T_{h+1} is not simple. The key point is that we create a single element for all deviations in X. This ensures that the size of $Candidate_{not-simple}$ is at most k, as at most one element is added to $Candidate_{not-simple}$ per path added to the output. More precisely, let $X = \{f_1, \cdots, f_r\}$ be the set of "non simple" deviations ordered in such a way that, for every $1 \leq i < j < l \leq r$, the tail of f_j is between (or equal to) the tails of f_i and f_l on the path $P_{v_h t}(T_{h+1})$. For every $1 \leq i \leq r$ and $f_i = u_i v_i$, let $lb_i = lb_{f_i} = w(P_1) + w(P_{v_h, u_i}(T_{h+1})) + w(f_i) + w(P_{v_i, t}(T_{h+1}))$. The PSB algorithm then adds the element $\varepsilon' = ((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, X, T_{h+1}), \min_{1 \leq i \leq r} lb_i)$ to $Candidate_{not-simple}$, and so the weight of ε' in the heap $Candidate_{not-simple}$ is the smallest lower bound among all lower bounds related to the "non simple" deviations in X.

Let us now consider a step of the PSB algorithm when an element is extracted from $Candidate_{not-simple}$. This happens, as in the SB algorithm, when the smallest key (lower bound) of the elements in $Candidate_{simple} \cup Candidate_{not-simple}$ corresponds only to an element of $Candidate_{not-simple}$. Let $\varepsilon = ((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, X, T_{h+1}), lb)$ be this element and let $X = \{f_1, \cdots, f_r\}$. Let also $e_h = u_h v'_h$, let $P_1 = (s = x_1, \cdots, x_o = v'_h)$ be the prefix (from s to v'_h) of the path represented by ε, let $P_{v'_h t}(T_{h+1}) = (v'_h, v'_{h+1}, \cdots, v'_p = t)$ and let $f_j = v'_{i_j} v_j$ for every $1 \leq j \leq r$ (by the way the f_j's are ordered, $h \leq i_j \leq i_{j'} \leq p$ for all $1 \leq j \leq j' \leq r$). The fact that the type of the elements in $Candidate_{not-simple}$ is more involved (so, allowing to decrease a lot the working memory) requires an advanced way to treat them (and potentially more costly in term of running time). To limit the increase of the running time, the PSB algorithm proceeds in such a way that several deviations in $\{f_1, \cdots, f_r\}$ are somehow considered "simultaneously". More precisely, it proceeds as follows.

Let $1 \leq i^* \leq r$ be the smallest integer such that $lb_{i*} = lb$. The PSB algorithm proceeds as follows to deal with the deviations $f_r, f_{r-1}, \cdots, f_{i^*}$ in this order. First, it applies Dijkstra's algorithm to compute an SP in-branching T'_r rooted at t in $D_r = D - \{x_1, \cdots, x_o = v'_h, \cdots, v'_{i_r}\}$ until it reaches v_r. If v_r is reached, then the path $Q_r = P_1 P_{v'_h v'_{i_r}}(T_{h+1}) P_{v_r t}(T'_r)$ is a simple s-t path and the element $((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, f_r, T'_r), w(Q_r))$ is added to $Candidate_{simple}$. However, the in-branching T'_r is not saved into $\mathcal{T}$ but only its name is kept (this allows to reduce the working memory size while it may require to recompute the tree T'_r later. The bet here is that it will not be necessary to actually redo this computation). Then, for $j = r - 1$ down to i^*, the SP in-branching T'_r is updated to become the SP in-branching T'_j rooted in t in $D_j = D - \{x_1, \cdots, x_o = v'_h, \cdots, v'_{i_j}\}$, possibly reaching v_j

and so providing a simple path Q_j. To speed up the computation of T'_j, it is actually computed by updating T'_{j+1} which is done using standard methods for updating a shortest path tree [9]. Finally, the element $((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, f_j, T'_j), w(Q_j))$ is added to $Candidate_{simple}$. In the current implementation of our PSB algorithm (the one used in the experiments described in next section), the in-branching T'_j is saved in $\mathcal{T}$ only if $j = i^*$[3]. Finally, the element $((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, X', T_{h+1}), \min_{1 \leq j < i^*} lb_j)$, with $X' = \{f_1, \cdots, f_{i^*-1}\}$, is added into $Candidate_{not-simple}$.

The correctness of the PSB algorithm follows from the one of the SB algorithm by noticing that the elements extracted from $Candidate_{simple} \cup Candidate_{not-simple}$ are the ones with smallest lower bound and the fact that, each time that a path is extracted, a shortest extension of each of its deviations is considered.

Finally, as already mentioned, the number of elements in the heap $Candidate_{not-simple}$ is bounded by k as each of its elements correspond to a path that has been added to the output, while with the SB algorithm, this heap may contain $O(km)$ elements. Furthermore, as for the SB algorithm, we may keep only the k elements with smallest lower bound in $Candidate_{simple}$. Hence, the working memory used by the PSB algorithm for the heaps is significantly smaller than for the SB algorithm. However, the largest part of the working memory is due to the number of SP in-branchings that are computed and stored in $\mathcal{T}$. Although this number seems difficult to evaluate, we observe experimentally (see Section 4) that it is significantly smaller with the PSB algorithm.

4 Experimental evaluation

4.1 Experimental settings

We have implemented[4] the algorithms NC (improvement of Yen's algorithm by Feng [7]), SB [16], SB* and PSB in C++14 and our code is publicly available at `https://gitlab.inria.fr/dcoudert/k-shortest-simple-paths`.

Following [16], we have implemented a pairing heap data structure [8] supporting the decrease key operation, and we use it for the Dijkstra shortest path algorithm. Our implementation of the Dijkstra shortest path algorithm is lazy, that is it stops computation as soon as the distance from query node v to t is proved to be the shortest one. Further computations might be performed later for another node v' at larger distance from t. Our implementation of Dijkstra's algorithm supports an update operation when a node or an arc is added to the graph. In addition, we have implemented a special copy operation that enables to update the in-branching when a set of nodes is removed from the graph. This corresponds to the operations performed when creating an in-branching T_{h+1} from T_h. Observe that in our implementations of NC, SB, SB* and PSB, the parameter k is not part of the input, and so the sets of candidates are simply implemented using pairing heaps. This choice enables to use these methods as iterators able to return the next shortest path as long as one exists (Note that if k is part of the input, the data structure used to store candidates could be changed in order to contain only the k best candidates, but the algorithm would only return exactly k paths even if more exist). We have evaluated the performances of our algorithms on some road networks from the 9th DIMACS implementation challenge [4]. The characteristics of these graphs are reported in Table 1. In the following, we refer to the graphs ROME,

[3] A way to establish an even better space versus time tradeoff would be to determine a good threshold τ such that an in-branching T'_j is stored in $\mathcal{T}$ if and only if $w(Q_j) \leq \tau$. Due to lack of time we have not been able to establish such a parameter τ but it will be one of the objectives of our future works.

[4] Despite several queries, we have not been granted access to the code used for experiments in [7, 16].

Table 1 Characteristics of the TIGER graphs used in kSSP experiments.

Area	ROME	DC	DE	NY	BAY	COL
Number of vertices	3 353	9 559	49 109	264 346	321 270	435 666
Number of edges	8 870	29 682	119 520	733 846	800 172	1 057 066

DC and DE as the small networks, and to the graphs NY, BAY and COL as the large networks. We also generated random networks using method `RandomGNM` of SageMath [18] with $n \in \{5000, 10000, 20000\}$ and for each n, we let $m = 10n, 50n$ and $100n$. For each network (both the random and the road networks), we have measured the execution time and the number of stored SP in-branchings. Note that the number of stored in-branchings gives a rigorous indication of the memory consumption (in particular, this is independent of the implementation) [13]. For each network we run the algorithms on a thousand pairs of vertices randomly chosen. We report in Tables 2 and 7 the average and the median of their running times and number of stored SP in-branchings.

All reported computations have been performed on computers equipped with 2 quad-core 3.20GHz Intel Xeon W5580 processors and 64GB of RAM.

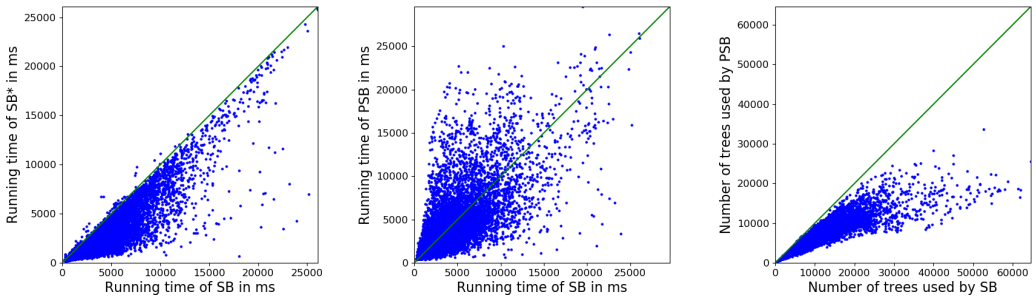
4.2 Experimental results

The simulations show that our tiny improvement SB* of SB algorithm allows to decrease the running-time by a factor between 1,5 (on NY) and 2 (on DE) on median (Tables 2 and 3). The fact that in each network a few queries are extremely slow makes the median a better indicator than the average. In particular, in all the networks considered, SB* algorithm is, for most of the queries, faster than SB algorithm (Figures 1a and 2a). By design, the number of stored in-branchings is the same in both algorithms. It is interesting to note that the gain increase with the size of the networks. It seems that the differences of performances depends on the structure of the queries and of the networks. In the future work, we plan to investigate the relationship between the kinds of queries and/or networks and the gain in running time.

The simulations comparing PSB algorithm and SB algorithm show a significant reduction of the working memory when using PSB (Tables 4 and 5 and Figures 1c and 2c). Again, the gain increases when considering larger networks. In term of running time, SB algorithm is slightly faster on average but Figures 1b and 2b indicate that globally, they are quite comparable. It would be interesting to understand the impact of the length of the queries on the performances of both algorithms.

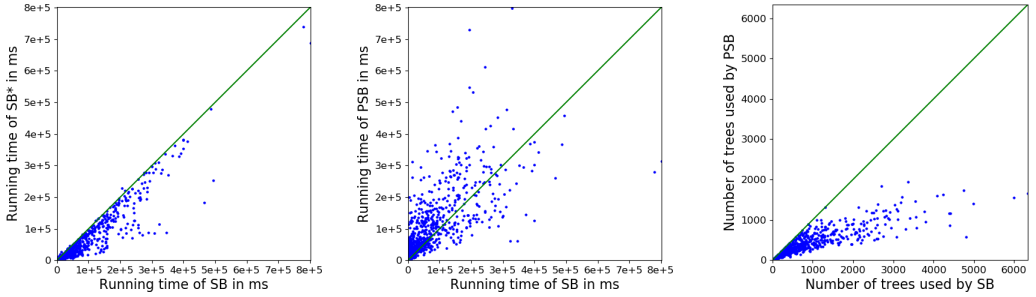
Finally, following some simulations in [16], we have also compared all the algorithms on random graphs (Edös-Rényi). Due to lack of time, we considered only one graph per setting (number of vertices, of edges and k) and the average is done on 1000 requests (note that this setting is similar to the one in [16]). The performances (Table 6) are similar than in the ones obtained for road networks. That is, the SB* algorithm is faster than all other algorithms (more than twice as fast in the case of large graphs when $k = 1000$). Surprisingly, the NC algorithm is sometimes (for dense graphs) faster than SB and PSB. Moreover, the PSB algorithm always uses less memory than SB algorithm (Table 7), with a more significant difference in the case of sparse large graphs.

In the future work, we will continue our experiments in order to discover which conditions (structure of graphs and queries...) make one of the prescribed algorithms faster or / and less memory consuming than the others.



(a) Running time of SB and SB*. **(b)** Running time of SB and PSB. **(c)** Number of trees of SB and PSB.

Figure 1 Comparison of the running time of SB versus SB* (Figure 1a) and SB versus PSB (Figure 1b) on ROME, and comparison of the number of stored trees for SB versus PSB (Figure 1c). Each dot corresponds to one pair source/destination ($k = 10,000$).



(a) Running time of SB and SB*. **(b)** Running time of SB and PSB. **(c)** Number of trees of SB and PSB.

Figure 2 Comparison of the running time of SB versus SB* (Figure 2a) and SB versus PSB (Figure 2b) on COL, and comparison of the number of stored trees for SB versus PSB (Figure 2c). Each dot corresponds to one pair source/destination ($k = 1,000$).

Table 2 Time consuming (average and median in ms) of SB, SB* and PSB on small networks.

Area		ROME			DC			DE		
k		100	1000	10,000	100	1000	10,000	100	1000	10,000
Avg	SB	43	440	4,502	15	175	2,051	773	7,867	82,916
	SB*	**24**	**272**	**2,939**	**11**	**118**	**1,388**	**532**	**5,721**	**61,294**
	PSB	42	427	4,380	21	248	2,859	865	8,762	90,226
Med	SB	33	347	3,663	8	74	893	403	5,382	42,613
	SB*	**15**	**185**	**2,105**	**6**	**45**	**538**	**196**	**2,184**	**28,294**
	PSB	30	314	3,252	9	101	1,185	654	6,517	73,046

Table 3 Time consuming (average and median in ms) of SB, SB* and PSB on big networks.

Area		NY			BAY			COL		
k		100	500	1000	100	500	1000	100	500	1000
Avg	SB	904	4,334	8,741	3,270	18,464	38,346	5,216	28,262	59,717
	SB*	**581**	**2,787**	**5,707**	**2,395**	**13,669**	**28,545**	**3,723**	**20,313**	**43,373**
	PSB	1,822	9,417	19,166	5,083	27,438	55,711	7,371	39,078	80,696
Med	SB	156	600	1,230	695	4,073	9,443	1,412	8,737	19,664
	SB*	**148**	**356**	**676**	**340**	**2,075**	**4,934**	**722**	**5,051**	**11,620**
	PSB	336	2,324	5,278	1,934	12,000	25,760	3,072	19,219	42,114

Table 4 Number of SP in-branching genereted and stored by SB and PSB on small networks.

Area		ROME			DC			DE		
k		100	1000	10,000	100	1000	10,000	100	1000	10,000
Avg	SB	106	1,108	11,446	14.9	209	2,594	88	928	9,945
	PSB	**53**	**667**	**6,956**	**10.6**	**140**	**1,716**	**36**	**390**	**4,212**
Med	SB	87	961	10,164	6	105	1,555	48	557	6,551
	PSB	**56**	**615**	**6,570**	**5**	**80**	**1,106**	**25**	**287**	**3,154**

Table 5 Number of SP in-branching genereted and stored by SB and PSB on big networks.

Area		NY			BAY			COL		
k		100	500	1000	100	500	1000	100	500	1000
Avg	SB	14.9	81	171	44.6	266	562	47	266	570
	PSB	**9.8**	**51**	**106**	**22**	**124**	**259**	**22**	**123**	**259**
Med	SB	3	21	45	13	90	209	13	101	234
	PSB	**2**	**16**	**35**	**9**	**57**	**124**	**9**	**63**	**142**

Table 6 Average time consuming (in ms) of NC, SB, SB* and PSB on random digraph with different densities.

Digraph		$n = 5,000$			$n = 10,000$			$n = 20,000$		
	m	$10n$	$50n$	$100n$	$10n$	$50n$	$100n$	$10n$	$50n$	$100n$
$k = 100$	NC	40	64	88	37	114	158	191	304	388
	SB	12	39	53	23	47	79	805	412	363
	SB*	**35**	**30**	**48**	**17**	**41**	**70**	**132**	**153**	**204**
	PSB	30	40	51	24	46	78	554	435	389
$k = 500$	NC	159	275	354	311	470	652	789	1211	1498
	SB	49	182	250	74	180	332	3869	1924	1637
	SB*	**54**	**121**	**213**	**40**	**139**	**271**	**690**	**573**	**787**
	PSB	66	175	229	74	166	314	3692	2005	1727
$k = 1000$	NC	313	546	697	617	924	1285	1545	2368	2920
	SB	98	370	512	144	356	669	7709	3890	3264
	SB*	**79**	**239**	**430**	**71**	**267**	**533**	**1010**	**1100**	**1528**
	PSB	112	349	463	145	322	620	5209	3978	3412

Table 7 Average of number of SP in-branching computed and stored using SB and SB* on random digraph with different densities.

Digraph		$n = 5,000$			$n = 10,000$			$n = 20,000$		
	m	$10n$	$50n$	$100n$	$10n$	$50n$	$100n$	$10n$	$50n$	$100n$
$k = 100$	SB	2.332	3.726	2.08	1.992	1.566	1.753	33.142	9.149	5.09
	PSB	**2.275**	**3.657**	**2.04**	**1.952**	**1.538**	**1.724**	**25.95**	**8.529**	**4.882**
$k = 500$	SB	8.88	16.07	7.477	6.287	4.615	5.493	161.844	42.866	22.094
	PSB	**8.434**	**15.57**	**7.175**	**6.041**	**4.465**	**5.269**	**126.23**	**39.603**	**21.018**
$k = 1000$	SB	17.477	32.207	14.98	12.023	8.623	10.532	323.231	85.178	43.193
	PSB	**16.538**	**31.132**	**14.34**	**11.471**	**8.307**	**10.059**	**252.151**	**78.28**	**40.948**

References

1 M. Arita. Metabolic reconstruction using shortest paths. *Simulation Practice and Theory*, 8(1-2):109–125, 2000. doi:10.1016/S0928-4869(00)00006-9.

2 M. Betz and H. Hild. Language models for a spelled letter recognizer. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 856–859. IEEE, 1995. doi:10.1109/ICASSP.1995.479829.

3 S. Clarke, A. Krikorian, and J. Rausen. Computing the n best loopless paths in a network. *Journal of the Society for Industrial and Applied Mathematics*, 11(4):1096–1102, 1963. doi:10.1137/0111081.

4 C. Demetrescu, A. Goldberg, and D. Johnson. 9th dimacs implementation challenge - shortest paths, 2006. URL: http://users.diag.uniroma1.it/challenge9/.

5 D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998. doi:10.1137/S0097539795290477.

6 David Eppstein. *k-Best Enumeration*, pages 1003–1006. Springer New York, New York, NY, 2016. doi:10.1007/978-1-4939-2864-4_733.

7 G. Feng. Finding k shortest simple paths in directed graphs: A node classification algorithm. *Networks*, 64(1):6–17, 2014. doi:10.1002/net.21552.

8 M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986. doi:10.1007/BF01840439.

9 D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000. doi:10.1006/jagm.1999.1048.

10 Eleni Hadjiconstantinou and Nicos Christofides. An efficient implementation of an algorithm for finding k shortest simple paths. *Networks*, 34(2):88–101, 1999. doi:10.1002/(SICI)1097-0037(199909)34:2<88::AID-NET2>3.0.CO;2-1.

11 J. Hershberger, M. Maxel, and S. Suri. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms*, 3(4):45, 2007. doi:10.1145/1290672.1290682.

12 W. Jin, S. Chen, and H. Jiang. Finding the k shortest paths in a time-schedule network with constraints on arcs. *Computers & operations research*, 40(12):2975–2982, 2013. doi:10.1016/j.cor.2013.07.005.

13 David S. Johnson. A theoretician's guide to the experimental analysis of algorithms. *Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges*, 59:215–250, 2002. doi:10.1090/dimacs/059/11.

14 N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for k shortest simple paths. *Networks*, 12(4):411–427, 1982. doi:10.1002/net.3230120406.

15 D. Kurz. *k-best enumeration - theory and application*. Theses, Technischen Universität Dortmund, March 2018. doi:10.17877/DE290R-19814.

16 D. Kurz and P. Mutzel. A sidetrack-based algorithm for finding the k shortest simple paths in a directed graph. In *Int. Symp. on Algorithms and Computation (ISAAC)*, volume 64 of *LIPIcs*, pages 49:1–49:13. Schloss Dagstuhl, 2016. doi:10.4230/LIPIcs.ISAAC.2016.49.

17 T. Shibuya and H. Imai. New flexible approaches for multiple sequence alignment. *Journal of Computational Biology*, 4(3):385–413, 1997. doi:10.1089/cmb.1997.4.385.

18 The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.9)*, 2019. https://www.sagemath.org.

19 W. Xu, S. He, R. Song, and S. S. Chaudhry. Finding the k shortest paths in a schedule-based transit network. *Computers & Operations Research*, 39(8):1812–1826, 2012. doi:10.1016/j.cor.2010.02.005.

20 J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971. doi:10.1287/mnsc.17.11.712.

An Algorithm for the Exact Treedepth Problem

James Trimble
School of Computing Science, University of Glasgow, UK
j.trimble.1@research.gla.ac.uk

Abstract

We present a novel algorithm for the minimum-depth elimination tree problem, which is equivalent to the optimal treedepth decomposition problem. Our algorithm makes use of two cheaply-computed lower bound functions to prune the search tree, along with symmetry-breaking and domination rules. We present an empirical study showing that the algorithm outperforms the current state-of-the-art solver (which is based on a SAT encoding) by orders of magnitude on a range of graph classes.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Algorithm design techniques

Keywords and phrases Treedepth, Elimination Tree, Graph Algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.19

Related Version A full version of the paper is available at `https://arxiv.org/abs/2004.08959`.

Supplementary Material Source code: `https://github.com/jamestrimble/treedepth-solver`

Funding *James Trimble*: This work was supported by the Engineering and Physical Sciences Research Council (grant number EP/R513222/1).

Acknowledgements Thanks to Ciaran McCreesh, David Manlove, Patrick Prosser and the anonymous referees for their helpful feedback, and to Robert Ganian, Neha Lodha, and Vaidyanathan Peruvemba Ramaswamy for providing software for the SAT encoding.

1 Introduction

This paper presents a practical algorithm for finding an optimal treedepth decomposition of a graph. A *treedepth decomposition* of graph $G = (V, E)$ is a rooted forest F with node set V, such that for each edge $\{u, v\} \in E$, we have either that u is an ancestor of v or v is an ancestor of u in F. The *treedepth* of G is the minimum depth of a treedepth decomposition of G, where depth is defined as the maximum number of vertices along a path from the root of the tree to a leaf.

Treedepth is closely related to a number of other problems. The treedepth of a connected graph G equals the minimum height of an elimination tree for G ([12], chapter 6), which equals the graph's vertex ranking number [3]. The treedepth of a graph G is also equal to the minimum number of colours in a centred colouring of G [12].

Finding an elimination tree of small height is applicable to the parallel Cholesky factorisation of sparse matrices [17]. Treedepth also has relevance to the design of fixed-parameter tractable (FPT) algorithms. For example, the Mixed Chinese Postman Problem is FPT when parameterised by treedepth, but W[1]-hard when parameterised by treewidth or pathwidth [9].

The decision variant of the treedepth problem is NP-complete [13]. However, it can be solved in linear time if the input graph is a tree [16], and in polynomial time for interval graphs [1], trapezoid graphs, permutation graphs and circular arc graphs [4]. There is a polynomial-time approximation algorithm for the problem that gives a result within $O(\log^2 n)$ of the optimal value. The problem is fixed parameter tractable with respect to both treewidth and treedepth [2, 14].

Although numerous heuristics for finding good elimination trees have been designed and implemented [8], we are aware of only two existing implementations of *exact* algorithms for minimum treedepth decomposition; both of these are introduced in [6, 7]. In each case, the optimisation problem is solved as a sequence of decision problems, with each decision problem encoded as an instance of the boolean satisfiability problem and solved using a general-purpose SAT solver.

This paper's contribution. This paper introduces a new algorithm for computing an optimal treedepth decomposition. The algorithm is self-contained and does not require an external solver, although it can optionally use the graph-automorphism library Nauty to break symmetries. The basic structure of the algorithm is very simple. To improve performance, three symmetry breaking and domination features and two lower-bounding functions are added to the algorithm. In a set of experiments, we show that our algorithm is typically orders of magnitude faster than the current (SAT-based) state of the art.

Structure of the paper. Section 2 introduces concepts and notation. Section 3 presents the core parts of our algorithm. Section 4 describes enhancements to the basic algorithm. Section 5 provides details of our implementation. Section 6 presents an experimental comparison with the existing SAT encoding for treedepth. Section 7 concludes.

2 Preliminaries

Let $G = (V, E)$ be a graph, where we assume that the elements of V are integers. We write $V(G)$ and $E(G)$ to denote the vertex and edge sets of G. The neighbourhood $N_G(v)$ of a vertex v is the set of vertices that are adjacent to v. For $S \subseteq V$, we denote by $G[S]$ the subgraph of G induced by S; that is, $(S, \{\{u, v\} \in E \mid u, v \in S\})$. We use the notation $G - v$ for the removal of one vertex and its incident edges; that is, $G - v = G[V(G) \setminus \{v\}]$.

The concepts *elimination forest* and *elimination tree* are defined recursively in terms of one another. An elimination forest of graph G is a rooted forest with vertex set $V(G)$ composed of elimination trees of each of G's connected components. An elimination tree of a non-empty connected graph C is a rooted tree T with vertex set $V(C)$. If C has only one vertex v, then T is a tree containing only v. Otherwise, T is formed by choosing a vertex $v \in V(C)$ as the root, finding an elimination forest of $C - v$, and making the trees in that forest the child subtrees of v.

We will use the graph G in Figure 1 to provide an example of an elimination tree, and as our running example throughout the paper. The second part of the figure shows an optimal elimination tree of G, which has depth 4. Observe that removing the root vertex of the tree (5) from G splits the graph into two connected components, and each of these components corresponds to one of the child subtrees of 5 in the elimination tree.

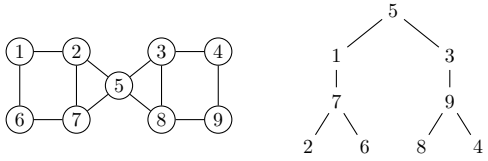


Figure 1 An example graph G (left) and an optimal elimination tree of G (right).

Every elimination forest is a treedepth decomposition, and for a given graph G there is at least one elimination forest whose depth equals the treedepth of G [12, 11]. Therefore, in order to find an optimal treedepth decomposition of G it is sufficient to search for a mimumum-depth elimination tree of G. This is the approach taken in this paper.

3 The Algorithm

Our algorithm is shown as pseudocode in Algorithm 1. The shaded parts, and the third parameter of each of the first two functions, will be introduced in later sections and can be disregarded for now.

Algorithm 1 An algorithm to find an elimination forest of minimum depth. To read the basic algorithm (without optimisations), disregard the shaded sections and the third parameter of each of the first two functions.

```
1  elimination_forest(G, k, w)
2  Data: Graph G, maximum depth k, and parent vertex w
3  Result: true if and only if an elimination forest of G with depth ≤ k exists
4  begin
5      if k = 0 and |V(G)| > 0 then return false
6      C ← the connected components of G
7      for C ∈ C do
8          if simple_lower_bound(|V(C)|) > k then return false
9      for C ∈ C do
10         if can_prune_by_path_lower_bound(C, k) then return false
11     for C ∈ C do
12         if not elimination_tree(C, k, w) then return false
13     return true

14 elimination_tree(G, k, w)
15 Data: Connected, nonempty graph G, maximum depth k ≥ 1, and parent vertex w
16 Result: true if and only if an elimination tree of G with depth ≤ k exists
17 begin
18     if |V(G)| = 1 then
19         v ← the unique element of V(G)
20         parent[v] ← w
21         return true
22     for v ∈ V(G) do
23         if v is ruled out by a symmetry or domination rule then continue
24         parent[v] ← w
25         if elimination_forest(G − v, k − 1, v) then return true
26     return false

27 optimise(G)
28 Data: A graph G
29 Result: The treedepth of G
30 begin
31     k ← 0
32     while elimination_forest(G, k, 0) = false do k ← k + 1
33     return k
```

Inputs and outputs. The algorithm's first function, `elimination_forest`(), takes a graph G and an integer $k \geq 0$, and returns *true* if and only if there exists an elimination forest of G of depth k or less. The function `elimination_tree`() takes a connected, non-empty graph G and an integer $k > 0$ and returns *true* if and only if there exists an elimination tree of depth k or less. The algorithm is run by calling `optimise`(), which takes a graph G and returns the treedepth of G.

Details of the functions. The first two functions are mutually recursive, and closely follow the definitions of elimination tree and forest. The function `elimination_forest`() begins by returning *false* – indicating infeasibility – if a elimination tree of depth zero is sought for a non-empty graph. The function then returns *true* if and only if an elimination tree of depth no greater than k exists for each connected component of the graph.

The function `elimination_tree`(G, k) returns *true* if G has a single vertex (lines 18 to 21). Otherwise, it tries each vertex $v \in V(G)$ in turn (line 22), and returns *true* if and only if one of these v values can be the root of an elimination tree of depth k – which is the case if and only if an elimination forest of depth no greater than $k-1$ exists for $G - v$.

The main function, `optimise`(), carries out repeated calls to `elimination_forest`() with ascending values of k until a feasible depth is reached. It would be possible to implement a branch-and-bound variant of the algorithm with a little additional effort, and it is likely that this would be somewhat faster than the approach we have taken. We decided not to do so for two reasons. First, having a sequence of decision problems simplifies the exposition of the algorithm. Second, we have observed in practice that two of the decision problems – the final unsatisfiable problem and the satisfiable problem after which the algorithm terminates – take up most of the run time. This suggests that a branch-and-bound approach would be of limited benefit.

Example. Returning to our example graph G from Figure 1, suppose G is passed to `elimination_tree`(). Figure 2 illustrates two of the nine subproblems explored at line 22. In the left part of the figure, $v = 1$. Choosing this vertex as the root of the elimination tree leaves a connected graph, which is passed to `elimination_forest`() at line 25. The right part of the figure corresponds to the decision $v = 5$. Choosing this vertex as the root leaves a disconnected graph, with two four-vertex components. This disconnected graph is passed to `elimination_forest`() at line 25, and subsequently `elimination_tree`() is called for each of the two components.

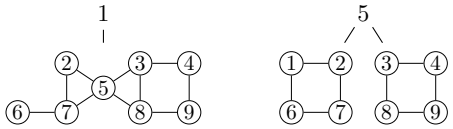


Figure 2 Two of the nine subproblems visited by the first call to `elimination_tree` on our example graph.

3.1 Generating an Elimination Forest

The algorithm described so far returns only a single integer: the treedepth of the input graph. We can easily modify the algorithm to also produce an elimination forest of that depth.

We assume that vertices of the input graph G are numbered from 1 to $|V(G)|$. A global array with $|V(G)|$ elements, *parent*, is used to record the parent of each vertex in the elimination forest. A value $parent[v] = 0$ indicates that vertex v is a root. Lines 20 and 24 of Algorithm 1 record the parent of v.

The functions `elimination_forest`() and `elimination_tree`() both have w as an extra parameter. Vertex w will be parent to the first vertex chosen from G. At the first level of recursion for either of these functions, $w = 0$, indicating that the next vertex to be selected will be a root.

When either `elimination_forest`() and `elimination_tree`() returns *true*, this indicates not only that an elimination tree of depth k of the subgraph G exists, but also that such a decomposition has been recorded in the *parent* array (with any *parent* value not in $V(G)$ indicating a root of the subproblem's decomposition).

4 Enhancements to the Algorithm: Symmetry Breaking and Domination Rules, Pruning, and Sorting

In this section we describe five improvements that can be made to the basic algorithm. The first three of these are symmetry breaking and domination rules that allow us to avoid choosing some values of v at line 22 of `elimination_tree`(). The fourth technique allows us to prune subproblems by quickly computing a lower bound on treedepth; we introduce two such bounds. The final technique is a re-ordering of vertices before solving.

4.1 Symmetry Breaking and Domination Rules

Recall that the loop at line 22 of `elimination_tree`() tries each vertex v as a potential root of the elimination tree, attempting to find an elimination tree of depth k or less. For some graphs, there are several values of v that may be chosen as the root of such a tree; let S be the set of such vertices. We can omit some choices of v at line line 22 without affecting the algorithm's correctness, provided at least one member of S is chosen. The three symmetry-breaking and domination rules that follow make use of this fact to avoid visiting some vertices. They ensure that the least-numbered vertex in S is visited, if S is non-empty.

Symmetry breaking using vertex orbits. Our first symmetry breaking technique is applied only to connected graphs. Before beginning the algorithm, we use Nauty [10] to compute the orbits of the vertices. (Recall that vertices v and v' are in the same orbit if and only if there is an automorphism that maps v to v'.) In the first call to `elimination_tree`() – that is, the call where G is the full input graph – we can avoid choosing any value of v in the loop if there exists a vertex $v' < v$ that is in the same orbit as v. This symmetry-breaking technique is valid because the subgraph created by the removal of v is isomorphic to the subgraph created by the removal of v', and thus has the same treedepth.

As an example, consider the graph in Figure 1. Since vertices 1, 4, 6, and 9 are in the same orbit, we can avoid choosing vertices 4, 6, and 9 in the first call to `elimination_tree`().

This technique is useful on highly symmetrical graphs, but of course cannot be expected to achieve a speedup of more than $|V(G)|$. A number of avenues for improved symmetry breaking could be explored in future work. It would be straightforward to extend the symmetry breaking to disconnected graphs by computing the orbits of vertices in each connected component separately. Our technique for symmetry breaking could also be used for subproblems; this may be useful for very symmetrical graphs, but we suspect that the cost of additional calls to Nauty would outweigh the benefit in many cases. We could move beyond finding the orbits of single vertices; it is possible to find the orbits of pairs (or larger sets) of vertices with a single call to Nauty and a small amount of extra work. Lastly, we could use symmetry-breaking techniques from constraint programming such as GE-trees [15].

Vertex domination. Suppose we have $v, v' \in V(G)$ (not necessarily adjacent) such that $v' < v$ and $N_G(v') \setminus \{v\} \supseteq N_G(v) \setminus \{v'\}$. We say that v' *dominates* v. Clearly, $G - v'$ is isomorphic to a subgraph of $G - v$. Thus, the minimum depth of a treedepth decomposition of

G rooted at v is no smaller that the minimum depth of a decomposition rooted at v'. We can use this fact in `elimination_tree`() (at any depth of recursion) by rejecting at line 22 any value of v such that there exists a lower-numbered v' such that $N_G(v') \setminus \{v\} \supseteq N_G(v) \setminus \{v'\}$.

As an example, suppose the input graph is a clique K_n, with the vertices numbered $\{1, \ldots, n\}$. At each call to `elimination_tree`(), the lowest-numbered vertex in G dominates the other vertices; thus only one vertex needs to be explored in the loop at line 22.

Our use of vertex domination is based on [6, 7], where the technique is used in a preprocessing step to generate constraints for the input graph, rather than applied to each subproblem.

Only-child vertices. Consider again our example graph in Figure 1. Suppose that the symmetry-breaking and domination rules described so far in this section are disabled, and that a decomposition of depth 3 is being sought. Figure 3(a) shows the program state after selecting vertex 6 as the root vertex of the tree and vertex 5 as its child. There are two subproblems: the subgraphs induced by $\{1, 2, 7\}$ and $\{3, 4, 8, 9\}$. We call 5 an *only child* in the tree because it has a parent (vertex 6) but has no siblings.

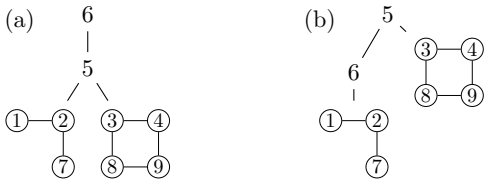


Figure 3 The only child rule allows us to avoid the effort of exploring the subproblem in the left part of the figure, since at least as good a decomposition can be achieved by choosing 5 as the root vertex.

Figure 3(b) shows the tree if vertex 5 is chosen before, rather than after, vertex 6. Observe that the same two subproblems appear, but one of them has been lifted to a higher level in the tree. It is clear that the optimal elimination tree based on Figure 3(a) will have depth no less than that of the optimal elimination tree based on Figure 3(b).

This is an example of a rule that holds in general.

▶ **Proposition 1.** *(Only-child rule) Let T be a depth-k treedepth decomposition of a connected graph G. If the root vertex of T has an only child v', then there exists a treedepth decomposition of G of depth no greater than k with v' as its root.*

Proof. Let v be the root of T, and v' its only child. Let $\{G_1, \ldots, G_a\}$ be the child subproblems that contain a vertex adjacent to v in G, and let $\{H_1, \ldots, H_b\}$ be the child subproblems that do not, as illustrated in Figure 4 (either of these sets of subproblems may be empty). For each graph in $\{G_1, \ldots, G_a\} \cup \{H_1, \ldots, H_b\}$, there must exist a decomposition of depth at most $k - 2$.

If we reverse the order of v and v', then the tree and its subproblems will be as shown in the second part of Figure 4 (where $\{H_1, \ldots, H_b\}$ are moved up a level). Clearly, it is possible to construct a decomposition with depth no greater than k by using the same decompositions of the subproblems $\{G_1, \ldots, G_a\} \cup \{H_1, \ldots, H_b\}$ as in T. ◀

This can be used for a simple domination breaking rule. If the removal of a vertex v chosen at line 22 of Algorithm 1 leaves a non-empty, connected graph, then the child vertex of v in the treedepth decomposition, v', must be an only child. The only-child rule allows us to omit any vertex v' that has a lower number than v.

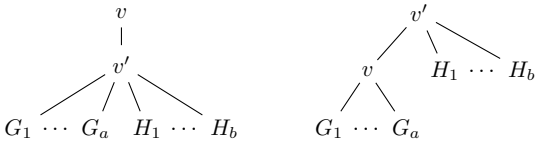


Figure 4 The general case of the only child rule. If a vertex v has an only child v' in the elimination tree, then an elimination tree of the no greater depth can be found by reversing the positions of v and v'.

4.2 Computing Lower Bounds

At lines 7 to 10 of Algorithm 1, lower bounds are computed with the goal of quickly proving that one of the subproblems is infeasible. For each connected component C, two functions are called to find lower bounds on the treedepth of C. If either of these bounds is greater than k, the current subproblem is unsatisfiable and *false* is returned. The first lower-bounding function uses a simple and very fast algorithm to compute a bound based on the number of vertices in the subproblem and an upper bound on the maximum degree. The second function greedily constructs a path in the graph, and uses as a lower bound a well-known formula for the treedepth of a path graph.

Simple lower bound. Let $b > 0$ be an upper bound on the maximum degree of a graph G. If G has no vertices, then its treedepth is zero. Otherwise, if we remove a single vertex and its incident edges from G, it is clear that the resulting graph can have at most b connected components and at least one of these components must have $\lceil (|V(G)|-1)/b \rceil$ or more vertices. Algorithm 2 is a recursive algorithm that makes use of this fact to give a lower bound on the treedepth of a graph.

Algorithm 2 The simple lower bound function.

```
1 simple_lower_bound(n)
2 begin
3     if n = 0 then return 0
4     return 1 + simple_lower_bound(⌈(n − 1)/b⌉)
```

In our implementation, b is a global variable equal to the maximum degree of the input graph. If $b = 0$, we do not use this lower bounding technique.

This bounding algorithm runs in time $O(\log n)$, where n is the argument passed to the function. As an optimisation, our implementation pre-computes `simple_lower_bound`(n) for $n \in \{0, \ldots, |V(G)|\}$, and saves these values in an array before calling `td_optimise`(), thus allowing the bounding function to run in constant time. However, we use a bitset popcount (number of set bits) operation to calculate the value of n, and therefore the overall time complexity of calculating this bound is $O(n)$, where n is the size of the input graph.

Our example graph in Figure 1 has 9 vertices and maximum degree 4. The bounding function therefore gives a lower bound of 3 on the graph's treedepth.

Path lower bound. A graph containing a path of k vertices has treedepth at least $\lceil \log_2(k+1) \rceil$ [12]. We can thus cheaply compute a lower bound on the treedepth of a graph G by greedily finding a path in G, as shown in Algorithm 3. We choose the lowest-numbered vertex v in G as our starting point, and attempt to grow the path from v in two directions (line 5). In each of these two growing phases, the algorithm extends the path by one vertex

at a time until the most-recently-visited vertex has no neighbours that are not on the path (lines 7 to 9). If $\lceil \log_2(k+1) \rceil$, where k is the length of the constructed path, exceeds the target treedepth, the algorithm returns *true*.

Algorithm 3 The path lower bound function.

```
1  can_prune_by_path_lower_bound(G, targetDepth)
2  begin
3      v ← min(V(G))
4      P ← {v}                          ▷ P is the set of vertices on the path
5      repeat 2 times
6          u ← v
7          while u has a neighbour that is not contained in P do
8              u ← the least such neighbour
9              P ← P ∪ {u}
10     return ⌈log2(|P| + 1)⌉ > targetDepth
```

As an example, consider again the graph in Figure 1. The algorithm `path_lower_bound`() begins with $v = 1$, then greedily extends the path with vertices $2, 5, 3, 4, 9$, and 8. As it is not possible to extend the path further from vertex 8, the algorithm returns to vertex 1 and prepends vertices 6 and 7 to the path. The path found is thus 7–6–1–2–5–3–4–9–8, which contains all nine of the graph's vertices and gives a lower bound of $\lceil \log_2(9+1) \rceil = 4$. (In this case, this equals the treedepth of the graph, so our optimisation algorithm is able to determine that the graph has treedepth greater than 3 without any recursive calls on subgraphs).

Since we use bitset operations to iterate over the neighbours of u at line 7 of Algorithm 3, the algorithm runs in $O(|V(G)|^2)$ time.

Our implementation has an additional small improvement to this lower bounding function. If the found path has three or more vertices, the program looks for a pair of vertices (v, w) that appear in the path, such that v and w are neighbours in G but do not appear side-by-side in the path. The section of the path from v to w thus forms a cycle, and it is possible to use the bound $1 + \lceil \log_2(k) \rceil$, where k is the number of vertices in the cycle [12]. The algorithm continues to visit all such pairs of vertices, stopping early if the calculated bound is greater than *targetDepth*. This cycle-finding extension of the bounding algorithm runs in $O(|V(G)|^2)$ time, and thus does not increase the algorithm's time complexity.

Comparison of the two bounds. Neither the simple lower bound nor the path lower bound dominates the other. We have already seen that for our example graph, the path lower bound is greater than the simple lower bound. Figure 5 is a graph for which the reverse is true. The graph has order 7 and maximum degree 3; therefore the simple lower bound is 3. The path lower-bounding function finds the path 1–5–2 which has length 3, giving a bound of 2.

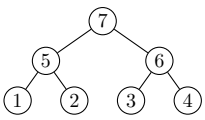


Figure 5 A graph for which the simple lower bound is greater than the path lower bound.

4.3 Initial Vertex Ordering by Degree

Before running our algorithm, the vertices of the input graph are reordered by non-increasing degree. We have observed that this leads to speed-ups on graph classes including binary trees and random graphs. We give two speculative reasons for this. First, it seems likely that for satisfiable values of the treedepth parameter k, we are most likely to find an elimination forest of depth k quickly by choosing high-degree vertices first, as these are most likely to split the remainder of the graph into small components. Second, the path-finding algorithm in our path lower bound function chooses low-numbered vertices first, and by preferring high-degree vertices it is less likely to run into "dead ends".

5 Bitset Implementation

We use bitsets to represent sets, including rows of adjacency matrices. Induced subgraphs are not stored explicitly in memory; our program simply passes a pointer to the full graph along with a pointer to the set of vertices that induce the subgraph.

The space complexity of our algorithm compares favourably to that of the partitioning-based SAT encoding, which uses $O(n^3)k$ clauses, where $n = |V(G)|$.

▶ **Proposition 2.** *Using a bitset implementation, the algorithm requires $O(n^2)$ space.*

Proof. The bit-matrix representation of the input graph G requires $O(n^2)$ space.

At most $n+1$ calls are made to `elimination_forest`(), and at most n calls are made to `elimination_tree`(), since line 25 of Algorithm 1 passes a graph with one vertex fewer than the graph passed to `elimination_tree`(). Each recursive call to these functions requires $O(n)$ space, with the exception of the connected components found at line 6. Since each connected component is stored in an n-element bitset, it remains to show that at most $O(n)$ connected components are stored at once.

We prove by induction that no more than n components are stored at one time. If `elimination_forest`() is called with a 1-vertex graph, then only one component is found, and no further components are found in recursive calls to the function. Now, let an n-vertex graph G be given, and assume that `elimination_forest`() and its recursive calls create no more than i components when called with any i-vertex graph ($1 \leq i \leq n$). Let c be the number of components created by the initial call `elimination_forest`$(G, \dots)$. Each of these components has at most $n - c + 1$ vertices, since otherwise the components would have more than n vertices in total. If a component with m vertices is passed to `elimination_tree`(), then the recursive call to `elimination_forest`() at line 25 passes a graph with at most $m - 1 \leq n - c$ vertices. By our inductive assumption, this call requires space for at most $n - c$ components, and therefore at most $c + n - c = n$ components in total are needed for the call to `elimination_forest`$(G, \dots)$. ◀

6 Experiments

This section presents an experimental comparison with the partition-based SAT encoding [6, 7] which is the existing state of the art for the exact treedepth problem. We also investigate the effect of switching off individual features of our algorithm.

The experiments were performed on a cluster of five machines with dual Intel Xeon E5-2697A v4 CPUs and 512 GBytes of RAM, running Ubuntu 18.04. We implemented our algorithm in C, using Nauty version 2.6r11 for vertex-orbit symmetry breaking. The program

for the SAT encoding[1] is written in Python and calls an external SAT solver; we made the same choice as the authors of the encoding – the sequential version of Glucose 4.0 (which is based on MiniSAT [5]). Our program and Glucose were compiled with GCC at optimisation level `-O3`. Both our algorithm and the program for SAT encoding are single-threaded. A time limit of 1000 seconds per instance was used. We verified that all solvers that solved an instance within this time limit returned the same treedepth.

Following Ganian et al. [6, 7], we used three classes of instances – famous named graphs (many of which are regular and highly symmetrical), standard graphs (binary trees, cliques, complete bipartite graphs, cycle graphs, path graphs, and square grids), and random graphs.

Table 1 Solving times in seconds for famous graphs. An asterisk indicates timeout at 1000 s.

Instance	n	m	td	All	−LB	−Sym	−Dom	SAT
Diamond	4	5	3	0.002	0.002	0.002	0.002	0.002
Bull	5	5	3	0.003	0.003	0.002	0.002	0.041
Butterfly	5	6	3	0.003	0.003	0.002	0.003	0.045
Prism	6	9	5	0.002	0.002	0.003	0.002	0.037
Moser	7	11	5	0.002	0.002	0.002	0.003	0.055
Wagner	8	12	6	0.002	0.002	0.002	0.003	0.067
Pmin	9	12	5	0.002	0.002	0.002	0.002	0.100
Petersen	10	15	6	0.002	0.002	0.002	0.002	0.129
Goldner	11	27	5	0.002	0.003	0.002	0.002	0.195
Grotzsch	11	20	7	0.003	0.003	0.003	0.002	0.187
Herschel	11	18	5	0.002	0.002	0.002	0.002	0.194
Chvatal	12	24	8	0.003	0.004	0.009	0.003	0.402
Durer	12	18	7	0.002	0.003	0.003	0.002	0.262
Franklin	12	18	7	0.002	0.002	0.004	0.003	0.273
Frucht	12	18	6	0.003	0.003	0.003	0.002	0.248
Tietze	12	18	7	0.003	0.003	0.003	0.002	0.266
Paley13	13	39	10	0.003	0.005	0.428	0.003	2.931
Poussin	15	39	9	0.005	0.009	0.101	0.005	2.478
Clebsch	16	40	10	0.005	0.020	0.974	0.004	14.147
Hoffman	16	32	8	0.003	0.010	0.013	0.003	1.500
Shrikhande	16	48	11	0.010	0.027	13.471	0.010	51.579
Sousselier	16	27	8	0.004	0.017	0.017	0.004	1.263
Errera	17	45	10	0.007	0.022	2.486	0.006	16.225
Paley17	17	68	14	0.056	0.072	*	0.052	*
Pappus	18	27	8	0.003	0.029	0.019	0.003	2.363
Robertson	19	38	10	0.018	0.365	3.441	0.021	43.926
Desargues	20	30	9	0.004	0.097	0.323	0.005	15.208
Dodecahedron	20	30	9	0.005	0.104	0.329	0.004	11.871
FlowerSnark	20	30	9	0.008	0.298	0.311	0.007	13.415
Folkman	20	40	9	0.004	0.056	0.118	0.007	10.071
Brinkmann	21	42	11	0.195	3.637	*	0.183	*
Kittell	23	63	12	0.405	3.094	*	0.559	*
McGee	24	36	11	0.219	24.042	344.762	0.175	*
Nauru	24	36	10	0.056	4.914	15.565	0.048	179.968
Holt	27	54	13	6.680	441.213	*	5.623	*
WatkinsSnark	50	75	13	*	*	*	870.345	*
B10Cage	70	105		*	*	*	*	*
Ellingham	78	117		*	*	*	*	*

Famous graphs. Table 1 shows run times in seconds for famous graphs. The second and third columns show the number of vertices and edges in each graph, and the fourth column shows the treedepth if it is known. The next four columns show run times for our algorithm;

[1] `https://github.com/nehal73/TCW_TD_to_SAT`

"All" has all features enabled, while "−LB", "−Sym", and "−Dom" have lower bounding, symmetry breaking, and domination rules turned off respectively. The final column shows run times for the partition-based SAT encoding [6, 7]. With all features on, our algorithm typically performs orders of magnitude faster than the SAT encoding. Both the symmetry breaking and lower bound features contribute to the algorithm's performance, but on this set of benchmark instances, the domination rule slows the algorithm down slightly. With the rule switched off, the algorithm closes two open instances – the Holt and Watkins Snark graphs – both of which have treedepth 13.

Standard graphs. Table 2 shows run times in seconds for standard instances. Again, our algorithm is typically much faster than the SAT encoding. The usefulness of the domination rule is demonstrated on these instances; with it switched off, the larger clique and bipartite instances could not be solved within the time limit. The lower bounding and symmetry breaking features also improve the run time on square grid graphs.

Table 2 Solving times in seconds for standard graphs. An asterisk indicates timeout at 1000 s.

Instance	n	m	td	All	−LB	−Sym	−Dom	SAT
Binary tree 10	10	9	3	0.003	0.003	0.002	0.002	0.088
Binary tree 20	20	19	4	0.002	0.003	0.002	0.002	0.804
Binary tree 30	30	29	5	0.003	0.004	0.002	0.003	4.319
Binary tree 40	40	39	5	0.002	0.008	0.002	0.002	19.021
Binary tree 50	50	49	5	0.002	0.040	0.002	0.003	52.950
Clique10	10	45	10	0.003	0.002	0.003	0.003	0.003
Clique20	20	190	20	0.003	0.003	0.002	0.879	0.006
Clique30	30	435	30	0.003	0.003	0.003	*	0.220
Clique40	40	780	40	0.003	0.004	0.003	*	0.021
Clique50	50	1225	50	0.005	0.004	0.004	*	0.035
Complete bipartite 10	10	25	6	0.003	0.003	0.003	0.002	0.132
Complete bipartite 20	20	100	11	0.002	0.003	0.003	0.012	97.689
Complete bipartite 30	30	225	16	0.003	0.004	0.008	20.394	*
Complete bipartite 40	40	400	21	0.005	0.005	0.194	*	*
Complete bipartite 50	50	625	26	0.007	0.010	7.565	*	*
Cycle 10	10	10	5	0.002	0.003	0.003	0.003	0.137
Cycle 20	20	20	6	0.002	0.004	0.002	0.003	2.194
Cycle 30	30	30	6	0.002	0.007	0.002	0.002	16.151
Cycle 40	40	40	7	0.002	0.409	0.002	0.002	67.870
Cycle 50	50	50	7	0.003	0.762	0.002	0.002	236.847
Path 10	10	9	4	0.003	0.003	0.003	0.003	0.146
Path 20	20	19	5	0.002	0.003	0.003	0.002	2.146
Path 30	30	29	5	0.002	0.010	0.003	0.003	15.688
Path 40	40	39	6	0.003	0.181	0.003	0.003	67.505
Path 50	50	49	6	0.002	0.405	0.002	0.002	251.987
Square grid 2×2	4	4	3	0.002	0.003	0.003	0.003	0.033
Square grid 3×3	9	12	5	0.003	0.003	0.003	0.002	0.100
Square grid 4×4	16	24	7	0.003	0.004	0.003	0.002	0.841
Square grid 5×5	25	40	9	0.025	2.219	0.797	0.026	17.869
Square grid 6×6	36	60	11	1.363	*	*	1.619	*

Random graphs. We generated random graphs using the Erdős-Rényi $G(n, p)$ model, with $n \in \{12, 16, 20\}$ vertices and edge probabilities $p \in \{0.1, 0.2, \ldots, 0.9\}$. Ten instances were generated for each n, p pair. Our algorithm solved each of the 270 instances in less than 0.3 seconds per instance; the SAT encoding exceeded the time limit of 1000 seconds on 53 of the 90 instances with 20 vertices.

Summary of experimental results. Figure 6 summarises, for all instances, the run times of our algorithm and the SAT encoding. Each point shows the two algorithms' run times for a single instance. Timeouts are shown as 1000 seconds. For the harder instances that take more than ten seconds to solve with the SAT encoding, our algorithm is typically around three orders of magnitude faster.

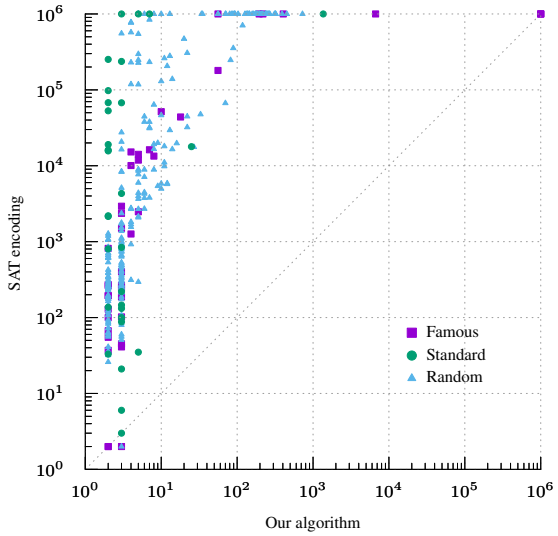


Figure 6 Run times in milliseconds. Each point represents one instance.

We end this section by noting that the SAT-encoding program writes each SAT instance it generates to disk, whereas our program performs no disk I/O while solving. For the famous graphs, the cost of this disk I/O does not meaningfully affect our comparison of run times; on the three most difficult famous instances that can be solved by the SAT-based program (Nauru, Shrikhande and Robertson), over 95% of the total run time is spent within the SAT solver on a single unsatisfiable instance of the SAT problem. For some of the standard and random graphs, such as complete bipartite graphs, a similar pattern holds. But for other graphs in these classes, such as binary trees, the SAT-based program spends most of its time creating encodings and writing them to disk, and for these instances it is likely that the program could be improved significantly by avoiding writing to disk.

7 Conclusion

We have introduced an algorithm for computing the exact treedepth of a graph, and shown experimentally that it runs orders of magnitude faster than the current state of the art on a varied set of benchmark instances. The core of the algorithm is a simple pair of mutually-recursive functions. To this basic algorithm, we have added symmetry breaking, domination, lower bounding, and vertex ordering rules. There is room for further improvement to each of these extensions of the algorithm.

There is also scope for improvement in finding a good upper bound on the treedepth quickly. SAT encodings of the problem have advantages in this regard: modern SAT solvers implement restarts and good heuristics, both of which help to find good solutions quickly. Future research could combine the benefits of SAT solvers with the techniques introduced in this paper, either by including some of the techniques in a SAT model or by adding features such as periodic restarts to our algorithm.

References

1 Bengt Aspvall and Pinar Heggernes. Finding minimum height elimination trees for interval graphs in polynomial time. *BIT Numerical Mathematics*, 34(4):484–509, December 1994. doi:10.1007/BF01934264.

2 Hans L. Bodlaender, Jitender S. Deogun, Klaus Jansen, Ton Kloks, Dieter Kratsch, Haiko Müller, and Zsolt Tuza. Rankings of graphs. *SIAM J. Discrete Math.*, 11(1):168–181, 1998. doi:10.1137/S0895480195282550.

3 Jitender S. Deogun, Ton Kloks, Dieter Kratsch, and Haiko Müller. On vertex ranking for permutations and other graphs. In Patrice Enjalbert, Ernst W. Mayr, and Klaus W. Wagner, editors, *STACS 94, 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen, France, February 24-26, 1994, Proceedings*, volume 775 of *Lecture Notes in Computer Science*, pages 747–758. Springer, 1994. doi:10.1007/3-540-57785-8_187.

4 Jitender S. Deogun, Ton Kloks, Dieter Kratsch, and Haiko Müller. On the vertex ranking problem for trapezoid, circular-arc and other graphs. *Discrete Applied Mathematics*, 98(1-2):39–63, 1999. doi:10.1016/S0166-218X(99)00179-1.

5 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.

6 Robert Ganian, Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. SAT-encodings for treecut width and treedepth. In Stephen G. Kobourov and Henning Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019.*, pages 117–129. SIAM, 2019. doi:10.1137/1.9781611975499.10.

7 Robert Ganian, Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. SAT-encodings for treecut width and treedepth. *CoRR*, abs/1911.12995, 2019. arXiv:1911.12995.

8 Chris Groër, Blair D Sullivan, and Dinesh Weerapurage. Inddgo: Integrated network decomposition & dynamic programming for graph optimization. *ORNL/TM-2012*, 176, 2012.

9 Gregory Z. Gutin, Mark Jones, and Magnus Wahlström. The mixed Chinese postman problem parameterized by pathwidth and treedepth. *SIAM J. Discrete Math.*, 30(4):2177–2205, 2016. doi:10.1137/15M1034337.

10 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60(0):94–112, 2014. doi:10.1016/j.jsc.2013.09.003.

11 Asier Mujika. About tree-depth. Bachelor's thesis, Universidad del País Vasco, 2015.

12 Jaroslav Nesetril and Patrice Ossona de Mendez. *Sparsity - Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012. doi:10.1007/978-3-642-27875-4.

13 A. Pothen. The complexity of optimal elimination trees. Technical Report CS 88-16, Department of Computer Science, Penn State, 1988.

14 Felix Reidl, Peter Rossmanith, Fernando Sánchez Villaamil, and Somnath Sikdar. A faster parameterized algorithm for treedepth. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 931–942. Springer, 2014. doi:10.1007/978-3-662-43948-7_77.

15 Colva M. Roney-Dougal, Ian P. Gent, Tom Kelsey, and Steve Linton. Tractable symmetry breaking using restricted search trees. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 211–215. IOS Press, 2004.

16 Alejandro A. Schäffer. Optimal node ranking of trees in linear time. *Inf. Process. Lett.*, 33(2):91–96, 1989. doi:10.1016/0020-0190(89)90161-0.

17 Earl Zmijewski and John R Gilbert. A parallel algorithm for large sparse Cholesky factorization on a multiprocessor. Technical Report TR 86-733, Cornell University, Ithaca, NY, 1986.

Algorithms for New Types of Fair Stable Matchings

Frances Cooper
School of Computing Science, University of Glasgow, UK
https://www.francescooper.net
f.cooper.1@research.gla.ac.uk

David Manlove
School of Computing Science, University of Glasgow, UK
http://www.dcs.gla.ac.uk/~davidm
david.manlove@glasgow.ac.uk

Abstract

We study the problem of finding "fair" stable matchings in the *Stable Marriage problem with Incomplete lists* (SMI). For an instance I of SMI there may be many stable matchings, providing significantly different outcomes for the sets of men and women. We introduce two new notions of fairness in SMI. Firstly, a *regret-equal stable matching* minimises the difference in ranks of a worst-off man and a worst-off woman, among all stable matchings. Secondly, a *min-regret sum stable matching* minimises the sum of ranks of a worst-off man and a worst-off woman, among all stable matchings. We present two new efficient algorithms to find stable matchings of these types. Firstly, the *Regret-Equal Degree Iteration Algorithm* finds a regret-equal stable matching in $O(d_0nm)$ time, where d_0 is the absolute difference in ranks between a worst-off man and a worst-off woman in the man-optimal stable matching, n is the number of men or women, and m is the total length of all preference lists. Secondly, the *Min-Regret Sum Algorithm* finds a min-regret sum stable matching in $O(d_sm)$ time, where d_s is the difference in the ranks between a worst-off man in each of the woman-optimal and man-optimal stable matchings. Experiments to compare several types of fair optimal stable matchings were conducted and show that the Regret-Equal Degree Iteration Algorithm produces matchings that are competitive with respect to other fairness objectives. On the other hand, existing types of "fair" stable matchings did not provide as close an approximation to regret-equal stable matchings.

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

Keywords and phrases Stable marriage, Algorithms, Optimality, Fair stable matchings, Regret-equality, Min-regret sum

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.20

Related Version arxiv.org/abs/2001.10875

Supplementary Material zenodo.org/record/3630383 and zenodo.org/record/3630349

Funding *Frances Cooper*: Supported by an Engineering and Physical Sciences Research Council Doctoral Training Account EP/N509668/1
David Manlove: Supported by Engineering and Physical Sciences Research Council grant EP/P028306/1

1 Introduction

1.1 Background

The Stable Marriage problem (SM) was first introduced by Gale and Shapley [5] in their seminal paper "College Admissions and the Stability of Marriage", and comprises a set of men and a set of women, where each man has a strict preference over all women and vice

versa. A *matching* in this setting is an assignment of men to women such that no man or woman is multiply assigned. A *stable matching* is then a matching in which there is no man-woman pair who would rather be assigned to each other than to their assigned partners.

In this paper we study an extension of SM, known as the Stable Marriage problem with Incomplete lists (SMI). An instance I of SMI comprises two sets of agents, men $U = \{m_1, m_2, ..., m_n\}$ and women $W = \{w_1, w_2, ..., w_n\}$. Each man (woman) ranks a subset of women (men) in strict preference order. Let m be the total length of all preference lists. A man m_i finds a woman w_j *acceptable* if w_j appears on m_i's preference list. Similarly, a woman w_j finds a man m_i *acceptable* if m_i appears on w_j's preference list. A pair (m_i, w_j) is *acceptable* if m_i finds w_j acceptable and w_j finds m_i acceptable. A *matching* M in this context is an assignment of men to women comprising acceptable pairs such that no man or woman is assigned to more than one person. Given a matching M, denote by $M(m_i)$ the woman m_i is assigned to in M (or if m_i is unassigned then $M(m_i)$ is undefined); the notation $M(w_j)$ is defined similarly for a woman w_j. A pair (m_i, w_j) is a *blocking pair* if 1) (m_i, w_j) is an acceptable pair, 2) m_i is unmatched or prefers w_j to $M(m_i)$, and 3) w_j is unmatched or prefers m_i to $M(w_j)$. Matching M is *stable* if it has no blocking pair.

In SMI, a stable matching always exists, and may be found in linear time using the Man-oriented Gale-Shapley Algorithm or the Woman-oriented Gale-Shapley Algorithm [5]. The Man-oriented Gale-Shapley Algorithm produces the *man-optimal* stable matching, that is, the unique stable matching in which each man is assigned their most-preferred woman in any stable matching. Unfortunately, the man-optimal stable matching is also *woman-pessimal* i.e., each woman is assigned their least-preferred man in any stable matching. Similarly the Woman-oriented Gale-Shapley Algorithm produces the woman-optimal (man-pessimal) stable matching.

Let I be an instance of SMI and n be the number of men or women in I. Let $\mathcal{M}$ be the set of all stable matchings in I, which may be exponential in size [10]. We note that by the *"Rural Hospitals" Theorem* [6], the same set of men and women are assigned in all stable matchings of $\mathcal{M}$. Thus in order to simplify future descriptions, we are able to use the Man-oriented Gale-Shapley Algorithm to find and remove all unassigned men and women from I prior to any other operation. Without loss of generality, we assume that from this point onwards, all men and women in I are assigned in any stable matching of I.

For an instance of SMI, it is natural to wish to find a stable matching in $\mathcal{M}$ which is in some sense fair for both sets of men and women. The *rank* of m_i with respect to M is defined as the location of $M(m_i)$ on m_i's preference list, and is denoted $\text{rank}(m_i, M(m_i))$. An analogous definition of $\text{rank}(w_j, M(w_j))$ holds for a woman w_j. We define the *man-degree* $d_U(M)$ of M as the largest rank of all men in M, that is, $d_U(M) = \max\{\text{rank}(m_i, M(m_i)) : m_i \in U\}$. Again an analogous definition of $d_W(M)$ holds for women. Define the *degree pair* of M, denoted $d(M) = (a, b)$ as the tuple of man- and woman-degrees in M, where $a = d_U(M)$ and $b = d_W(M)$. The *man-cost* $c_U(M)$ of matching M is defined as the sum of ranks of all men, that is, $c_U(M) = \sum_{m_i \in U} \text{rank}(m_i, M(m_i))$. A similar definition of $c_W(M)$ holds for women. Finally, the *degree* of a matching M is given by $d(M) = \max\{d_U(M), d_W(M)\}$ and the *cost* of matching M is given by $c(M) = c_U(M) + c_W(M)$.

We now define four notions of fairness in the SMI context. Given a stable matching M, define its *balanced score* to be $\max\{c_U(M), c_W(M)\}$. M is *balanced* [4] if it has minimum balanced score over all stable matchings in $\mathcal{M}$. Feder [4] showed that the problem of finding a balanced stable matching in SMI is NP-hard, although can be approximated within a factor of 2. This approximation factor was improved to $2 - \frac{1}{l}$, where l is the length of the longest preference list, by Eric McDermid as noted in Manlove [14, pg. 110]. Gupta et al. [7]

showed that a balanced stable matching can be found in $O(f(n)8^t)$ time when parameterised by $t = k - \min\{c_U(M_0), c_W(M_z)\}$, where $f(n)$ is a function polynomial in n and k is the balanced score. The *sex-equal score* of M is defined to be $|c_U(M) - c_W(M)|$. M is *sex-equal* [9] if it has minimum sex-equal score over all stable matchings in $\mathcal{M}$. Finding a sex-equal stable matching was shown to be NP-hard by Kato [12]. This result was later strengthened by McDermid and Irving [15] who showed that, even in the case when preference lists have length at most 3, the problem of deciding whether there is a stable matching with sex-equal score 0 is NP-complete. Additionally, a polynomial-time algorithm to find a sex-equal stable matching is described for instances in which men have preference lists of length at most 2 (women's preference lists remaining unbounded) [15]. A stable matching M is *egalitarian* [13] if $c(M)$ is minimum over all stable matchings in $\mathcal{M}$, and may be found in $O(m^{1.5})$ time [4]. Finally, a stable matching M is *minimum regret* [13] if $d(M)$ is minimum among all stable matchings in $\mathcal{M}$. It is possible to find a minimum regret stable matching in $O(m)$ time [8]. These definitions of fairness are summarised in Table 1.

Table 1 Commonly used definitions of fair stable matchings in SMI. Our contributions are labelled with an *.

	Cost	Degree
Minimising the maximum	$\min_{M\in\mathcal{M}} \max\{c_U(M), c_W(M)\}$	$\min_{M\in\mathcal{M}} \max\{d_U(M), d_W(M)\}$
	Balanced stable matching [4]	Minimum regret stable matching [13]
Minimising the absolute difference	$\min_{M\in\mathcal{M}} \lvert c_U(M) - c_W(M)\rvert$	$\min_{M\in\mathcal{M}} \lvert d_U(M) - d_W(M)\rvert$
	Sex-equal stable matching [9]	Regret-equal stable matching *
Minimising the sum	$\min_{M\in\mathcal{M}} (c_U(M) + c_W(M))$	$\min_{M\in\mathcal{M}} (d_U(M) + d_W(M))$
	Egalitarian stable matching [13]	Min-regret sum stable matching *

In Table 1 there are two new natural definitions of fairness that can be studied.

- We define the *regret-equality score* $r(M)$ as $|d_U(M) - d_W(M)|$ for a given stable matching M. M is *regret-equal* if $r(M)$ is minimum, taken over all stable matchings in $\mathcal{M}$. Note that in general we will prefer a regret-equal stable matching M such that $d_U(M) + d_W(M)$ is minimised (e.g. $d(M) = (3, 3)$ rather than $d(M) = (10, 10)$).
- We define the *regret sum* as $d_U(M) + d_W(M)$ for a given stable matching M. M is *min-regret sum* if $d_U(M) + d_W(M)$ is minimum taken over all stable matchings in $\mathcal{M}$.

1.2 Motivation

Matching algorithms are widely used in the real world to solve allocation problems based on SMI and its variants. A famous example of this is the *National Resident Matching Program* (NRMP). This scheme has been running in the US since 1952, and involves the allocation of thousands of graduating medical students to hospitals [16]. Other matching schemes involve the allocation of students to projects [1] and the allocation of kidney donors to kidney patients [2].

Let mentees take the place of men and mentors take the place of women. Thus, mentees (mentors) rank a subset of mentors (mentees) and may only be allocated one mentor (mentee) in any matching. If we used the (renamed) Mentee-Oriented Gale-Shapley Algorithm [5] to find a stable matching of mentees to mentors, then we would find a mentee-optimal stable matching M. However, as previously discussed, this would also be a mentor-pessimal stable matching. A similar but reversed situation happens using the (also renamed) Mentor-Oriented Gale-Shapley Algorithm [5]. Therefore we may wish to find a stable matching that is in some sense fair between mentees and mentors using some of the criteria described in the previous section. All the types of fair stable matchings described in Table 1 are viable candidates. However, as previously described, each of the problems of finding a balanced stable matching or a sex-equal stable matching is NP-hard, and there are existing polynomial time algorithms in the literature to find only two types of fair stable matchings, namely an egalitarian stable matching (in $O(m^{1.5})$ time) [4] and a minimum regret stable matching (in $O(m)$ time) [8]. Therefore, additional definitions of new, fair stable matchings and polynomial-time algorithms to calculate them provide additional choice for a matching scheme administrator.

Moreover, we may be interested in finding a measure that gives a worst-off mentee a partner of rank as close as possible to that of a worst-off mentor. However, from our experimental work in Section 5, we found that there was no other type of optimal stable matching that closely approximates the regret-equality score of the regret-equal stable matching. Indeed, results show that there exist regret-equal stable matchings with balanced score, cost and degree that are close to that of a balanced stable matching, an egalitarian stable matching and a minimum regret stable matching, respectively. This motivates the search for efficient algorithms to produce a regret-equal stable matching that has "good" measure relative to other types of fair stable matching.

Whilst the practical motivation for studying min-regret sum stable matchings may not be as strong as in the regret-equality case, theoretical motivation comes from completing the study of the algorithmic complexity of computing all types of fair stable matchings relative to cost and degree, as shown in Table 1.

1.3 Contribution

In this paper, we present two efficient algorithms: one to find a regret-equal stable matching, and one to find a min-regret sum stable matching, in an instance I of SMI. Let M_0 and M_z be the man-optimal and woman-optimal stable matchings in I. First we present the *Regret-Equal Degree Iteration Algorithm* (REDI), to find a regret-equal stable matching in an instance I of SMI, with time complexity $O(d_0nm)$, where $d_0 = |d_U(M_0) - d_W(M_0)|$. This is the main result of the paper. Second we present the *Min-Regret Sum Algorithm* (MRS), to find a min-regret sum stable matching in an instance I of SMI, with time complexity $O(d_sm)$, where $d_s = d_U(M_z) - d_U(M_0)$. In addition to this theoretical work, the REDI algorithm was implemented and its performance was compared against an algorithm to enumerate all stable matchings [8] (exponential in the worst case). Finally, experiments were conducted to compare six different types of optimal stable matchings (balanced, sex-equal, egalitarian, min-regret, regret-equal, min-regret sum), and output from Algorithm REDI, over a range of measures (including balanced score, sex-equal score, cost, degree, regret-equality score, regret sum). In addition to the observations already discussed in Section 1.2, we found a large variation in sex-equal scores and regret-equality scores among the six different types of optimal stable matching, and, a far smaller variation for the balanced score, cost, degree and regret sum measures. This smaller variation also includes outputs of Algorithm REDI, indicating that we are able to find a regret-equal stable matching in polynomial time with a

likely good balanced score, cost and degree using this algorithm. Indeed, we find in practice that Algorithm REDI approximates these types of optimal stable matchings at an average of 9.0%, 1.1% and 3.0% over their respective optimal values, for randomly-generated instances with $n = 1000$.

1.4 Structure of the paper

Section 2 describes a *rotation* and related concepts in SMI that will be used later in the paper. Sections 3 and 4 describe Algorithm REDI and Algorithm MRS respectively, giving in each case pseudocode, correctness proofs and time complexity calculations. An experimental evaluation is given in Section 5. Finally, future work is presented in Section 6.

2 Structure of stable matchings

For some stable matching M in an instance I of SMI, let $s(m_i, M)$ denote the next woman on m_i's preference list (starting from $M(m_i)$) who prefers m_i to $M(s(m_i, M))$ (their partner in M). A *rotation* ρ is then a sequence of man-woman pairs $\{(m_1, w_1), (m_2, w_2), ..., (m_q, w_q)\}$ in M, such that $m_{i+1} = M(s(m_i, M))$ for $1 \leq i \leq q$ where addition is taken modulo q [11]. We say rotation ρ is *exposed* on M if $\{(m_1, w_1), (m_2, w_2), ..., (m_q, w_q)\} \subseteq M$. If ρ is exposed on M, we may *eliminate* ρ on M, that is, remove all pairs of ρ from M and add pairs (m_i, w_{i+1}) for $1 \leq i \leq q$, where addition is taken modulo q, in order to produce another stable matching M' of I. The *rotation poset* $R_p(I)$ of I indicates the order in which rotations may be eliminated. Rotation ρ is said to *precede* rotation τ if τ is not exposed until ρ has been eliminated. There is a one-to-one correspondence between the set of stable matchings and the set of closed subsets of $R_p(I)$ [11, Theorem 3.1]. Gusfield and Irving [9] describe a graphical structure known as the *rotation digraph* $R_d(I)$ of I which is based on $R_p(I)$ and allows for the enumeration of all stable matchings in $O(m + n|\mathcal{M}|)$ time.

Let R be the set of rotations of I. Then $R_j(M)$ is the set of rotations that contain a women of rank j in M, that is, $R_j(M) = \{\sigma \in R : (m, w) \in \sigma \wedge \text{rank}(w, M(w)) = j)\}$. Let M_z be the woman-optimal stable matching [5]. For any stable pair $(m_i, w_j) \notin M_z$, let $\phi(m_i, w_j)$ denote the unique rotation containing pair (m_i, w_j). Finally, denote by $c(\rho)$ the closure of rotation ρ and similarly denote by $c(R')$ the closure of set of rotations R'. We say that the closure of an undefined rotation or an empty set of rotations is the empty set.

3 Algorithm to find a regret-equal stable matching in SMI

3.1 Description of the Algorithm

Algorithm REDI, which finds a regret-equal stable matching in a given instance I of SMI, is presented as Algorithm 1. For an instance I of SMI, Algorithm REDI begins with operations to find the man-optimal and woman-optimal stable matchings, M_0 and M_z, found using the Man-oriented and Women-oriented Gale-Shapley Algorithm [5]. The set of rotations R is also found using the Minimal Differences Algorithm [11].

Let $d(M_0) = (a_0, b_0)$. If $a_0 = b_0$ then we must have an optimal stable matching and so we output M_0 on Line 5. If $a_0 > b_0$ then any other matching M', where $d(M') = (a', b')$, must have $a' \geq a_0$ and $b' \leq b_0$ since any rotation (or combination of rotations) eliminated on the man-optimal matching M_0 will make men no better off and women no worse off. Therefore M_0 is optimal and so it is returned on Line 5. Now suppose $a_0 < b_0$. Throughout the algorithm we save the best matching found so far to the variable M_{opt} starting with M_0.

We know that a matching exists with $d_0 = b_0 - a_0$ and so we try to improve on this, by finding a matching M with $r(M) < d_0$.

We create several "columns" of possible degree pairs of a regret-equal matching as follows. The top-most pairs for columns $k \geq 1$ are given by the sequence

$$\big((a_0, b_0), (a_0 + 1, b_0), (a_0 + 2, b_0), ..., (\min\{n, 2b_0 - a_0 - 1\}, b_0)\big).$$

The sequence of pairs for column k ($1 \leq k \leq \min\{2d_0, n - a_0 + 1\}$) from top to bottom is given by

$$\big((a_0 + k - 1, b_0), (a_0 + k - 1, b_0 - 1), (a_0 + k - 1, b_0 - 2), ..., (a_0 + k - 1, \max\{a_0 - d_0 + k, 1\})\big).$$

At this point as long as the size n of the instance satisfies $n \geq 2b_0 - a_0 - 1$ and $a_0 - d_0 + 1 \geq 1$, the possible degree pairs of a regret-equal matching are shown in Figure 3 of [3]. We know this accounts for all possible degree pairs since, as above, if M' is any matching not equal to M_0, where $d(M') = (a', b')$, it must be that $a' \geq a_0$ and $b' \leq b_0$. Setting $b' = b_0$, the largest a' could be is given by b_0 added to the maximum possible improved difference $d_0 - 1$, that is, $a' = b_0 + d_0 - 1 = 2b_0 - a_0 - 1$. If $n < 2b_0 - a_0 - 1$ then we only consider the first $n - a_0 + 1$ columns in Figure 3 of [3]. The $a_0 - d_0 + k$ value is obtained by noting that if x is the final value of women's degree for the column sequence above then $a_0 + k - 1 - x = d_0 - 1$ and so $x = a_0 + k - d_0$. Figure 4 of [3] shows an example of the possible regret-equal degree pairs when $d(M_0) = (2, 6)$ and $n \geq 9$.

The column operation (Algorithm 2) works as follows. Let local variable M hold the current matching for this column, and let local variable Q be the set of rotations corresponding to M. Iteratively we first test if $r(M) < r(M_{opt})$ setting M_{opt} to M if so. We now check whether $d_U(M) \geq d_W(M)$. If it is, then any further rotation for this column will only make $r(M)$ larger, and so we stop iterating for this column, returning M_{opt}. Next, we find the set of rotations Q' in the closure of $R_b(M) \subseteq R$ that are not already eliminated to reach M. If eliminating these rotations would either increase the men's degree or not decrease the women's degree, then we return M_{opt}. Otherwise, set M to be the matching found when eliminating these rotations.

If after the column operation, $d_U(M_{opt}) = d_W(M_{opt})$, then we have a regret-equal matching and it is immediately returned on Lines 10 or 24 of Algorithm 1.

The column operation described above is called first from the man-optimal stable matching M_0 on Line 8, to iterate down the first column. Then for each man m_i we do the following. Let M be set to M_0. Iteratively we eliminate $(m_i, M(m_i))$ from M by eliminating rotation ρ and its predecessors (not already eliminated to reach M) such that $(m_i, M(m_i)) \in \rho$. We continue doing this until both the men's degree increases and $\text{rank}(m_i, M(m_i)) = d_U(M)$ (in the same operation). This has the effect of jumping our focus from some column of possible degree pairs, to another column further to the right with m_i being one of the lowest ranked men in M. Once we have moved to a new column we perform the column operation described above. If either m_i has the same partner in M as in M_z (hence there are no rotations left that move m_i) or $d_U(M) > d_W(M)$ (further rotations will only increase the regret-equality score), then we stop iterating for m_i. In this case we restart this process for the next man, or return M_{opt} if we have completed this process for all men. Note that since at the end of a while loop iteration, if $r(M) = 0$ then M_{opt} is returned, it is not possible for the condition $d_U(M) = d_W(M)$ to ever be satisfied in the while loop clause.

Algorithm 1 REDI(I), returns a regret-equal stable matching for an instance I of SMI.

Require: An instance I of SMI.
Ensure: Return a regret-equal stable matching M_{opt}.

$M_0 \leftarrow$ MGS(I) ▷ M_0 is the man-optimal stable matching found using the Man-oriented Gale-Shapley Algorithm (MGS) [5].
$M_z \leftarrow$ WGS(I) ▷ M_z is the woman-optimal stable matching found using the Woman-oriented Gale-Shapley Algorithm (WGS) [5].
$R \leftarrow$ MIN-DIFF(I) ▷ R is the set of rotations found using the Minimal Differences Algorithm (MIN-DIFF) [11].
if $d_U(M_0) \geq d_W(M_0)$ **then**
    return M_0
end if
$M_{opt} \leftarrow M_0$ ▷ M_{opt} is the best stable matching found so far.
$M_{opt} \leftarrow$REDI-COL($I, M_0, \emptyset, M_{opt}$) ▷ Find the best matching for the first column.
if $r(M_{opt}) = 0$ **then**
    return M_{opt}
end if
for each $m_i \in U$ **do** ▷ For each man.
    $M \leftarrow M_0$ ▷ M is the matching we start from for m_i at the beginning of each column.
    $Q \leftarrow \emptyset$ ▷ Q is the set of rotations corresponding to M.
    while $(m_i, M(m_i)) \notin M_z$ **and** $d_U(M) < d_W(M)$ **do**
        $\rho = \phi(m_i, M(m_i))$
        $a \leftarrow d_U(M)$
        $Q' \leftarrow c(\rho) \backslash Q$
        $M \leftarrow M/Q'$ ▷ Rotations in Q' are eliminated in order defined by the rotation poset of I.
        $Q \leftarrow Q \cup Q'$
        if $d_U(M) > a$ **and** rank$(m_i, M(m_i)) = d_U(M)$ **then** ▷ The men's degree has increased and m_i is a worst ranked man in M.
            $M_{opt} \leftarrow$REDI-COL(I, M, Q, M_{opt}) ▷ Find the best matching for this column.
            if $r(M_{opt}) = 0$ **then**
                return M_{opt}
            end if
        end if
    end while
end for
return M_{opt}

3.2 Correctness proof and time complexity

In this section we state the correctness and time complexity results for Algorithm REDI. The proofs of these theorems may be found in [3, Appendix A.2].

▶ **Theorem 1.** *Let I be an instance of SMI. Any matching produced by Algorithm REDI is a regret-equal stable matching of I.*

▶ **Theorem 2.** *Let I be an instance of SMI. Algorithm REDI always terminates within $O(d_0nm)$ time, where $d_0 = |d_U(M_0) - d_W(M_0)|$, n is the number of men or women in I, m is the total length of all preference lists and M_0 is the man-optimal stable matching.*

Algorithm 2 REDI-COL(I, M, Q, M_{opt}), subroutine for Algorithm 1. Column operation for the current column $d_U(M)$. Returns M_{opt}, the best stable matching found so far (according to the regret-equality score).

Require: An instance I of SMI, stable matching M, the closure of M, Q and M_{opt} the best stable matching found so far (according to the regret-equality score).

Ensure: Finds the best stable matching (according to the regret-equality score) found when incrementally eliminating women of worst rank from the current matching, without increasing the men's degree. If an improvement is made then M_{opt} is updated. M_{opt} is returned. All variables used within Algorithm 2 are understood to be local.

1: $a \leftarrow d_U(M)$
2: **while true do**
3: **if** $r(M) < r(M_{opt})$ **then**
4: $M_{opt} \leftarrow M$
5: **end if**
6: **if** $d_U(M) \geq d_W(M)$ **then** ▷ Further rotations for this column would only increase the difference in degree of men and women.
7: **return** M_{opt}
8: **end if**
9: $b \leftarrow d_W(M)$
10: $Q' \leftarrow c(R_b(M))\backslash Q$
11: **if** $d_U(M/Q') > a \vee d_W(M/Q') = b$ **then**
12: **return** M_{opt}
13: **else**
14: $M \leftarrow M/Q'$ ▷ Rotations in Q' are eliminated in order defined by the rotation poset of I.
15: $Q \leftarrow Q \cup Q'$
16: **end if**
17: **end while**

3.3 Regret-equal stable matchings with minimum cost

We may seek a regret-equal stable matching with minimum cost over all regret-equal stable matchings. This may be achieved in $O(nm^{2.5})$ time using the following process.

We define the *deletion* of pair (m_i, w_j) as the removal of w_j from m_i's preference list and the removal of m_i from w_j's preference list. *Truncating men's preference lists at* t, where $1 \leq t \leq n$, is then the process of deleting pair (m_i, w_j) for each (m_i, w_j) such that $\text{rank}(m_i, w_j) > t$. An analogous definition holds for women. For a given SMI instance I, first find the regret-equality score r of the regret-equal stable matching using Algorithm REDI in $O(d_0nm)$ time. Then, iterate over all possible man-woman degree pairs (a, b) such that $|a - b| = r$ (there are $O(n)$ such pairs). For each such degree pair (a, b), truncate men at a and women at b, creating instance I'. Then, for each of the $O(m)$ man-woman pairs (m_i, w_j) in I', fix m_i with his ath-choice partner and w_j with her bth-choice partner (where ranks are taken with respect to instance I), if possible. If this is not possible then continue to the next degree pair. Assume that w'_j is m_i's ath-choice partner, and m'_i is w_j's bth-choice partner. In I', we now delete pairs (m''_i, w''_j) for any w''_j such that m_i prefers w''_j to w'_j and w''_j prefers m_i to m''_i. Also delete the pair (m''_i, w''_j) for any m''_i such that w_j prefers m''_i to m'_i and m''_i prefers w_j to w''_j. Next we delete all remaining preference list elements of m_i except w'_j and all remaining preference list elements of w_j except m'_i. The Gale-Shapley Algorithm is run to

check that a stable matching of size n exists in I'. If no such stable matching exists then we move on to the next degree pair. Feder's Algorithm may then be used to find an egalitarian stable matching in the reduced SMI instance I' in $O(m^{1.5})$ time (using the original ranks in I as costs). This makes a total of $O(nm^{2.5})$ time to find a regret-equal stable matching with minimum cost.

4 Algorithm to find a min-regret sum stable matching in SMI

Algorithm MRS, which finds a min-regret sum stable matching, given an instance of SMI, is presented as Algorithm 3. First, the man-optimal and woman-optimal stable matchings, M_0 and M_z, are found using the Man-oriented and Women-oriented Gale-Shapley Algorithms [5]. The best matching found so far, denoted M_{opt} is initialised to M_0. We then iterate over each possible man degree a between $d_U(M_0)$ and $d_U(M_z)$ inclusive, where an improvement of M_{opt}, according to the regret sum, is still possible. As an example, suppose M_{opt} has a regret sum of 5 with $d_U(M_{opt}) = 2$ and $d_W(M_{opt}) = 3$. Then, it is not worth iterating over any man degree greater than 3 since it will not be possible to improve on the regret sum of 5 by doing so. For each iteration of the while loop, we truncate the men's preference lists at a, and find the woman-optimal stable matching M_z^T for this truncated instance. If the regret sum of M_z^T is smaller than that of M_{opt}, we update M_{opt} to M_z^T. After all iterations over possible men's degrees are completed, M_{opt} is returned.

Algorithm 3 MRS(I), returns a min-regret sum stable matching for an instance I of SMI.

Require: An instance I of SMI.
Ensure: Return a min-regret sum stable matching M_{opt}.
1: $M_0 \leftarrow$ MGS(I) ▷ M_0 is the man-optimal stable matching found using the Man-oriented Gale-Shapley Algorithm (MGS) [5].
2: $M_z \leftarrow$ WGS(I) ▷ M_z is the woman-optimal stable matching found using the Woman-oriented Gale-Shapley Algorithm (WGS) [5].
3: $M_{opt} \leftarrow M_0$
4: $a \leftarrow d_U(M_0)$
5: **while** $a \leq d_U(M_z)$ **and** $a + 1 < d_U(M_{opt}) + d_W(M_{opt})$ **do**
6: $I_T \leftarrow$instance I where men's preference lists are truncated at rank a.
7: $M_z^T \leftarrow$ WGS(I_T)
8: **if** $d_U(M_z^T) + d_W(M_z^T) < d_U(M_{opt}) + d_W(M_{opt})$ **then**
9: $M_{opt} \leftarrow M_z^T$
10: **end if**
11: $a \leftarrow a + 1$
12: **end while**
13: **return** M_{opt}

Let d_s denote the difference between the degree of men in the woman-optimal stable matching M_z, and in the man-optimal stable matching M_0, that is $d_s = d_U(M_z) - d_U(M_0)$. Theorem 3 as follows states that Algorithm MRS produces a min-regret sum stable matching in $O(d_s m)$ time. See [3, Appendix B] for the proof of this Theorem.

▶ **Theorem 3.** *Let I be an instance of SMI. Algorithm MRS produces a min-regret sum stable matching in $O(d_s m)$ time, where $d_s = d_U(M_z) - d_U(M_0)$, m is the total length of all preference lists, and M_0 and M_z are the man-optimal and woman-optimal stable matchings respectively.*

5 Experiments

5.1 Methodology

An Enumeration Algorithm (ENUM) exists to find the set of all stable matchings of an instance I of SMI in $O(m + n|\mathcal{M}|)$ time [8]. Within this time complexity, it is possible to output a regret-equal stable matching from this set of stable matchings, by keeping track of the best stable matching found so far (according to the regret-equality score) as they are created. We randomly generated instances of SM, in order to compare the performance of Algorithms REDI and ENUM. Using output from Algorithm ENUM, we also investigated the effect of varying instance sizes, for six different types of optimal stable matchings (balanced, sex-equal, egalitarian, min-regret, regret-equal, min-regret sum), and also output from Algorithm REDI, over a range of measures (including balanced score, sex-equal score, cost, degree, regret-equality score, regret sum). Tests were run over 19 different instance types with varying instance size ($n \in \{10, 20, ..., 100, 200, ..., 1000\}$). All instances tested were complete with uniform distributions on preference lists. Experiments were run over 500 instances of each instance type.

Each instance was run over the two algorithms described above with a timeout time of 1 hour for each algorithm. No instances timed out for these experiments. Experiments were conducted on a machine running Ubuntu version 18.04 with 32 cores, 8×64GB RAM and Dual Intel® Xeon® CPU E5-2697A v4 processors. Instance generation, correctness and statistics summarisation programs, and plot and LaTeX table generation were all written in Python and run on Python version 3.6.1. All other code was written in Java and compiled using Java version 1.8.0. Each instance was run on a single thread with 16 instances run in parallel using GNU Parallel [17]. Serial Java garbage collection was used with a maximum heap size of 2GB distributed to each thread. Code and data repositories for these experiments can be found at zenodo.org/record/3630383 and zenodo.org/record/3630349 respectively. Comprehensive correctness testing was conducted, a description of which may be seen in [3, Appendix C.1].

5.2 Experimental results summary

Figure 1 shows a comparison of the time taken to execute the two algorithms over increasing values of n. Precise data for this plot can be seen in Table 2 of [3, Appendix C.2] which gives the mean, median, 5th percentile and 95th percentile durations for Algorithms REDI and ENUM. In Figure 1, the median values of time taken for each algorithm are plotted and a 90% confidence interval is displayed using the 5th and 95th percentile measurements. Additional experiments and evaluations not discussed here may also be found in [3, Appendix C.3].

Figure 2 shows comparisons of six different types of optimal stable matchings (balanced, sex-equal, egalitarian, min-regret, regret-equal, min-regret sum), and output from Algorithm REDI, over a range of measures (including balanced score, sex-equal score, cost, degree, regret-equality score, regret sum), as n increases. Optimal stable matching statistics involving a measure determined by cost (respectively degree) are given a green (respectively blue) colour. For a particular fairness objective A and a particular fairness measure B, there may be a set of several stable matchings that are optimal with respect to A. In this case we choose a matching from this set that has best possible measure with respect to B. For example, if we are looking at the regret-equality score, for a particular instance, we find a sex-equal stable matching that has smallest regret-equality score (over the set of all sex-equal stable

matchings) and use this value to plot the regret-equality score for this type of optimal stable matching. This process is replicated for the other types of optimal stable matching. In each case the mean measure value is plotted for the given type of optimal stable matching. Data for these plots may be found in Tables 3, 4, 5, 6, 7 and 8 of [3, Appendix C.2].

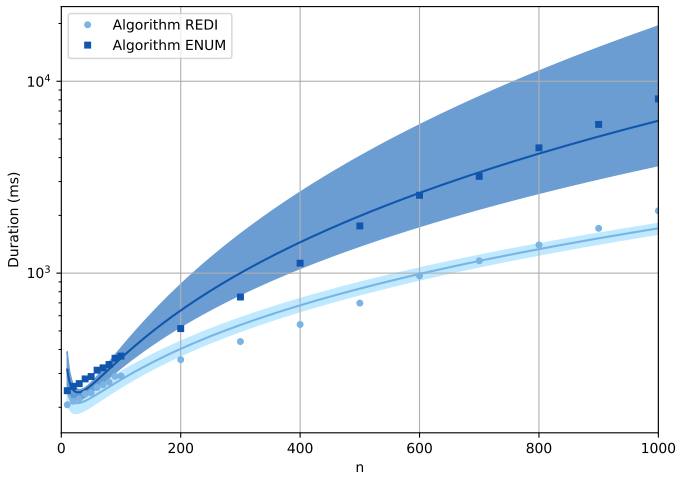
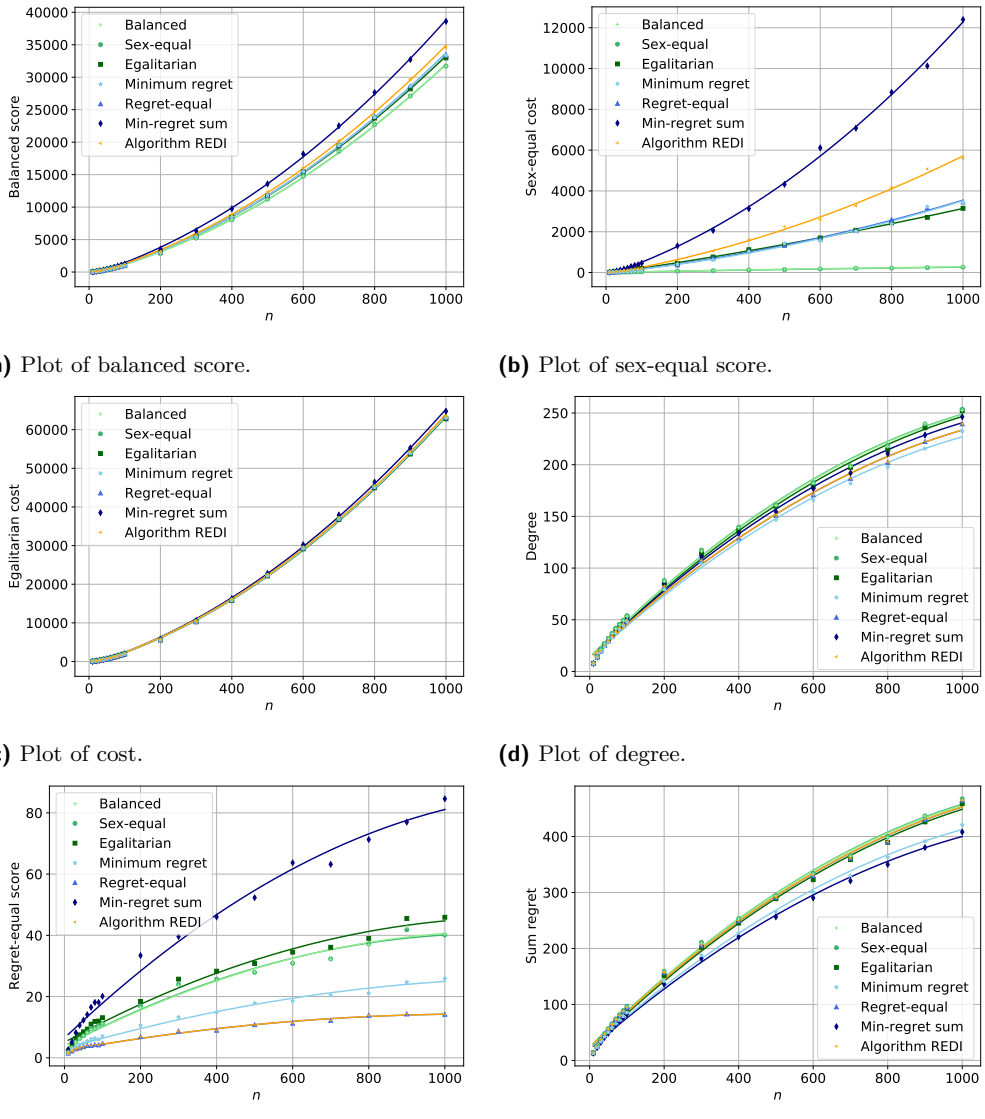


Figure 1 A log plot of the time taken to execute Algorithms REDI and ENUM. A second order polynomial model has been assumed for best-fit lines.

The main results of these experiments are:

- *Time taken:* It is clear from Figure 1 in that Algorithm REDI is the faster algorithm in practice, taking approximately 2s to solve an instance of size $n = 1000$ with very little variation. In contrast, Algorithm ENUM takes around 8s for an instance of size $n = 1000$ with a far larger variation.
- *Sex-equal score:* A wide variation in sex-equal score over the six optimal matchings can be seen in Figure 2b (and Table 4 in [3]). Sex-equal and balanced stable matchings are extremely closely aligned giving a mean sex-equal score of 265.0 and 284.0 respectively for the instance type with $n = 1000$. Min-sum regret stable matchings, on the other hand, performed the least well with a mean sex-equal score of 12400.0 for the same instance type.
- *Regret-equality score:* Similar to the previous point we see a wide variation in regret-equality score over the six optimal stable matchings in Figure 2e (and Table 7 in [3]). For the instance type with $n = 1000$, this ranges from a mean regret-equality score of 14.2 for the regret-equal stable matching to 84.6 for the minimum regret stable matching. It is interesting to note that the type of optimal stable matching (out of the six optimal stable matchings tested) whose regret-equality score tends to be furthest away from that of a regret-equal stable matching is the min-regret sum stable matching. This may be due to the fact that minimising the sum of two measures does not necessarily force the two measures to be close together.
- *Output from Algorithm REDI:* Due to the wide variation of regret-equality scores among different types of optimal stable matchings (as described above) it is clear that no other optimal stable matching is able to closely approximate a regret-equal stable matching, which highlights the importance of Algorithm REDI that is designed specifically for optimising this measure. Interestingly, Algorithm REDI is also competitive in terms of balanced score, cost and degree. Indeed, we can see from Tables 3, 5 and 6 in [3], that

Algorithm REDI approximates these types of optimal stable matchings at an average of 9.0%, 1.1% and 3.0% over their respective optimal values, for instances with $n = 1000$. Over all instance sizes, these values are within ranges $[4.0\%, 10.9\%]$, $[1.1\%, 3.4\%]$ and $[1.3\%, 3.7\%]$, respectively. This gives a good indication of the high-quality of output from this algorithm even on seemingly unrelated measures.



(a) Plot of balanced score.

(b) Plot of sex-equal score.

(c) Plot of cost.

(d) Plot of degree.

(e) Plot of regret-equality score.

(f) Plot of regret sum.

Figure 2 Plots of experiments to compare six different optimal stable matchings (balanced, sex-equal, egalitarian, min-regret, regret-equal, min-regret sum), and output from Algorithm REDI, over a range of measures (including balanced score, sex-equal score, cost, degree, regret-equality score, regret sum). A second order polynomial model has been assumed for all best-fit lines.

6 Future work

We introduced two new notions of fair stable matchings for SMI, namely, the regret-equal stable matching and the min-regret sum stable matching. We presented algorithms that are able to compute matchings of these types in polynomial time: $O(d_0nm)$ time for the regret-equal stable matching, where $d_0 = |d_U(M_0) - d_W(M_0)|$; and $O(d_sm)$ time for the min-regret sum stable matching, where $d_s = d_U(M_z) - d_U(M_0)$. It remains open as to whether these time complexities can be improved.

References

1 D.J. Abraham, R.W. Irving, and D.F. Manlove. Two algorithms for the Student-Project allocation problem. *Journal of Discrete Algorithms*, 5(1):79–91, 2007.

2 P. Biró, J. van de Klundert, D. Manlove, W. Pettersson, T. Andersson, L. Burnapp, P. Chromy, P. Delgado, P. Dworczak, B. Haase, A. Hemke, R. Johnson, X. Klimentova, D. Kuypers, A. Nanni Costa, B. Smeulders, F. Spieksma, M.O. Valentín, and A. Viana. Modelling and optimisation in european kidney exchange programmes. *European Journal of Operational Research*, 13(4):1–10, 2019.

3 F. Cooper and D.F. Manlove. Algorithms for new types of fair stable matchings. Technical Report 2001.10875, Computing Research Repository, Cornell University Library, 2020. Available from `https://arxiv.org/abs/2001.10875`.

4 T. Feder. *Stable Networks and Product Graphs*. PhD thesis, Stanford University, 1990. Published in *Memoirs of the American Mathematical Society*, vol. 116, no. 555, 1995.

5 D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–15, 1962.

6 D. Gale and M. Sotomayor. Some remarks on the stable matching problem. *Discrete Applied Mathematics*, 11:223–232, 1985.

7 S. Gupta, S. Roy, S. Saurabh, and M. Zehavi. Balanced stable marriage: How close is close enough? In *Proceedings of WADS '19: the 16th Algorithms and Data Structures Symposium*, Lecture Notes in Computer Science, pages 423–437. Springer, 2019.

8 D. Gusfield. Three fast algorithms for four problems in stable marriage. *SIAM Journal on Computing*, 16(1):111–128, 1987.

9 D. Gusfield and R.W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 1989.

10 R.W. Irving and P. Leather. The complexity of counting stable marriages. *SIAM Journal on Computing*, 15(3):655–667, 1986.

11 R.W. Irving, P. Leather, and D. Gusfield. An efficient algorithm for the "optimal" stable marriage. *Journal of the ACM*, 34(3):532–543, 1987.

12 A. Kato. Complexity of the sex-equal stable marriage problem. *Japan Journal of Industrial and Applied Mathematics*, 10:1–19, 1993.

13 D.E. Knuth. *Mariages Stables*. Les Presses de L'Université de Montréal, 1976. English translation in *Stable Marriage and its Relation to Other Combinatorial Problems*, volume 10 of CRM Proceedings and Lecture Notes, American Mathematical Society, 1997.

14 D.F. Manlove. *Algorithmics of Matching Under Preferences*. World Scientific, 2013.

15 E. McDermid and R.W. Irving. Sex-equal stable matchings: Complexity and exact algorithms. *Algorithmica*, 68:545–570, 2014.

16 E. Peranson and R.R. Randlett. The NRMP matching algorithm revisited: Theory versus practice. *Academic Medicine*, 70(6):477–484, 1995.

17 O. Tange. GNU parallel - the command-line power tool. *The USENIX Magazine*, pages 42–47, 2011.

Crystal Structure Prediction via Oblivious Local Search

Dmytro Antypov
Department of Chemistry, University of Liverpool, UK
Leverhulme Research Centre for Functional Materials Design, University of Liverpool, UK
D.Antypov@liverpool.ac.uk

Argyrios Deligkas
Department of Computer Science, Royal Holloway University of London, UK
Argyrios.Deligkas@rhul.ac.uk

Vladimir Gusev
Leverhulme Research Centre for Functional Materials Design, University of Liverpool, UK
Department of Chemistry, University of Liverpool, UK
Vladimir.Gusev@liverpool.ac.uk

Matthew J. Rosseinsky
Department of Chemistry, University of Liverpool, UK
M.J.Rosseinsky@liverpool.ac.uk

Paul G. Spirakis
Department of Computer Science, University of Liverpool, UK
Computer Engineering and Informatics Department, University of Patras, Greece
P.Spirakis@liverpool.ac.uk

Michail Theofilatos
Leverhulme Research Centre for Functional Materials Design, University of Liverpool, UK
Department of Computer Science, University of Liverpool, UK
Michail.Theofilatos@liverpool.ac.uk

Abstract

We study Crystal Structure Prediction, one of the major problems in computational chemistry. This is essentially a continuous optimization problem, where many different, simple and sophisticated, methods have been proposed and applied. The simple searching techniques are easy to understand, usually easy to implement, but they can be slow in practice. On the other hand, the more sophisticated approaches perform well in general, however almost all of them have a large number of parameters that require fine tuning and, in the majority of the cases, chemical expertise is needed in order to properly set them up. In addition, due to the chemical expertise involved in the parameter-tuning, these approaches can be *biased* towards previously-known crystal structures. Our contribution is twofold. Firstly, we formalize the Crystal Structure Prediction problem, alongside several other intermediate problems, from a theoretical computer science perspective. Secondly, we propose an oblivious algorithm for Crystal Structure Prediction that is based on local search. Oblivious means that our algorithm requires minimal knowledge about the composition we are trying to compute a crystal structure for. In addition, our algorithm can be used as an intermediate step by *any* method. Our experiments show that our algorithms outperform the standard basin hopping, a well studied algorithm for the problem.

2012 ACM Subject Classification Applied computing → Chemistry

Keywords and phrases crystal structure prediction, local search, combinatorial neighborhood

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.21

Related Version `https://arxiv.org/pdf/2003.12442.pdf`

Supplementary Material The source code of this project can be found in `https://bit.ly/2w4zG1L`.

Funding The authors were supported by the Leverhulme Trust.

18th International Symposium on Experimental Algorithms (SEA 2020).
Editors: Simone Faro and Domenico Cantone; Article No. 21; pp. 21:1–21:14

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

1 Introduction

The discovery of new materials has historically been made by experimental investigation guided by chemical understanding. This approach can be both time consuming and challenging because of the large space to be explored. For example, a "traditional" method for discovering inorganic solid structures relies on knowledge of material chemistry coupled with repeating synthesis experiments and systematically varying elemental ratios, each of which can take lots of time [25, 26]. As a result there is a very large unexplored space of chemical systems: only 72% of binary systems, 16% of ternary, and just 0.6% of quaternary systems have been studied experimentally [27].

These inefficiencies forced physical scientists to develop computational approaches in order to tackle the problem of finding new materials. The first approach is based on data mining where *only* pre-existing knowledge is used [7, 11, 13, 15, 23]. Although this approach has proven to be successful, there is the underlying risk of missing best-in-class materials by being biased towards *known* crystal structures. Hence, the second approach tries to fill this gap and aims at finding new materials with *little, or no,* pre-existing knowledge, by *predicting the crystal structure* of the material. This approach has led to the discovery of several new, counterintuitive, materials whose existence could not be deduced by the structures of previously-known materials [6].

Several heuristic methods have been suggested for crystal structure prediction. All these methods are based on the same fundamental principle. Every arrangement of ions in the 3-dimensional Euclidean space corresponds to an energy value and it defines a point on the *potential energy surface*. Then, the crystal structure prediction problem is formulated as a *mathematical optimization* problem where the goal is to compute the structure that corresponds to the global minimum of the potential energy surface, since this is the most likely structure that corresponds to a stable material. The difficulties in solving this optimization problem is that the potential energy surface is *highly non convex*, with *exponentially many*, with respect to the number of ions, local minima [17]. For this reason, several different algorithmic techniques were proposed ranging from simple techniques, like *quasi-random sampling* [9, 21, 20, 22], *basin hopping* [12, 28], and *simulated annealing* techniques [18, 24], to more sophisticated techniques, like *evolutionary and genetic algorithms* [4, 8, 14, 16, 30], and *tiling* approaches [6, 5]. A recent comprehensive review on these techniques can be found in [17].

The simple searching techniques are easy to understand, usually easy to implement, and they are *unbiased*, but they can be slow in practice. On the other hand, the more sophisticated approaches perform well in general, however almost all of them have a large number of parameters that require fine tuning and, in the majority of the cases, chemical expertise is needed in order to properly set them up. In addition, due to the chemical expertise involved in the parameter-tuning, these approaches can be *biased* towards previously-known structures.

The majority of the abovementioned heuristic techniques work, at a very high level, in a similar way. Given a current solution x for the crystal structure prediction problem, i.e., a location for every ion in the 3-dimensional space, they iteratively perform the following three steps.

1. Choose a new potential solution x'. This can be done by taking into account, or modifying, x.
2. Perform gradient descent on the potential energy surface starting from x', until a local minimum is found. This process is called *relaxation* of x'.
3. Decide whether to keep x as the candidate solution or to update it to the solution found after relaxing x'.

For example, basin hopping algorithms randomly choose x', they relax x' and if the energy of the relaxed structure is lower than the x, or a Metropolis criterion is satisfied, they accept this as a current solution; else they keep x and they randomly choose x''. The procedure usually stops when the algorithm fails to find a structure with lower energy within a predefined number of iterations. The more sophisticated algorithms take into account knowledge harvested from chemists and put constraints on the way x' is selected. For example, the MC-EMMA [6] and the FUSE [5] algorithms use a set of building blocks to construct x'. These building blocks are local configurations of ions that are present in, or similar to, known crystal structures. These approaches restrict the search space, which accelerate search, but reduce the number of possible solutions.

This general algorithm is easy to understand, however there are some hidden difficulties that make the problem more challenging. Firstly, it is not trivial even how to *evaluate* the potential energy of a structure. There are several different methods for calculating the energy of a structure, ranging from *quantum mechanical* methods, like *density functional theory* [1], to *force fields* methods [2], like the *Buckingham-Coulomb* potential function. All of which though, are hard to compute (see Section 2.1) from the point of view of (theoretical) computer science and thus only numerical methods are known and used in practice for them [10]; still there are cases where some methods need considerable time to calculate the energy of a structure. This yields another, more important, difficulty, the relaxation of a structure. Since it is hard to compute the energy of a structure, it is even harder to apply gradient descent on the potential energy surface. For these reasons, the majority of the heuristic algorithms depend on *external*, well established, codes [10] for computing the aforementioned quantities. Put differently, both energy computations and relaxations of structures are treated as *oracles* or *black boxes*.

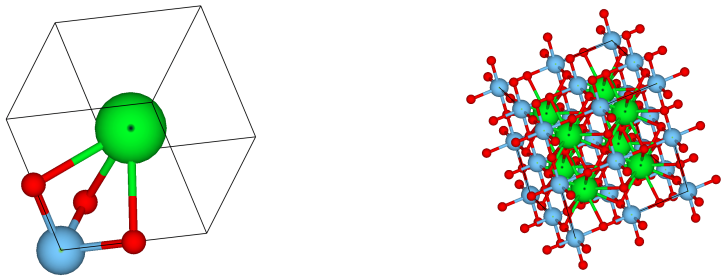
1.1 Our contribution

Our contribution is twofold. Firstly, we formalize the Crystal Structure Prediction problem from the theoretical computer science perspective; to the best of our knowledge, this is among the few papers that attempt to connect computational chemistry and computer science. En route to this, we introduce several intermediate open problems from computational chemistry in CS terms. Any (partial) positive solution to these questions can significantly help computational chemists to identify new materials. On the other hand, any negative result can formally explain why the discovery of new materials is a notoriously difficult task.

Our second contribution is the partial answer for some of the questions we cast. In general, our goal is to create *oblivious* algorithms that are easy to implement, they are fast, and they work well in practice. With oblivious we mean that we are seeking for *general procedures* that require *minimal input* and they have zero, or just a few, parameters chosen by the user.

- We propose a purely combinatorial method for estimating the energy of a structure, which we term *depth energy computation*. We choose to compare our method against GULP [10], which is considered to be the state of the art for computing the energy of a structure and for performing relaxations when the Buckingham-Coulomb energy is used. Our method requires only the charges of the atoms and their corresponding Buckingham coefficients to work; see Eq. 2 in Section 2.1. In addition, it needs only one parameter, the

[1] `https://en.wikipedia.org/wiki/Density_functional_theory`
[2] `https://en.wikipedia.org/wiki/Force_field_(chemistry)`



(a) Unit cell. **(b)** Supercell.

Figure 1 Most stable configuration of $SrTiO_3$.

depth k. We experimentally demonstrate that our method monotonically approximates with respect to k the energy computed by GULP and that it achieves an error of 0.0032 for $k = 6$. Our experiments show that the structure that achieves the minimum energy in depth 1 is likely to be the structure with the minimum energy overall. In fact we show something much stronger. If the energy of x is lower than the energy of x' when it is computed via the depth energy computation for $k = 1$, then, almost always, the energy of x will be lower than the energy of x' when it is computed via GULP.

- We derive oblivious algorithms for choosing which structure to relax next. All of our algorithms are based on local search. More formally, starting with x and using only local changes we select x'. We define several "combinatorial neighborhoods" and we evaluate their efficiency. Our neighborhoods are oblivious since they only need access to an oracle that calculates the energy of a structure. We show that our method outperforms basin hopping. Moreover, we view our algorithms as an intermediate step before relaxation that can be applied to *any* existing algorithm.

2 Preliminaries

A *crystal* is a solid material whose atoms are arranged in a highly ordered configuration, forming a *crystal structure* that extends in all directions. A crystal structure is characterized by its *unit cell*; a parallelepiped that contains atoms in a specific arrangement. The unit cell is the *period* of the crystal; unit cells are stacked in the three dimensional space to form the crystal. In this paper we focus on *ionically bonded crystals*, which we describe next; what follows is relevant only on crystals of this type. In order to fully define the unit cell of a ionically bonded crystal structure, we have to specify a *composition*, *unit cell parameters*, and an *arrangement* of the ions.

Composition. A composition is the chemical formula that describes the ratio of ions that belong to the unit cell. The chemical formula contains *anions*, negatively charged ions, and *cations*, positively charged ions. The chemical formula is a way of presenting information about the *chemical proportions* of ions that constitute a particular chemical compound, and it does not provide any information about the exact number of atoms in the unit cell. More formally, the composition is defined by a set of distinct chemical elements $\{e_1, e_2, \ldots, e_m\}$, their multiplicity n_i, and a non-zero integer charge q_i for each element i. The number m denotes the total number of distinct chemical elements, and $n_i / \sum_{j=1}^{m} n_j$ is the proportion of the atoms of type e_i in the unit cell. It is required that the sum of the charges adds up to zero, i.e. , $\sum_{i=1}^{m} q_i n_i = 0$, so that the unit cell is charge neutral. For example, the

composition for Strontium Titanate, $SrTiO_3$, denotes that the following hold. For every ion of strontium (Sr) in the unit cell, there exists one ion of titanium (Ti) and three ions of oxygen (O). Furthermore, the charge of every ion of Sr is $+2$, of Ti is $+4$, and of O is -2. Hence, when the ratios of the ions are according to the composition, the charge of the unit cell is zero.

Another parameter for every atom is the *atomic radius*. This usually corresponds to the distance from the center of the nucleus to the boundary of the surrounding shells of electrons. Since the boundary is not a well-defined physical entity, there are various non-equivalent definitions of atomic radius. In crystal structures though, the *ionic radius* is used and usually is treated as a hard sphere. Thus, we will use ρ_i to denote the ionic radius of the element e_i.

Unit cell parameters. Unit cell parameters provide a formal description of the parallelepiped that represents the unit cell. These include the lengths y_1, y_2, y_3 of the parallelepiped in every dimension and the angles θ_{12}, θ_{13}, and θ_{23} between the corresponding facets. For brevity, we denote $y = (y_1, y_2, y_3)$ and $\theta = (\theta_{12}, \theta_{13}, \theta_{23})$, and we use (y, θ) to denote the unit cell parameters.

Arrangement. An arrangement describes the position of each atom of the composition in the unit cell. The position of ion i is specified by a point $x_i = (x_{i1}, x_{i2}, x_{i3})$ in the parallelepiped defined by the unit cell parameters; fractional coordinates x_i denote the location of the nucleus of the ion i in the unit cell. A unit cell parameters-arrangement combination (y, θ, x) in a unit cell with n ions is a point in the $3n + 6$-dimensional space. For any two points x_i and x_j we will use $\mathtt{d}(x_i, x_j)$ to denote their Euclidean distance.

As we have already said, a unit cell parameters-arrangement configuration (y, θ, x) defines the period of an infinite structure that covers the whole 3d space. To get some intuition, assume that we have an orthogonal unit cell, i.e., all the angles are 90 degrees. Then for every ion with position (x_{i1}, x_{i2}, x_{i3}) in the unit cell, there exist "copies" of the ion in the positions $(k_1 \cdot y_1 + x_{i1}, k_2 \cdot y_2 + x_{i2}, k_3 \cdot y_3 + x_{i3})$ for every possible combination of integers k_1, k_2, and k_3. A unit cell parameters-arrangement configuration is *feasible* if the hard spheres of any two ions of the crystal structure do not overlap; formally, it is feasible if for every two ions i and j it holds that $\mathtt{d}(x_i, x_j) \geq \rho_i + \rho_j$.

2.1 Energy

Any unit cell parameters-arrangement configuration of a composition corresponds to a *potential energy*. When the number of ions in the unit cell is fixed, the set of configurations define the *potential energy surface*.

Buckingham-Coulomb potential is among the most well adopted methods for computing energy [3, 29] and it is the sum of the Buckingham potential and the Coulomb potential. The Coulomb potential is *long-range* and depends only on the charges and the distance between the ions; for a pair of ions i and j, the Coulomb energy is defined by

$$CE(i, j) := \frac{q_i q_j}{\mathtt{d}(x_i, x_j)}. \tag{1}$$

Note, ions i and j can be in *different* unit cells.

The Buckingham potential is *short-range* and depends on the species of the ions and their distance. More formally, it depends on positive composition-dependent constants

A_{e_i,e_j}, B_{e_i,e_j}, and C_{e_i,e_j} for every pair of species e_i and e_j; here i can be equal to j [3]. So, for the pair of ions i, of specie e_i, and j, of specie e_j, the Buckingham energy is

$$BE(i,j) := A_{e_i,e_j} \cdot \exp(-B_{e_i,e_j} \cdot \mathtt{d}(x_i,x_j)) - \frac{C_{e_i,e_j}}{\mathtt{d}(x_i,x_j)^6}. \tag{2}$$

Again, ions i and j can be in different unit cells.

Let $S(x_i,\rho)$ denote the sphere with centre x_i and radius ρ. The total energy of a crystal structure whose unit cell is characterized by n ions with arrangement $x = (x_1,\ldots,x_n)$ is then defined

$$E(y,\theta,x) = \lim_{\rho\to\infty} \sum_{i=1}^{n} \sum_{j\neq i, j\in S(x_i,\rho)} \left(BE(i,j) + CE(i,j)\right).$$

$E(y,\theta,x)$ *conditionally converges* to a certain value [19] and usually numerical approaches are used to compute it. For this reason, and since we aim for an oblivious algorithm, we view the computation of the energy of a structure as a *black box*. More specifically, we assume that we have an oracle that given any structure (y,θ,x), it returns its corresponding energy.

▶ **Open Question 1.** *Given a composition and Buckingham parameters for it, find a simple, purely combinatorial way that approximates the energy for every crystal structure.*

▶ **Open Question 2.** *Given a composition $\{e_1,e_2,\ldots,e_m\}$ and an oracle that computes the energy of every structure for this composition, learn efficiently (with respect to the number of oracle calls) the Buckingham parameters A_{e_i,e_j}, B_{e_i,e_j}, and C_{e_i,e_j} for every $i,j\in[m]$.*

Relaxation. The *relaxation* of a crystal structure (y,θ,x) computes a stationary point on the potential energy surface by applying gradient descent starting from (y,θ,x). The relaxation of a structure can change *both* the arrangement x of the ions in the unit cell *and* the unit cell parameters (y,θ) of the unit cell. We follow a similar approach as we did with the energy and we assume that there is an oracle that given a crystal structure (y,θ,x) it returns the relaxed structure.

▶ **Open Question 3.** *Find an alternative, quicker, way to compute an approximate local minimum when:*

a) *the unit cell parameters (y,θ) of the unit cell are fixed;*

b) *the arrangement x of the ions is fixed;*

c) *both unit cell parameters and arrangement are free.*

2.2 Crystal Structure Prediction problems

In crystal structure prediction problems the general goal is to minimize the energy in the unit cell. There are two kinds of problems we are concerned. The first cares only about the value of the energy and the second one cares for the arrangement and the unit cell parameters that achieve the minimum energy. From the computational chemistry point of view, both questions are interesting in their own right. The existence of a unit cell parameters-arrangement that

[3] The Buckingham constants are composition-depended since they can have small discrepancies in different compositions. For example the constants $A_{\mathrm{Ti,O}}, B_{\mathrm{Ti,O}}$, and $C_{\mathrm{Ti,O}}$ for $SrTiO_3$ can be different than those for $MgTiO_3$. There is a long line of research in computational chemistry that tries to learn/estimate the Buckingham constants for various compositions. In addition, more than one set of Buckingham constants can be available for a given composition.

achieves lower-than-currently-known energy usually suffices for constructing a new material. On the other hand, identifying the arrangement and the unit cell parameters of a crystal structure that achieves the lowest possible energy can help physical scientists to predict the properties of the material.

MinEnergy

Input: A composition with its corresponding Buckingham constants, a positive integer n, and a rational $\hat{E}$.
Question: Is there a crystal structure (y, θ, x) for the composition with n ions that is neutrally charged and achieves Buckingham-Coulomb energy $E(y, \theta, x) < \hat{E}$?

MinStructure

Input: A composition with its corresponding Buckingham constants, a positive integer n.
Task: Find a crystal structure (y, θ, x) for the composition with n ions that is neutrally charged and the Buckingham-Coulomb energy $E(y, \theta, x)$ is minimized.

The second class of problems, the ones that ultimately computational chemists would like to solve, take as input only the composition and the goal is to construct a unit cell, with any number of atoms, such that the *average energy per ion* is minimized.

AvgEnergy

Input: A composition with its corresponding Buckingham constants and a rational $\hat{E}$.
Question: Is there a crystal structure for the composition that is neutrally charged and $\frac{E(y,\theta,x)}{n} < \hat{E}$?

AvgStructure

Input: A composition with its corresponding Buckingham constants.
Task: Find a crystal structure for the composition that is neutrally charged and the average Buckingham-Coulomb energy per ion in the unit cell, $\frac{E(y,\theta,x)}{n}$, is minimized.

Although the problems are considered to be intractable [17], only recently the first *correct* NP-hardness result was proven for a variant of CSP [1]. However, for the problems presented, there are no correct NP-hardness results in the literature.

▶ **Open Question 4.** *Provide provable lower bounds and upper bounds for the four problems defined above.*

▶ **Open Question 5.** *Construct a heuristic algorithm that works well in practice.*

3 Local Search

Local search algorithms start from a feasible solution and iteratively obtain better solutions. The key concept for the success of such algorithms, is given a feasible solution, to be able to *efficiently find* an improved one. Put formally, a local search algorithm is defined by a *neighbourhood function* N and a *local rule* r. In every iteration, the algorithm does the following.

- Has the current best solution x.
- Computes the neighbourhood $N(x)$.
- If there is an improved solution x' in $N(x)$, then it updates x according to the rule r, i.e. $x' = r(N(x))$; else it terminates and outputs x.

The neighborhood $N(x)$ of a solution x consists of all feasible solutions that are "close" in some sense to x. The size of the neighborhood can be constant or a function of the input. In principle, the larger the size of the neighborhood, the better the quality of the locally optimal solutions. However, the downside of choosing large neighborhoods is that, in general, it makes each iteration computationally more expensive. Running time and quality of solutions are competing considerations, and the trade off between them can be determined through experimentation.

We study the following *combinatorial* neighborhoods for Crystal Structure Prediction. All of them keep the unit cell parameters fixed and change only the arrangement x of the n ions. Thus, for notation brevity, we define the neighbourhoods only with respect to the arrangement x.

1. ***k*-ion swap.** This neighborhood consists of all feasible arrangements that are produced by swapping the locations of k ions. The size of this neighbourhood is $O(n^k k!)$.
2. ***k*-swap.** This neighborhood is parameterized by a discretization step δ. Using δ we discretize the unit cell and then we perform swaps of k ions with the content of every point of the discretization. So, an ion can swap positions with another ion, or simply move to another vacant position. Again, we take into account only the feasible arrangements of the ions. The size of this neighbourhood is $O(n^k k!/\delta^{3k})$.
3. **Axes.** This neighborhood has a parameter δ and computes the following for every ion i. Firstly, for every dimension it computes a plane parallel to the corresponding facet of the unit cell and contains the ion i. The intersection of any pair of these planes defines an "axis". Then, this axis is discretized according to δ. The neighborhood locates the ion to every point on the discretization on the three axes and we keep only the arrangements that are feasible. The size of this neighbourhood is $O(n/\delta)$.

In all of our neighborhoods, we are using a *greedy* rule to choose x'; x' is an arrangement that achieves the minimum energy in $N(x)$.

4 Algorithms

We propose two algorithms. The first one is a step towards answering Open Question 1 while the second is a heuristic for MinStructure problem.

For Open Question 1, we propose the *depth energy computation* for estimating the energy of a structure. Our algorithm has a single parameter, the depth parameter k, and works as follows. Given a crystal structure, it creates k layers around the unit cell with copies of the structure. So, for the unit cell parameters (y, θ) and the arrangement x of n atoms the energy is $E(y, \theta, x) = \sum_{i=1}^{n} \sum_{j \neq i, j \in D(k)} (BE(i,j) + CE(i,j))$, where $D(k)$ denotes the set of ions in the k layers of unit cells, and $BE(i,j)$ and $CE(i,j)$ are computed as in Equations 2 and 1 respectively.

For MinStructure problem, we slightly modify basin hopping. In a step of basing hopping, a structure is randomly chosen and it is followed by a relaxation. Our algorithm applies a combinatorial local search using the Axes neighborhood, since this turned out to be the best among our heuristics, before the relaxation. So, we will perform a relaxation, *only after* combinatorial local search cannot further improve the solution. Our algorithm can be

used as a standalone one and it can also be integrated into *any other* heuristic algorithm for the Crystal Structure Prediction problem since it is oblivious. In addition, it provides a very fast criterion that when it succeeds it guarantees finding a lower energy crystal structure.

5 Experiments

In this section we evaluate our algorithms via experimental simulations. We first focus on $SrTiO_3$ which we use as a benchmark. We do this because it is a well studied composition for which the Crystal Structure Prediction problem is solved. We have implemented the algorithms in Python 2.7 and we use the Atomic Simulation Environment (`https://wiki.fysik.dtu.dk/ase/`) package for setting up, manipulating, running, visualizing and analyzing atomistic simulations. All experiments were performed on a 4-core Intel i7-4710MQ with 8GB of RAM.

Table 1 Comparison between depth approach and GULP for $SrTiO_3$. Energy is in electronvolts (eV). The energy difference shows the average difference in energy between the depth approach and the energy calculated by GULP. Results averaged over 2000 random feasible structures.

	Energy difference					
k	1	2	3	4	5	6
15 atoms	0.0639	0.0226	0.0114	0.0068	0.0045	0.0032
20 atoms	0.0670	0.0238	0.0120	0.0072	0.0047	0.0033

We evaluate the depth energy computation in several different dimensions. For all the experiments we performed for energy computation, we fixed the unit cell to be cubic. Firstly, we evaluate how depth energy computation behaves with respect to k. We see that the method converges very fast and $k = 6$ already achieves accuracy of three decimal points. Then, we compare our depth approach against GULP; see Table 1. Our goal is to provide an intuitively simpler to interpret and work with method for computing the energy. Even though the energy calculated by the depth approach differs from the one calculated by GULP, we observe that the relative energies between two random arrangements remain usually the same even for $k = 1$. To be more precise, let $E_1(x)$ denote the energy of a feasible arrangement x when $k = 1$ and let $E_G(x)$ denote the energy of this arrangement as it is computed by GULP. Our experiments show that if for two random feasible arrangements x_1 and x_2 it holds that $E_1(x_1) < E_1(x_2)$, then $E_G(x_1) < E_G(x_2)$ for 99.8% of 1000 pairs of arrangements. This percentage reaches 100% for $k = 6$. For the "special" arrangement of ions x^* that minimizes the energy computed by GULP, that is $x^* = \text{argmin} E_G(x)$, our experiments show that it is *always true* that $E_k(x^*) < E_k(x)$, for every $k = 1, \ldots, 6$, where x is a random feasible structure over 10000 of them. So, this is a good indication that the arrangement that minimizes the energy for $k = 1$, also minimizes the energy overall. We view this as a striking result; it significantly simplifies the problem thus new, analytical, methods can be derived for the problem.

The next set of experiments compares the three neighborhoods described in Section 3 for $SrTiO_3$[4]. We compare them in several different dimensions: the average CPU time they need in order to find a local optimum with respect to their combinatorial neighborhood and the average drop in energy until they reach such a local optimum (Tables 2 and 4); the average CPU time the relaxation needs starting from such local minimum and the average drop in

[4] The values of the Buckingham parameters can be found in the full version of the paper [2]

energy from relaxation (Tables 3 and 5). In addition, for the case of $SrTiO_3$, we compare how often we can find the optimal arrangement from a single structure. We observe that the Axes neighborhood has the best tradeoff between energy drop and CPU time. The 2-ion-swap neighbourhood outperforms the other two in terms of running time, however it seems to decrease the probability of finding the best arrangement when performing a relaxation on the resulting structures. This renders the use of 2-ion-swap neighbourhood inappropriate. Axes neighborhood is significantly faster and performs smoother in terms of running time than the 2-swap neighbourhood. However, the latter one performs better with respect to the energy drop, which is expected since axes is a subset of the 2-swap neighbourhood. In addition, the relaxation from the local minimum found by 2-swap significantly improves the probability of finding the best arrangement with only one relaxation. We should highlight that there exist structures where the relaxation cannot improve their energy, but the neighborhoods do; hence using Axes neighborhood we can escape from some local minima of the continuous space.

Table 2 Comparison of local neighbourhoods for reaching a combinatorial minimum for $SrTiO_3$ with 15 atoms per unit cell and $\delta = 1$Å (375 grid points). Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arrangements.

Neighbourhood	Running time	Time stdev	Energy drop	Energy drop stdev
Axes	5.36	1.54	13.46	10.60
2-ion swap	0.96	0.33	7.75	8.45
2-swap	34.66	14.06	16.21	10.94

Table 3 Evaluation of relaxation procedure after using a combinatorial neighborhood for $SrTiO_3$ with 15 atoms per unit cell. Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arragnements.

Neighbourhood	Running time	Time stdev	Energy drop	Energy drop stdev	Global minimum
Random structures-GULP	8.80	6.35	18.60	10.26	6.6%
Axes-GULP	7.92	6.16	5.53	2.28	10.0%
2-ion swap-GULP	8.82	6.25	11.09	5.64	4.7%
2-swap-GULP	5.14	5.08	2.79	1.05	14.8%

Table 4 Comparison of local neighbourhoods for reaching a combinatorial minimum for $Y_2Ti_2O_7$ and $\delta = 1$Å (343 grid points). Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arrangements.

Neighbourhood	Running time	Time stdev	Energy drop	Energy drop stdev
Axes	7.56	2.08	8.23	3.71
2-ion swap	0.88	0.43	1.16	1.80
2-swap	27.93	9.98	10.26	4.05

Next, we compare our algorithm for MinStructure against basin hopping where the next structure to relax is chosen at random. Based on the results of our previous experiments, we have chosen the Axes neighbourhood as an intermediate step before the relaxation. We have run these algorithms 200 times for $SrTiO_3$ with 15 atoms per unit cell, and 25 times for $SrTiO_3$ with 20 atoms per unit cell. We report how the energy varies with respect to

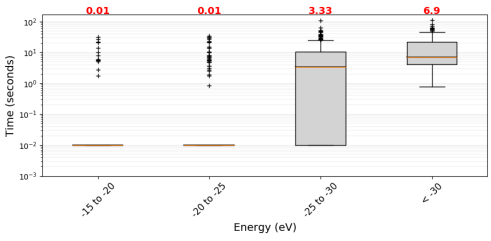
Table 5 Evaluation of relaxation procedure after using a combinatorial neighborhood for $Y_2Ti_2O_7$. Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arragnements.

Neighbourhood	Running time	Time stdev	Energy drop	Energy drop stdev
Random structures-GULP	2.81	1.45	12.84	4.88
Axes-GULP	2.30	1.20	5.09	1.81
2-ion swap-GULP	2.47	2.25	12.29	4.38
2-swap-GULP	1.99	1.09	3.11	0.93

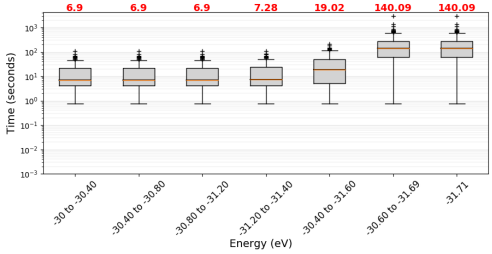
time until the best arrangement is found (Fig. 2) and we report other statistics that further validate our approach (Table 6). As we can see, it is relatively easy to reach low levels of energy and the majority of time is needed to find the absolute minimum. In addition, the overhead posed by the use of the neighbourhood search divided by the time needed by the basin hopping to find the global minimum decreases as the number of the atoms in the unit cell increases.

Table 6 Statistics from the experiments depicted in Figure 2 ($SrTiO_3$ with 15 atoms per unit cell). The corresponding Figures for $SrTiO_3$ with 20 atoms per unit cell can be found in the full version of this paper [2].

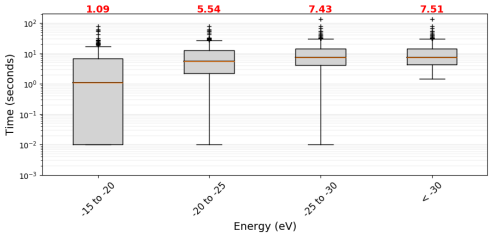
Algorithm	Number of atoms	Total time mean	Total time stdev	Relaxations	Time for relaxations	Time for local search
Axes-GULP	15	227.89	287.21	13.24	126.26	101.63
	20	2280.57	781.66	104.33	1016.72	1049.13
Basin hopping	15	167.89	114.89	18.14	160.79	–
	20	5766.20	4748.33	450.66	4895.60	–



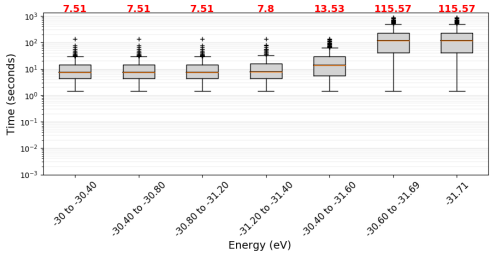
(a) Axes - GULP coarse.



(b) Axes - GULP fine.



(c) GULP coarse.



(d) GULP fine.

Figure 2 Time to reach specific energy levels for $SrTiO_3$ (15 atoms). Figures (a) and (b) correspond to the algorithm of Section 4. Figures (c) and (d) correspond to basin hopping. The median times needed to reach every energy level are depicted in red on the top of each plot.

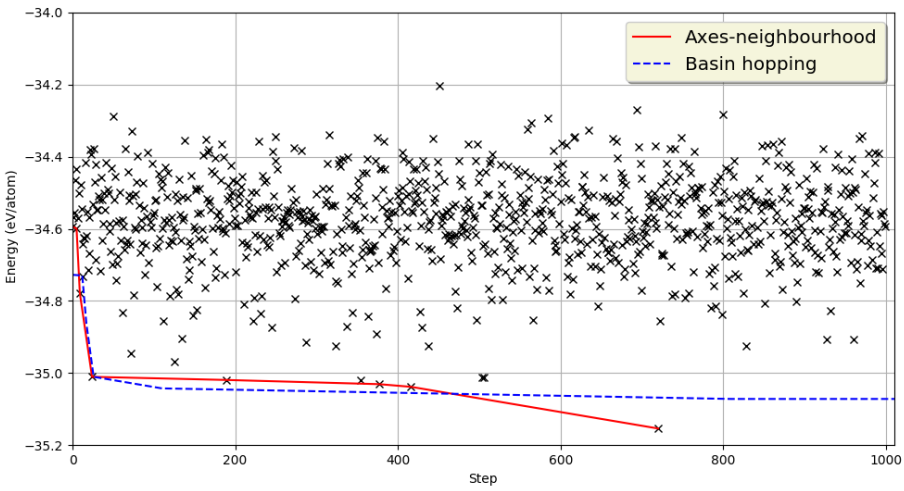


Figure 3 Performance of the Axes algorithm for $Y_2Ti_2O_7$. The black points correspond to the energy found after the relaxation of a point computed by the Axes neighborhood at each step. The red line is the lower envelope of the energy found by our algorithm while the blue line corresponds to the lower envelope of basin hopping.

In our last set of experiments, we compare our algorithm against basin hopping algorithm for $Y_2Ti_2O_7$ which contains 22 atoms in its primitive unit cell. In this set of experiments, we produced 3500 random structures. We simulated basin hopping by sequentially relaxing the constructed structures. However, none of the relaxations managed to find the optimal configuration. Our algorithm, using the same order of structures as before, first used the Axes neighborhood as an intermediate step followed by a relaxation; it managed to find the optimal configuration after visiting only 720 structures (Fig. 3).

6 Conclusions

In this paper we have introduced and studied the Crystal Structure Prediction problem through the lens of computer science. This is an important and very exciting problem in computational chemistry, which computer scientists are not actively studying yet. We have identified several open questions whose solution would have significant impact to the discovery of new materials. These problems are challenging and several different techniques and machineries from computer science could be applied for solving them. Our simple-to-understand algorithms are a first step towards their solution. We hope that our algorithms will be used as benchmarks in the future, since more sophisticated techniques for basin hopping can be invented. For the energy computation via the depth approach, we conjecture that the arrangement that minimizes the energy for $k = 1$ or $k = 2$, matches the arrangement that minimizes the energy when it is computed via GULP. Our numerical simulations provide significant evidence towards this. A formal result of this would greatly simplify the objective function of the optimization problem and it would give more hope to faster methods for relaxation. In addition, it could provide the foundations for new techniques for crystal structure prediction. Our algorithm that utilizes the Axes neighborhood as an intermediate step before relaxation, seems to speed up the time the standard basin hopping needs to find

the global minimum. Are there any other neighborhoods that outperform the Axes one? Can local search, or Axes neighborhood in particular, improve existing methods for crystal structure prediction by a simple integration as an intermediate step? We believe that this is indeed the case.

References

1. Duncan Adamson, Argyrios Deligkas, Vladimir V. Gusev, and Igor Potapov. On the hardness of energy minimisation for crystal structure prediction. In *SOFSEM 2020: Theory and Practice of Computer Science - 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20-24, 2020, Proceedings*, pages 587–596, 2020. doi:10.1007/978-3-030-38919-2_48.
2. Dmytro Antypov, Argyrios Deligkas, Vladimir Gusev, Matthew J Rosseinsky, Paul G Spirakis, and Michail Theofilatos. Crystal structure prediction via oblivious local search. *arXiv preprint arXiv:2003.12442*, 2020.
3. Richard A Buckingham. The classical equation of state of gaseous helium, neon and argon. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 168(933):264–283, 1938.
4. Seth T Call, Dmitry Yu Zubarev, and Alexander I Boldyrev. Global minimum structure searches via particle swarm optimization. *Journal of computational chemistry*, 28(7):1177–1186, 2007.
5. C Collins, GR Darling, and MJ Rosseinsky. The flexible unit structure engine (fuse) for probe structure-based composition prediction. *Faraday discussions*, 211:117–131, 2018.
6. C Collins, MS Dyer, MJ Pitcher, GFS Whitehead, M Zanella, P Mandal, JB Claridge, GR Darling, and MJ Rosseinsky. Accelerated discovery of two crystal structure types in a complex inorganic phase field. *Nature*, 546(7657):280, 2017.
7. Stefano Curtarolo, Gus LW Hart, Marco Buongiorno Nardelli, Natalio Mingo, Stefano Sanvito, and Ohad Levy. The high-throughput highway to computational materials design. *Nature materials*, 12(3):191, 2013.
8. David M Deaven and Kai-Ming Ho. Molecular geometry optimization with a genetic algorithm. *Physical review letters*, 75(2):288, 1995.
9. CM Freeman, JM Newsam, SM Levine, and CRA Catlow. Inorganic crystal structure prediction using simplified potentials and experimental unit cells: application to the polymorphs of titanium dioxide. *Journal of Materials Chemistry*, 3(5):531–535, 1993.
10. Julian D Gale and Andrew L Rohl. The general utility lattice program (gulp). *Molecular Simulation*, 29(5):291–341, 2003.
11. Romain Gautier, Xiuwen Zhang, Linhua Hu, Liping Yu, Yuyuan Lin, Tor OL Sunde, Danbee Chon, Kenneth R Poeppelmeier, and Alex Zunger. Prediction and accelerated laboratory discovery of previously unknown 18-electron abx compounds. *Nature chemistry*, 7(4):308, 2015.
12. Stefan Goedecker. Minima hopping: An efficient search method for the global minimum of the potential energy surface of complex molecular systems. *The Journal of chemical physics*, 120(21):9911–9917, 2004.
13. Geoffroy Hautier, Chris Fischer, Virginie Ehrlacher, Anubhav Jain, and Gerbrand Ceder. Data mined ionic substitutions for the discovery of new compounds. *Inorganic chemistry*, 50(2):656–663, 2010.
14. David C Lonie and Eva Zurek. Xtalopt: An open-source evolutionary algorithm for crystal structure prediction. *Computer Physics Communications*, 182(2):372–387, 2011.
15. Nicola Nosengo. Can artificial intelligence create the next wonder material? *Nature News*, 533(7601):22, 2016.

16 Artem R Oganov and Colin W Glass. Crystal structure prediction using ab initio evolutionary techniques: Principles and applications. *The Journal of chemical physics*, 124(24):244704, 2006.

17 Artem R Oganov, Chris J Pickard, Qiang Zhu, and Richard J Needs. Structure prediction drives materials discovery. *Nature Reviews Materials*, page 1, 2019.

18 J Pannetier, J Bassas-Alsina, J Rodriguez-Carvajal, and V Caignaert. Prediction of crystal structures from crystal chemistry rules by simulated annealing. *Nature*, 346(6282):343, 1990.

19 Chris J Pickard. Real-space pairwise electrostatic summation in a uniform neutralizing background. *Physical Review Materials*, 2(1):013806, 2018.

20 Chris J Pickard and RJ Needs. High-pressure phases of silane. *Physical Review Letters*, 97(4):045504, 2006.

21 Chris J Pickard and RJ Needs. Ab initio random structure searching. *Journal of Physics: Condensed Matter*, 23(5):053201, 2011.

22 Martin U Schmidt and Ulli Englert. Prediction of crystal structures. *Journal of the Chemical Society, Dalton Transactions*, 1996(10):2077–2082, 1996. doi:10.1039/DT9960002077.

23 J Christian Schön. How can databases assist with the prediction of chemical compounds? *Zeitschrift für anorganische und allgemeine Chemie*, 640(14):2717–2726, 2014.

24 J Christian Schön and Martin Jansen. First step towards planning of syntheses in solid-state chemistry: determination of promising structure candidates by global optimization. *Angewandte Chemie International Edition in English*, 35(12):1286–1304, 1996.

25 Theo Siegrist and Terrell A Vanderah. Combining magnets and dielectrics: Crystal chemistry in the bao- fe2o3- tio2 system. *European Journal of Inorganic Chemistry*, 2003(8):1483–1501, 2003.

26 TA Vanderah, JM Loezos, and RS Roth. Magnetic dielectric oxides: subsolidus phase relations in the bao: Fe2o3: Tio2system. *Journal of Solid State Chemistry*, 121(1):38–50, 1996.

27 P Villars, K Brandenburg, M Berndt, S LeClair, A Jackson, Y-H Pao, B Igelnik, M Oxley, B Bakshi, P Chen, et al. Binary, ternary and quaternary compound former/nonformer prediction via mendeleev number. *Journal of alloys and compounds*, 317:26–38, 2001.

28 David J Wales and Jonathan PK Doye. Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. *The Journal of Physical Chemistry A*, 101(28):5111–5116, 1997.

29 Jearl D Walker. *Fundamentals of physics extended*. Wiley, 2010.

30 Yanchao Wang, Jian Lv, Li Zhu, and Yanming Ma. Crystal structure prediction via particle-swarm optimization. *Physical Review B*, 82(9):094116, 2010.

Variable Shift SDD: A More Succinct Sentential Decision Diagram

Kengo Nakamura
NTT Communication Science Laboratories, Kyoto, Japan
kengo.nakamura.dx@hco.ntt.co.jp

Shuhei Denzumi
Graduate School of Information Science and Technology, The University of Tokyo, Japan
denzumi@mist.i.u-tokyo.ac.jp

Masaaki Nishino
NTT Communication Science Laboratories, Kyoto, Japan
masaaki.nishino.uh@hco.ntt.co.jp

Abstract

The Sentential Decision Diagram (SDD) is a tractable representation of Boolean functions that subsumes the famous Ordered Binary Decision Diagram (OBDD) as a strict subset. SDDs are attracting much attention because they are more succinct than OBDDs, as well as having canonical forms and supporting many useful queries and transformations such as model counting and `Apply` operation. In this paper, we propose a more succinct variant of SDD named *Variable Shift SDD* (VS-SDD). The key idea is to create a unique representation for Boolean functions that are equivalent under a specific variable substitution. We show that VS-SDDs are never larger than SDDs and there are cases in which the size of a VS-SDD is exponentially smaller than that of an SDD. Moreover, despite such succinctness, we show that numerous basic operations that are supported in polytime with SDD are also supported in polytime with VS-SDD. Experiments confirm that VS-SDDs are significantly more succinct than SDDs when applied to classical planning instances, where inherent symmetry exists.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Computing methodologies → Knowledge representation and reasoning

Keywords and phrases Boolean function, Data structure, Sentential decision diagram

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.22

Related Version A full version of the paper is available at `https://arxiv.org/abs/2004.02502`.

1 Introduction

The succinct representations of a Boolean function have long been studied in the computer science community. Among them, the *Ordered Binary Decision Diagram* (OBDD) [5] has been used as a prominent tool in various applications. An OBDD represents a Boolean function as a directed acyclic graph (DAG). The reason for the popularity of OBDDs is that it can often represent a Boolean function very succinctly while supporting many useful queries and transformations in polytime with respect to the compilation size.

In the last few years, the *Sentential Decision Diagram* (SDD) [9], which is again a DAG representation, has also attracted attention [23, 20]. SDDs have a tighter bound on the compilation size than OBDDs [9], and there are cases in which the use of SDDs can make the size exponentially smaller than OBDDs [3]. In addition, SDDs also support a number of queries and transformations in polytime. Among them, the most important polytime operation is the `Apply` operation, which takes two SDDs representing two Boolean functions f, g and binary operator $\circ$, such as conjunction ($\wedge$) and disjunction ($\vee$), and returns the

SDD representing the Boolean function $f \circ g$. This operation is fundamental in compiling an arbitrary Boolean function into an SDD, as well as in proving the polytime solvability of various important and useful operations.

One of the reasons why OBDDs and SDDs, as well as many other such DAG representations, can express a Boolean function succinctly is that they share identical substructures that represent the equivalent Boolean function; they represent a Boolean function by recursively decomposing it into subfunctions that can also be represented as DAGs. If a decomposition generates equivalent subfunctions, we do not need to have multiple DAGs, and thus it can more succinctly represent the original Boolean function. Since the effectiveness of such representations depend on the DAG size, representations that are more succinct while still supporting useful operations are always in demand.

In this paper, we propose a new SDD-based structure named *Variable Shift SDD* (VS-SDD); it can even more succinctly represent Boolean functions, while supporting polytime `Apply` operations. The key idea is to extend the condition for sharing DAGs. While an SDD can share DAGs representing identical Boolean functions, a VS-SDD can share DAGs representing Boolean functions that are equivalent under a specific variable substitution. For example, consider two Boolean functions $f = A \wedge B$ and $g = C \wedge D$ defined over variables A, B, C, D. An SDD cannot share DAGs representing f and g since they are not equivalent. On the other hand, VS-SDD can share them since f and g are equivalent under the variable substitution that exchanges A with C and B with D. Such Boolean functions appear in a wide range of situations. One typical example is modeling time-evolving systems; such as, we want to find a sequence of assignments of variables $\mathbf{x}_1, \ldots, \mathbf{x}_T$ over timestamps $t = 1, \ldots, T$ such that every $\mathbf{x}_t$ satisfies the condition that $h(\mathbf{x}_t) = \mathit{true}$. Such a sequence is modeled as Boolean function $f(\mathbf{X}_1, \ldots, \mathbf{X}_T) = h^{(1)}(\mathbf{X}_1) \wedge \cdots \wedge h^{(T)}(\mathbf{X}_T)$, where $h^{(t)}$ is $h(\mathbf{X})$ defined over $\mathbf{X}_t$. Since all $h^{(t)}(\mathbf{X}_t)$ are equivalent under variable substitutions, it is highly possible that VS-SDD can yield more succinct representations.

Technically, these advantages of VS-SDD are obtained by introducing the indirect specification of depending variables. Every SDD is associated with a set of variables that the corresponding Boolean function depends on. In SDD, such set of variables are represented by IDs, where each set of variables has a unique ID. On the other hand, VS-SDD represents such sets of variables by storing the *difference* of IDs. This allows the sharing of the Boolean functions that are equivalent under specific types of variable substitutions.

Our main results are as follows:

- VS-SDDs are never larger than their SDDs equivalents. Moreover, there is a class of Boolean functions for which VS-SDDs are exponentially smaller than SDDs.
- VS-SDD supports polytime `Apply`. Moreover, the queries and transformations listed in [10] that SDDs support in polytime are also supported in polytime by VS-SDDs.
- We experimentally confirm that when applied to classical planning instances, VS-SDDs are significantly smaller than SDDs.

To summarize, VS-SDDs incur no additional overhead over SDDs while being potentially much smaller than SDDs.

The rest of this paper is organized as follows. Sect. 2 reviews related works. Sect. 3 gives the preliminaries. Sect. 4 introduces SDD, on which our proposed structure is based. Sect. 5 describes the formal definition of the equivalence relation we want to share, the definition of VS-SDD, and the relation between them. Sect. 6 examines the properties of VS-SDDs. Sect. 7 deals with the operations on VS-SDDs, especially `Apply`. Sect. 8 mentions some implementation details that ensure that VS-SDDs suffer no overhead penalty relative to SDDs. Sect. 9 provides experiments and their results, and Sect. 10 gives concluding remarks.

2 Related Works

There have been studies that attempt to share the substructures that represent the "equivalent" Boolean functions up to a conversion. For OBDDs, the most famous among them are complement edges and attributed edges [16, 18]. For example, with complement edges, we can share the substructures representing the equivalent Boolean functions up to taking a negation. However, this study does not focus on the solvability of the operations in the compressed form. Actually, some `Apply` operations cannot be performed in a compressed form. After that, the differential BDD [1], especially ↑ΔBDD, was proposed to share equivalent Boolean functions up to the shifting of variables, that is, given the total order of the variables, shift them uniformly to share isomorphic substructures. This structure supports operations like `Apply`, but its complexity depends on the number of variables, which means that this operation is not supported in polytime of the compilation size. With regard to other representations, Sym-DDG/FBDD [2], based on DDG [11] and FBDD [12], can share equivalent functions up to variable substitution. Since their method adopts a permutation of variables, it can, in principle, treat any variable substitution. However, Sym-DDG/FBDD fails to support some important operations such as conditioning and `Apply`. With regard to these previous works, VS-SDD differs in three points. First, to the best of our knowledge, VS-SDD is the first attempt to extend the equivalence relationships of an SDD. We should note that VS-SDD is not obtained by a straightforward application of the techniques invented for OBDDs. Second, VS-SDD has theoretical guarantees on its size. Last, it supports the flexible polytime `Apply` operation.

3 Preliminaries

We use an uppercase letter (e.g., X) to represent a variable and a lowercase letter (e.g., x) to denote its assignment (either *true* or *false*). A bold uppercase letter (e.g., $\mathbf{X}$) represents a set of variables and a bold lowercase letter (e.g., $\mathbf{x}$) denotes its assignment. *Boolean function* $f(\mathbf{X})$ is a function that maps each assignment of $\mathbf{X}$ to either *true* or *false*. The *conditioning* of f on instantiation $\mathbf{X}$, written $f|\mathbf{x}$, is the subfunction that results from setting variables $\mathbf{X}$ to their values in $\mathbf{x}$. We say f *essentially depends* on variable X iff $f|X \neq f|\neg X$. We take $f(\mathbf{Z})$ to mean that f can only essentially depend on variables in $\mathbf{Z}$. A *trivial* function maps all its inputs to *false* (denoted *false*) or maps all to *true* (denoted *true*).

Consider an ordered full binary tree. For two nodes u, w in it, we say w is a *left descendant* (resp. *right descendant*) of u if w is a (not necessarily proper) descendant of the left (resp. right) child of u.

4 Sentential Decision Diagrams

First, we introduce SDD. It is a data structure that can represent a Boolean function as a directed acyclic graph (DAG) like OBDD.

Let f be a Boolean function and $\mathbf{X}, \mathbf{Y}$ be non-intersecting sets of variables. The $(\mathbf{X}, \mathbf{Y})$-*decomposition* of f is $f = \bigvee_{i=1}^{n}[p_i(\mathbf{X}) \wedge s_i(\mathbf{Y})]$, where $p_i(\mathbf{X})$ and $s_i(\mathbf{Y})$ are Boolean functions. Here $p_1, \ldots, p_n$ are called *primes* and $s_1, \ldots, s_n$ are called *subs*. We denote $(\mathbf{X}, \mathbf{Y})$-decomposition as $\{(p_1, s_1), \ldots, (p_n, s_n)\}$, where n is the size of the decomposition. The pair (p_i, s_i) is called an *element*. An $(\mathbf{X}, \mathbf{Y})$-decomposition is called $\mathbf{X}$-*partition* iff $p_i \wedge p_j = \mathit{false}$ for all $i \neq j$, $\bigvee_{i=1}^{n} p_i = \mathit{true}$, and $p_i \neq \mathit{false}$ for all i. If $s_i \neq s_j$ for all $i \neq j$, the partition is called *compressed*. It is known that a function $f(\mathbf{X}, \mathbf{Y})$ has exactly one compressed $\mathbf{X}$-partition (see Theorem 3 of [9]).

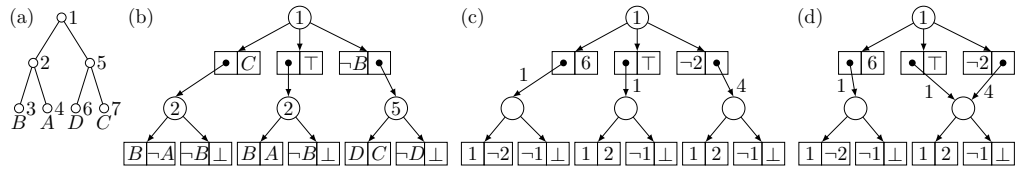


Figure 1 (a) An example of a vtree. (b) The SDD of $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ that respects the vtree of (a). (c)(d) The VS-SDD of $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ given the vtree (a) with offset 1. Here (d) is the more reduced form than (c).

An SDD decomposes a Boolean function by recursively applying **X**-partitions. The structure of partitions is determined by an ordered full binary tree called the *vtree*; its leaves have a one-to-one correspondence with variables. Here, each internal node partitions the variables into those in the left subtree (**X**) and those in the right subtree (**Y**). For example, the vtree in Fig. 1(a) shows the recursive partition of variables A, B, C, D. The root node represents the partition of variables to $\{A, B\}, \{C, D\}$, while the left child of the root represents the partition $\{A\}, \{B\}$. SDD implements **X**-partitions by following these recursive partitions of variables.

Let $\langle \cdot \rangle$ be a mapping from an SDD to a Boolean function (i.e., the semantics of SDD). The SDD is defined recursively as follows.

▶ **Definition 1.** *The following α is an SDD that respects vtree node v.*

- (constant) $\alpha = \top$ *or* $\alpha = \bot$. *Semantics:* $\langle \top \rangle = true$ *and* $\langle \bot \rangle = false$.
- (literal) $\alpha = X$ *or* $\alpha = \neg X$, *and v is a leaf node with variable X. Semantics:* $\langle X \rangle = X$ *and* $\langle \neg X \rangle = \neg X$.
- (decomposition) $\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}$, *and v is an internal node. Here each p_i is an SDD respecting a left descendant node of v, each s_i is an SDD respecting a right descendant node of v, and $\langle p_1 \rangle, \ldots, \langle p_n \rangle$ form a partition. Semantics:* $\langle \alpha \rangle = \bigvee_{i=1}^{n} [\langle p_i \rangle \wedge \langle s_i \rangle]$.

The size of α (denoted by $|\alpha|$) is defined as the sum of the sizes of all its decompositions.

Given the vtree of Fig. 1(a), Fig. 1(b) depicts an SDD that respects the vtree node labeled 1 and represents $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$. At the top level, f is decomposed as $[(\neg A \wedge B) \wedge C] \vee [(A \wedge B)] \vee [\neg B \wedge (C \wedge D)]$. This is the compressed $\{A, B\}$-partition since primes $\neg A \wedge B$, $A \wedge B$ and $\neg B$ satisfy the condition for $\{A, B\}$-partition and subs are all different. Here each circle represents a decomposition node, and the number inside each circle indicates the respecting vtree node ID. The size of the SDD is 9.

There are two classes of canonical SDDs. We say a class of SDDs is *canonical* iff, given a vtree, for any Boolean function f, there is exactly one SDD in this class that represents f. Here we consider only *reduced* SDDs, i.e. the SDDs such that the identical substructures are fully merged.

▶ **Definition 2.** *We say SDD α is* compressed *iff all partitions in α are compressed. We say α is* trimmed *iff it does not have decompositions of the form $\{(\top, \beta)\}$ and $\{(\beta, \top), (\neg\beta, \bot)\}$, and* lightly trimmed *iff it does not have decompositions of the form $\{(\top, \top)\}$ and $\{(\top, \bot)\}$. We say α is* normalized *iff for each decomposition that respects vtree node w, its primes respect the left child of w and its subs respect the right child of w.*

▶ **Theorem 3** ([9])**.** *Compressed and trimmed SDDs are canonical. Also, compressed, lightly trimmed, and normalized SDDs are canonical.*

The key property of SDDs is that they support the polytime `Apply` operation, which takes, given vtree v, two SDDs α, β and binary operation $\circ$, and computes a new SDD that represents $\langle \alpha \rangle \circ \langle \beta \rangle$ in $O(|\alpha||\beta|)$ time. Using `Apply`, we can compile an arbitrary Boolean function into an SDD.

5 Variable Shift Sentential Decision Diagrams

We now introduce our more succinct variant of SDD, named *variable shift SDD* (VS-SDD). As mentioned above, an SDD expresses a Boolean function succinctly by sharing equivalent substructures that represent the same Boolean subfunction. The motivation to introduce VS-SDD is, in addition to this, to share substructures that represent equivalent Boolean functions under a particular variable substitution.

First, we briefly describe the idea by using an intuitive example. Let us consider two Boolean functions $f = X_1 \wedge X_2$ and $g = X_3 \wedge X_4$ defined over variables $X_1, \ldots, X_4$. Apparently, f and g are not equivalent, but they are equivalent if we exchange X_1 with X_3 and X_2 with X_4. We formally define this equivalency of Boolean functions below.

▶ **Definition 4.** *We say two Boolean functions f, g defined over $\mathbf{X}$ are* substitution-equivalent *with permutation π if $f(X_1 = x_1, \ldots, X_M = x_M) = g(X_1 = x_{\pi(1)}, \ldots, X_M = x_{\pi(M)})$ for any assignment $\mathbf{x}$, where $M = |\mathbf{X}|$ and $\pi : \{1, \ldots, M\} \mapsto \{1, \ldots, M\}$ is a bijection.*

In the above example, f and g are substitution equivalent with π satisfying $\pi(3) = 1$ and $\pi(4) = 2$. For $i = 1, 2$, this permutation is defined by simply adding constant to an input, i.e., $\pi(i) = i + c$ $(i = 1, 2)$ where the constant $c = 2$. This result implies that a class of substitution-equivalent functions can be represented as the pair of a base representation and constant value c. VS-SDD exploits this idea.

5.1 Definition of the structure

Now we consider the structure and semantics of VS-SDD. VS-SDD shares many properties with SDDs; it is defined with a vtree and a DAG structure representing recursive $\mathbf{X}$-partitions following the vtree. VS-SDD has two main differences from SDD. First, it associates every vtree node with an integer ID and it considers some mathematical operations over them. We use $\mathtt{ID}(v)$ to represent the ID associated with vtree node v, and $\mathtt{ID}^{-1}(i)$ to represent the vtree node that corresponds to ID i. In the following, we assume that integer IDs of vtree nodes are assigned following a preorder traversal of the vtree. The IDs assigned to the vtree in Fig. 1(a) satisfy this condition. Second, while SDD represents a Boolean function as a node of a DAG, VS-SDD represents a Boolean function as a pair (α, k) of node α in a DAG and integer k. We say α is the VS-SDD *structure* and k is its *offset*. We use $\langle \alpha, k \rangle$ as a mapping from VS-SDD (α, k) to the corresponding Boolean function.

▶ **Definition 5.** *Given vtree v, the following (α, k) is a VS-SDD.*

- (constant) *$\alpha = \top$ or $\alpha = \bot$. Semantics: $\langle \top, \cdot \rangle = true$ and $\langle \bot, \cdot \rangle = false$.*
- (literal) *$\alpha = \mathbf{v}$ or $\alpha = \neg\mathbf{v}$, and $\mathtt{ID}^{-1}(k)$ is a leaf vtree node. Semantics: $\langle \mathbf{v}, k \rangle = l(\mathtt{ID}^{-1}(k))$ and $\langle \neg\mathbf{v}, k \rangle = \neg l(\mathtt{ID}^{-1}(k))$, where $l(v)$ is a variable corresponding to vtree node v.*
- (decomposition) *$\alpha = \{([p_1, d_1], [s_1, e_1]), \ldots, ([p_n, d_n], [s_n, e_n])\}$, and $\mathtt{ID}^{-1}(k)$ is an internal node of v. Here each p_i is a VS-SDD structure and d_i is an integer such that $\mathtt{ID}^{-1}(d_i + k)$ is a left descendant vtree node of $\mathtt{ID}^{-1}(k)$. Similarly, each s_i is a VS-SDD structure and e_i is integer such that $\mathtt{ID}^{-1}(e_i + k)$ is a right descendant node of $\mathtt{ID}^{-1}(k)$ and Boolean functions $\langle p_1, d_1 + k \rangle, \ldots, \langle p_n, d_n + k \rangle$ form a partition. Semantics: $\langle \alpha, k \rangle = \bigvee_{i=1}^{n} (\langle p_i, d_i + k \rangle \wedge \langle s_i, e_i + k \rangle)$.*

The size of α (denoted by $|\alpha|$) is defined as the sum of the sizes of all decompositions.

Given the vtree of Fig. 1(a), Fig. 1(c)-(d) depict the VS-SDDs representing $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$, where Fig. 1(d) is a further reduced form created by sharing the identical substructures in Fig. 1(c). Here the offset is written in the circle of the root node.

Every prime $[p_i, d_i]$ is drawn as an arrow to structure p_i annotated with d_i, except for the following cases. If $p_i = \mathbf{v}$ (resp. $\neg\mathbf{v}$), it is drawn as simply d_i (resp. $\neg d_i$). If p_i is either of $\top$ or $\bot$, it is represented by p_i itself, since the value of d_i has no effect on the semantics. Subs $[s_i, e_i]$ are treated in the same way.

We first give an interpretation of VS-SDD. By comparing the SDD in Fig. 1(b) with the VS-SDD in Fig. 1(c) having the same structure, we find they differ only in the labels of nodes and edges. Actually, we can construct the SDD of Fig. 1(b) from the VS-SDD in Fig. 1(c) in the following way. Let P_α be a path from the root to VS-SDD structure α and D_{P_α} be the sum of the offset and edge values appearing along the path. Then, $\mathtt{ID}^{-1}(D_{P_\alpha})$ is the vtree node that the corresponding SDD node respects. For example, the leftmost child of the root node in the VS-SDD in Fig. 1(c) has offset value 6. The sum of offset values for this node is $1 + 6 = 7$ and $\mathtt{ID}^{-1}(D_{P_\alpha})$ corresponds to the leaf vtree node having variable C. In this way, VS-SDD can be seen as an SDD variant that employs an indirect way of representing the respecting vtree nodes.

5.2 Substitution-equivalency in VS-SDDs

Next we show how substitution-equivalent functions are shared in VS-SDD. In Fig. 1(d), we should observe that the bottom-right node (say β) represents two substitution-equivalent functions $A \wedge B$ and $C \wedge D$. There are two different paths (say P_1 and P_2) from the root to β, and they correspond to different offset values $D_{P_1} = 1 + 1 = 2$ and $D_{P_2} = 1 + 4 = 5$. Therefore, β is used in two VS-SDDs $(\beta, 2)$ and $(\beta, 5)$ and they correspond to $A \wedge B$ and $C \wedge D$, respectively. In this way, substitution-equivalent functions are represented by VS-SDDs with the same structure and different offsets.

Now we proceed to the formal description. Let u, w be isomorphic subtrees of vtree v, $\mathbf{X}$ be the set of variables corresponding to the leaves of v, and M be the number of variables. We consider permutation $\pi_{u,w} : \{1, \ldots, M\} \mapsto \{1, \ldots, M\}$ that preserves the relation between u and w. That is, let X_i and X_j be the variables associated with leaf nodes u' in u and w' in w, respectively. We assume that u' and w' are associated through the graph isomorphism between u and w. Then $\pi_{u,w}$ is the bijection satisfying $\pi_{u,w}(j) = i$ for every pair of X_i and X_j corresponding to the leaf nodes of u and w. If u and w are isomorphic and we employ preorder IDs, then the difference in IDs of corresponding nodes of u and w is unique. We call this the *shift* between u, w and denote it as δ. For example, in the vtree in Fig. 1(a), two child nodes of the root node represent isomorphic vtrees. In these vtrees $\delta = 3$ for every corresponding node pair.

▶ **Theorem 6.** *Let f, g be Boolean functions that essentially depend on isomorphic vtrees u and w (resp.), where u and w are nodes in the entire vtree v. If f and g are substitution-equivalent with $\pi_{u,w}$ then the compressed and trimmed VS-SDDs (α, k) and (β, ℓ) representing f and g satisfies $\alpha = \beta$ and $\ell = k + \delta$.*

Proof. The Boolean function $\langle \alpha, k + \delta \rangle$ is the one wherein every appearance of every variable $l(\mathtt{ID}^{-1}(i))$ in $\langle \alpha, k \rangle$ is replaced with $l(\mathtt{ID}^{-1}(i + \delta))$. It is equivalent to $\langle \beta, \ell \rangle$. ◀

It is possible that there exist two VS-SDDs (α, k) and (β, ℓ) where $\alpha = \beta$ but vtrees $\mathtt{ID}^{-1}(k)$ and $\mathtt{ID}^{-1}(\ell)$ are not isomorphic. In such case, we do not share their structure. In other words, we share the identical structures only when for the offsets k and ℓ, $\mathtt{ID}^{-1}(k)$ and $\mathtt{ID}^{-1}(\ell)$ are isomorphic. We call this the *identical vtree rule*. The VS-SDD in Fig. 1(d) satisfies this rule. Such rule is unique to VS-SDDs, since in SDDs all identical structures are fully merged (i.e. reduced). This rule is crucial for guaranteeing some attractive properties of VS-SDDs introduced in later sections.

6 Properties of VS-SDD

We show here some basic VS-SDD properties. First, we prove the canonicity of some classes of VS-SDD. Then we give proofs on VS-SDD size.

6.1 Canonicity

We say a class of VS-SDD is canonical iff, given a vtree, for any Boolean function f, there is exactly one VS-SDD in this class representing f. We first introduce two classes of VS-SDDs, both have counterparts in SDDs.

▶ **Definition 7.** *We say VS-SDD (α, k) is* compressed *iff for each VS-SDD (β, ℓ) appearing in (α, k) where β is a decomposition, it forms compressed X-partition. We say a VS-SDD (α, k) is* trimmed *if it contains no decompositions with form of $\{([\top, \cdot], [\beta, d])\}$ and $\{([\beta, d], [\top, \cdot]), ([\neg\beta, d], [\bot, \cdot])\}$. We also say VS-SDD is* lightly trimmed *if it contains no decompositions with form of $\{([\top, \cdot], [\top, \cdot])\}$ and $\{([\top, \cdot], [\bot, \cdot])\}$. We say VS-SDD (α, k) is* normalized *iff for each VS-SDD (β, ℓ) appearing in (α, k) where β is a decomposition, every prime $[p_i, d_i]$ ensures that $\texttt{ID}^{-1}(d_i + \ell)$ is the left child of vtree node $\texttt{ID}^{-1}(\ell)$ and every sub $[s_i, e_i]$ ensures that $\texttt{ID}^{-1}(e_i + \ell)$ is the right child of vtree node $\texttt{ID}^{-1}(\ell)$.*

The proof of canonicity is almost identical to that for SDDs. We first introduce some concepts and notations. We use $(\alpha, k) \equiv (\beta, \ell)$ to represent that the corresponding Boolean functions are identical.

▶ **Definition 8.** *A Boolean function f* essentially depends *on vtree node v if f is not trivial and f is a deepest node that includes all variables that f essentially depends on.*

▶ **Lemma 9** ([9])**.** *A non-trivial function essentially depends on exactly one vtree node.*

▶ **Lemma 10.** *Let (α, k) be a trimmed and compressed VS-SDD. If $(\alpha, k) \equiv$ false, then $\alpha = \bot$. If $(\alpha, k) \equiv$ true, then $\alpha = \top$. Otherwise, $\texttt{ID}^{-1}(k)$ always equals to the vtree node v that $\langle \alpha, k \rangle$ essentially depends on.*

The above lemma suggests that compressed and trimmed VS-SDDs can be partitioned into groups depending on the offset. We can prove the canonicity by exploiting this fact.

▶ **Theorem 11.** *Compressed and trimmed VS-SDDs with the same vtree, v, are canonical. Also, compressed, lightly trimmed, and normalized VS-SDDs with the same vtree,v, are canonical.*

Proof. Here we give the proof for the case of compressed and trimmed VS-SDDs. The proof for compressed, lightly trimmed and normalized SDDs can be constructed in a similar way.

If two compressed SDDs (α, k) and (β, ℓ) satisfy $(\alpha, k) = (\beta, \ell)$, then $(\alpha, k) \equiv (\beta, \ell)$ from the definition. Suppose $\langle \alpha, k \rangle = \langle \beta, \ell \rangle$ and let $f = \langle \alpha, k \rangle = \langle \beta, \ell \rangle$. If $f = \mathit{true}$, then $\alpha = \beta = \top$ and they are canonical. Similarly, if $f = \mathit{false}$, then $\alpha = \beta = \bot$.

Next we consider the case of f being non-trivial. From Lemma 10, $\texttt{ID}^{-1}(k) = w = \texttt{ID}^{-1}(\ell)$ where w is the vtree node that f essentially depends on. Suppose w is a leaf, then VS-SDDs must be literals and hence $(\alpha, k) = (\beta, \ell)$. Suppose now that w is internal and that the theorem holds for VS-SDDs whose offsets correspond to descendant nodes of $\texttt{ID}^{-1}(k)$. Let w^l and w^r be the left and the right subtree of w, respectively. Let $\mathbf{X}$ be variables in w^l, $\mathbf{Y}$ be variables in w^r, $\alpha = \{([p_1, d_1], [s_1, e_1]), \ldots, ([p_n, d_n], [s_n, e_n])\}$ and $\beta = \{([q_1, b_1], [r_1, c_1]), \ldots, ([q_m, b_m], [r_m, c_m])\}$. By the definition, offsets $d_i + k$ and

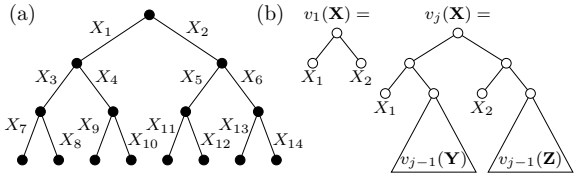


Figure 2 (a) A complete binary tree with variable-labeled edges. (b) The recursive structure of vtree $v_j(\mathbf{X})$. Here X_1 and X_2 indicate the first and second (resp.) variables of $\mathbf{X}$.

$b_j + \ell$ correspond to vtree nodes in w^l and offsets $e_i + k$ and $c_j + \ell$ correspond to vtree nodes in w^r. Since compressed $\mathbf{X}$-partitions $\{(\langle p_1, d_1\rangle, \langle s_1, e_1\rangle), \ldots, (\langle p_n, d_n\rangle, \langle s_n, e_n\rangle)\}$ and $\{(\langle q_1, b_1\rangle, \langle r_1, c_1\rangle), \ldots, (\langle q_m, b_m\rangle, \langle r_m, c_m\rangle)\}$ are identical (see Theorem 3 of [9]), $n = m$ and there is a one-to-one $\equiv$-correspondence between the primes and subs. From the inductive hypothesis, this means there is a one-to-one $=$-correspondence between the primes and subs. This implies $\alpha = \beta$ and thus $(\alpha, k) = (\beta, \ell)$. ◀

6.2 About the Size: Exponential Compression

We here compare VS-SDD size with SDD size. First of all, we observe that VS-SDD is always smaller than SDDs since it is made by sharing substitution-equivalent nodes in SDDs and no other size changes occur.

▶ **Proposition 12.** *For any SDD α defined with vtree v, there exists a VS-SDD whose size is not larger than $|\alpha|$.*

We turn our focus to the best compression ratio of the VS-SDD. Since a vtree has M leaves where M is the number of variables, a vtree might have at most M isomorphic subtrees. Thus the lower bound of VS-SDD size is $1/M$ of SDD when we employ the identical vtree rule. Here we prove that there is a series of functions that almost achieves this compression ratio asymptotically.

▶ **Theorem 13.** *There exists a sequence of Boolean functions $f_1, f_2, \ldots$ such that f_j uses $O(2^j)$ variables, the size of a compressed SDD representing f_j is $\Omega(2^j)$ with any vtree, and that of a compressed VS-SDD representing f_j is $O(j)$ with a particular vtree.*

The compression ratio is $O(j/2^j) = O(\log M/M)$. Theorem 13 makes a stronger statement, because "any" vtree can be considered for SDD.

One of the sequences satisfying Theorem 13 is as follows:

$$f_j(\mathbf{X}) = (\neg X_1 \vee \neg X_2) \wedge \textstyle\bigwedge_{i=1}^{2^j-2} \big((\neg X_i \vee \neg X_{2i+1}) \wedge (\neg X_i \vee \neg X_{2i+2}) \wedge (\neg X_{2i+1} \vee \neg X_{2i+2})\big).$$

By considering a complete binary tree like Fig. 2(a), we observe that $f_j(\mathbf{x}) = \mathit{true}$ iff the edges whose corresponding variables are set to *true* constitute a matching.

We outline the proof here; details are given in the full version. Since the first part of Theorem 13 can easily be proved, we refer to the second part. We define vtree $v_j(\mathbf{X})$ in a recursive manner as shown in Fig. 2(b). Here $\mathbf{Y}$ includes the variables corresponding to the edges below X_1 when considering the complete binary tree as in Fig. 2(a) (namely $X_3, X_4, \ldots$) and $\mathbf{Z}$ includes those corresponding to the edges below X_2 ($X_5, X_6, \ldots$). Now we decompose $f_j(\mathbf{X})$ with respect to $v_j(\mathbf{X})$ by using $f_{j-1}(\mathbf{Y})$, $f_{j-1}(\mathbf{Z})$ and some other subfunctions. We then use the fact that $f_{j-1}(\mathbf{Y})$ and $f_{j-1}(\mathbf{Z})$ are substitution-equivalent with $\pi_{v_{j-1}(\mathbf{Y}), v_{j-1}(\mathbf{Z})}$. By repetitively applying this argument, we observe that by decomposing f_j with respect to v_j, the SDD of f_j has 2^i nodes that represent $f_{j-i}(\cdot)$, which all represent substitution-equivalent functions, and thus the VS-SDD reduces the size exponentially.

Algorithm 1 $\texttt{Apply}(\alpha, \beta, k, \circ)$, which computes a VS-SDD representing $\langle \alpha, k \rangle \circ \langle \beta, k \rangle$ for two normalized VS-SDDs $(\alpha, k), (\beta, k)$ and a binary operator $\circ$.

$\texttt{Cache}(\cdot, \cdot, \cdot) = \texttt{nil}$ initially. $\texttt{Expand}(\alpha)$ returns $\{([\top, \cdot], [\top, \cdot])\}$ if $\alpha = \top$; $\{([\top, \cdot], [\bot, \cdot])\}$ if $\alpha = \bot$; else α. $\texttt{UniqueD}(\gamma)$ returns $\top$ if $\gamma = \{([\top, \cdot], [\top, \cdot])\}$; $\bot$ if $\gamma = \{([\top, \cdot], [\bot, \cdot])\}$; else the unique VS-SDD with elements γ.

if α and β are either of $\top, \bot, \mathbf{v}, \neg\mathbf{v}$ **then**
  return the pair of corresponding value and offset.
else if $\texttt{Cache}(\alpha, \beta, \circ) \neq \texttt{nil}$ **then**
  $\lambda \leftarrow \texttt{Cache}(\alpha, \beta, \circ)$
  return (λ, k)
else
  $\gamma \leftarrow \{\}$
  for all elements $([p_i, d], [s_i, e])$ in $\texttt{Expand}(\alpha)$ **do**
    for all elements $([q_j, d], [r_j, e])$ in $\texttt{Expand}(\beta)$ **do**
      $(p, \ell_p) \leftarrow \texttt{Apply}(p_i, q_j, d + k, \circ)$
      if (p, ℓ_p) is consistent **then**
        $(s, \ell_s) \leftarrow \texttt{Apply}(s_i, r_j, e + k, \circ)$
        add element $([p, \ell_p - k], [s, \ell_s - k])$ to γ
$\lambda \leftarrow \texttt{UniqueD}(\gamma)$, $\texttt{Cache}(\alpha, \beta, \circ) \leftarrow \lambda$
return (λ, k)

7 Operations of VS-SDD

The most important property of VS-SDDs is that they support numerous key operations in polytime. We focus here on the important queries and transformations shown in [10]. Most of these operations are based on `Apply`. `Apply` takes, given a vtree, two VS-SDDs (α, k), (β, ℓ) and binary operation $\circ$ such as $\vee$ (disjunction), $\wedge$ (conjunction) and $\oplus$ (exclusive-or), and returns a VS-SDD of $\langle \alpha, k \rangle \circ \langle \beta, \ell \rangle$. By repeating `Apply` operations, we can flexibly construct VS-SDDs representing various Boolean functions.

To simplify the explanation of `Apply`, we assume that VS-SDDs are normalized and thus have the same offset value k. Given two normalized VS-SDDs (α, k), (β, k), Alg. 1 provides pseudocode for the function $\texttt{Apply}(\alpha, \beta, k, \circ)$. The mechanism behind the `Apply` computation of VS-SDDs is as follows. Let f, g be Boolean functions with the same variable set, and suppose that f is $\mathbf{X}$-partitioned as $f = \bigvee_{i=1}^{n} [p_i(\mathbf{X}) \wedge s_i(\mathbf{Y})]$ and g is also $\mathbf{X}$-partitioned (with the same $\mathbf{X}$) as $g = \bigvee_{j=1}^{m} [q_j(\mathbf{X}) \wedge r_j(\mathbf{Y})]$. Then, $f \circ g$ can be expressed as $\bigvee_{i=1}^{n} \bigvee_{j=1}^{m} [(p_i(\mathbf{X}) \wedge q_j(\mathbf{X})) \wedge (s_i(\mathbf{Y}) \circ r_j(\mathbf{Y}))]$, where $(p_i(\mathbf{X}) \wedge q_j(\mathbf{X})) \wedge (p_{i'}(\mathbf{X}) \wedge q_{j'}(\mathbf{X})) = \mathit{false}$ for $(i, j) \neq (i', j')$ and $\bigvee_{i=1}^{n} \bigvee_{j=1}^{m} (p_i(\mathbf{X}) \wedge q_j(\mathbf{X})) = \mathit{true}$. Thus, computing $p_i \wedge q_j$ and $s_i \circ r_j$ for each (i, j) pair and ignoring the pairs such that $p_i \wedge q_j = \mathit{false}$ yields the $\mathbf{X}$-partition of $f \circ g$. Alg. 1 follows this recursive definition.

▶ **Proposition 14.** $\texttt{Apply}(\alpha, \beta, k, \circ)$ *runs in* $O(|\alpha||\beta|)$ *time.*

The above result is the same as in the case of `Apply` for SDDs. The key to achieving this result is that we use `Cache` without using offset k as a key. We use the fact that if a pair of functions $f(\mathbf{X})$, $f'(\mathbf{Y})$ and $g(\mathbf{X})$, $g'(\mathbf{Y})$, where $\mathbf{X}$ and $\mathbf{Y}$ are non-overlapping, are substitution-equivalent with permutation π, then the composed functions $f \circ g$ and $f' \circ g'$ are also substitution-equivalent with the same permutation π. For example, let $f = A \wedge B$, $f' = C \wedge D$, $g = \neg A$ and $g' = \neg C$, in which f and f', and g and g' (resp.) are substitution-equivalent with permutation $\pi_{\texttt{ID}^{-1}(2),\texttt{ID}^{-1}(5)}$ defined with the vtree in Fig. 1(a). Then $f \vee g = \neg A \vee B$ and $f' \vee g' = \neg C \vee D$ are also substitution-equivalent with $\pi_{\texttt{ID}^{-1}(2),\texttt{ID}^{-1}(5)}$. This means the results of $\texttt{Apply}(\alpha, \beta, k, \circ)$ with different k are all substitution-equivalent. Therefore, we can reuse the result obtained with different offsets.

Table 1 List of supported (a) queries and (b) transformations for SDDs (S), VS-SDDs (V), compressed SDDs (S(C)) and compressed VS-SDDs (V(C)). ✓ indicates the existence of a polytime algorithm, while • indicates such polytime algorithm is shown to be impossible.

	(a) Query	S	**V**	S(C)	**V(C)**
CO	consistency	✓	✓	✓	✓
VA	validity	✓	✓	✓	✓
CE	clausal entailment	✓	✓	✓	✓
IM	implicant check	✓	✓	✓	✓
EQ	equivalence check	✓	✓	✓	✓
CT	model counting	✓	✓	✓	✓
SE	sentential entailment	✓	✓	✓	✓
ME	model enumeration	✓	✓	✓	✓

	(b) Transformation	S	**V**	S(C)	**V(C)**
$\wedge$**C**	conjunction	•	•	•	•
$\wedge$**BC**	bounded conjunction	✓	✓	•	•
$\vee$**C**	disjunction	•	•	•	•
$\vee$**BC**	bounded disjunction	✓	✓	•	•
$\neg$**C**	negation	✓	✓	✓	✓
CD	conditioning	✓	✓	•	•
FO	forgetting	•	•	•	•
SFO	singleton forgetting	✓	✓	•	•

If VS-SDDs are trimmed, we can also define `Apply` operations for them. While similar to the case for trimmed SDDs, the `Apply` for trimmed VS-SDDs are more complicated than that of normalized VS-SDDs since we have to take different operations depending on the combination of offset values of input VS-SDDs. However, the complexity of `Apply` for trimmed VS-SDDs is also $O(|\alpha||\beta|)$. We detail the `Apply` for trimmed VS-SDDs in the full version.

Note that even if two VS-SDDs are compressed, the resulting VS-SDD cannot be assumed to be compressed since the same sub may appear. It is said in [4] that there is a case in which compression makes an SDD exponentially larger, and thus a similar statement holds for VS-SDDs. Therefore, if we oblige the output to be compressed, Prop. 14 does not hold. Note that during `Apply`, compression can be performed by taking the disjunction of primes when the same subs emerge.

By extensively using Prop. 14 with some other algorithms, it can be shown that the various important queries and transformations in [10] can be performed in polytime. The proof is in the full version.

▶ **Proposition 15.** *The results in Table 1 hold.*

Note that some applications, e.g. probabilistic inference [22, 10], need *weighted* model counting, where each variable has a weight. Though this cannot be performed in $O(|\alpha|)$ time for VS-SDD α, it can be performed at least as fast as is possible by using the corresponding SDD, by preparing, for each node, as many counters as the number of unified nodes in the original SDD. Moreover, if the weights of variables are the same for the same vtree structures, we can share counters, which speeds up the computation.

8 Implementation

We should address implementation in order to ensure space-efficiency. One suspects that even if VS-SDD size is never larger than SDD size, the memory usage may increase because we store the information of respecting vtree node ids in the edges of a diagram (differentially) instead of in the nodes. This is true if VS-SDDs are implemented as is.

However, a small modification avoids this problem. First, for a normalized VS-SDD, we simply ignore the differences of vtree node ids attached to the edges. Even so, we can recover the respecting vtree node because if an SDD node respects vtree v, its primes respect the left child of v and its subs respect the right child of v. Second, for a general VS-SDD, we just reuse the structure of the original SDD. Among the SDD nodes that are merged into one in the VS-SDD structure, we just leave one representative (e.g. the one with the smallest respecting vtree node id). Then each of the other nodes has a pointer to the representative node instead of storing the prime-sub pairs. An example of such a structure is drawn in

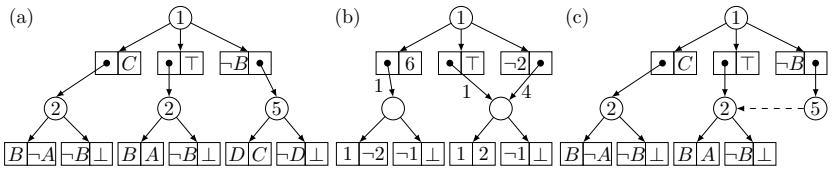


Figure 3 (a)(b) An SDD and a VS-SDD that are the same as Fig. 1. (c) The example of the representation of VS-SDD (b) using original SDD structure (a). The dashed arrow indicates a pointer to the representative node.

Fig. 3(c). Here the dashed arrow indicates a pointer to the representative node described above. Since each decomposition node has at least one prime-sub pair that typically uses two pointers, replacing it by single pointer will never increase memory usage. Working with such a structure does not violate any properties about VS-SDDs, including the operations described above.

9 Evaluation

We use some benchmarks of Boolean functions to evaluate how our approach reduces the size of an SDD. we compile a CNF into an SDD with the dynamic vtree search [7] and then compare the sizes yielded by the SDD and its VS form (VS-SDD). To compile a CNF, we use the SDD package version 2.0 [8] with a balanced initial vtree. Here note that we use for both SDD and VS-SDD the same vtree, which is searched to suit for SDD. All the experiments are conducted on a 64-bit macOS (High Sierra) machine with 2.5 GHz Intel Core i7 CPU (1 thread) and 16 GB RAM.

Here we focus on the planning CNF dataset that was used in the experiment of Sym-DDG [2]. The planning problem naturally exhibits symmetries, e.g. see [21]. Given time horizon T, this data represents a deterministic planning problem with varying initial and goal states. Here we can choose an action from a fixed action set for each time point, and a plan for this problem is a time series of actions for $t = 0, \ldots, T-1$ that leads from the initial state to the goal state. For more details, see [2]. We use the planning problems that were also used in the experiment of Sym-DDG: "blocks-2", "bomb-5-1", "comm-5-2", "emptyroom-4/8", and "safe-5/30", with varying time horizons $T = 3, 5, 7, 10$.

The next focus is on the benchmarks with apparent symmetries. The first one is the N-queens problem, that is, given an $N \times N$ chessboard, place N queens such that no two queens attack each other. We assign a variable to each square in the chessboard, and consider a Boolean function that evaluates *true* iff the *true* variables constitute one answer for this problem. This problem is used as a benchmark in Zero-suppressed BDD and other DD studies [17, 6]. The second one is enumerating matchings of grid graphs. Subgraph enumeration with decision diagrams has several applications; see [14] and [19]. Here the grid graph is often used as a benchmark [13], because it is closely related to self-avoiding walk [15], and subgraph enumeration becomes much harder for larger grids despite their simplicity. We can observe that both the chessboard and the grid have line symmetries and point symmetry. Again we exploit dynamic vtree search implemented in the SDD package.

Table 2 shows the results of our experiments. The "S" column represents SDD size, "V" represents VS-SDD size, and "ratio" indicates the ratio of VS-SDD size compared to SDD size. Here the problems in which the SDD compilation took more than 10 minutes are omitted. For planning problems, the suffix "_tn" stands for $T = n$, and for matching problems, the suffix indicates the grid size. It is observed that for many planning problems, the VS-SDD

Table 2 Results for experiments. The "S" column represents SDD size, "V" represents VS-SDD size, and "ratio" indicates the ratio of VS-SDD size compared to SDD size.

Problem	#vars	S	**V**	ratio
blocks-2_t3	248	8811	7057	80.1%
blocks-2_t5	406	31861	28858	90.6%
bomb-5-1_t3	348	3798	2278	60.0%
bomb-5-1_t5	564	6327	3960	62.6%
bomb-5-1_t7	780	11212	7287	65.0%
bomb-5-1_t10	1104	16514	10426	63.1%
comm-5-2_t3	488	20584	18033	87.6%
emptyroom-4_t3	116	1822	1146	62.9%
emptyroom-4_t5	188	3090	1885	61.0%
emptyroom-4_t7	260	5073	3001	59.2%
emptyroom-4_t10	368	106737	103417	96.8%
emptyroom-8_t3	244	10511	8549	81.3%
safe-5_t3	54	567	441	77.8%
safe-5_t5	86	898	640	71.2%
safe-5_t7	118	1710	1314	76.8%
safe-5_t10	166	2506	1756	70.1%
safe-30_t3	304	5476	4067	74.3%
safe-30_t5	486	8710	6328	72.7%
safe-30_t7	668	14449	10371	71.8%
safe-30_t10	941	23469	17421	74.2%
8-Queens	64	2222	1624	73.1%
9-Queens	81	5559	4767	85.8%
10-Queens	100	10351	9159	88.5%
11-Queens	121	30611	28876	94.3%
Matching-6x6	60	13091	12671	96.8%
Matching-8x8	112	98200	97103	98.8%
Matching-6x18	192	36228	34241	94.5%

reduces the size to around 60% to 80% of the original SDD. We observe that for these cases, many nodes representing substitution-equivalent functions are found among the bottom nodes of the original SDD, which yields the substantial size decrease. These compression ratios are competitive to, and for some cases better than, that of the Sym-DDG [2] compared to the DDG. For the N-queens problems, still better compression ratios are achieved except for $N = 11$. However, for matching enumeration problems, the effect of variable shift is relatively small. One reason is the asymmetry of primes and subs, that is, primes must form a partition while subs do not have such a limitation. The success in planning datasets may be explained as follows. The dynamic vtree search typically gathers variables with strong dependence locally to achieve succinctness. For planning problems, the variables with near time points are gathered, which captures the symmetric nature of the problem.

10 Conclusion

We proposed a variable shift SDD (VS-SDD), a more succinct variant of SDD that is obtained by changing the way in which respecting vtree nodes are indicated. VS-SDD keeps the two important properties of SDDs, the canonicity and the support of many useful operations. The size of a VS-SDD is always smaller than or equal to that of an SDD, and there are cases where the VS-SDD is exponentially smaller than the SDD. Experiments show that our idea effectively captures the symmetries of Boolean functions, which leads to succinct compilation.

References

1 Anuchit Anuchitanukul, Zohar Manna, and Tomás E. Uribe. Differential BDDs. In *Computer Science Today*, pages 218–233, 1995. doi:10.1007/BFb0015246.

2 Anicet Bart, Frédéric Koriche, Jean-Marie Lagniez, and Pierre Marquis. Symmetry-driven decision diagrams for knowledge compilation. In *ECAI*, pages 51–56, 2014. doi:10.3233/978-1-61499-419-0-51.

3 Simone Bova. SDDs are exponentially more succinct than OBDDs. In *AAAI*, pages 929–935, 2016.

4 Guy Van den Broeck and Adnan Darwiche. On the role of canonicity in knowledge compilation. In *AAAI*, pages 1641–1648, 2015.

5 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35:677–691, 1986. doi:10.1109/TC.1986.1676819.

6 Randal E. Bryant. Chain reduction for binary and zero-suppressed decision diagrams. In *TACAS*, pages 81–98, 2018. doi:10.1007/978-3-319-89960-2_5.

7 Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *AAAI*, pages 187–194, 2013.

8 Arthur Choi and Adnan Darwiche. The SDD package: version 2.0. http://reasoning.cs.ucla.edu/sdd/, 2018.

9 Adnan Darwiche. SDD: a new canonical representation of propositional knowledge bases. In *IJCAI*, pages 819–826, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-143.

10 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002. doi:10.1613/jair.989.

11 Héiène Fragier and Pierre Marquis. On the use of partially ordered decision graphs for knowledge compilation and quantified Boolean formulae. In *AAAI*, pages 42–47, 2006.

12 Jordan Gergov and Christoph Meinel. Efficient Boolean manipulation with OBDD's can be extended to FBDD's. *IEEE Trans. Comput.*, 43:1197–1209, 1994. doi:10.1109/12.324545.

13 Hiroaki Iwashita, Yoshio Nakazawa, Jun Kawahara, Takeaki Uno, and Shin-ichi Minato. Efficient computation of the number of paths in a grid graph with minimal perfect hash functions. Technical Report TCS-TR-A-13-64, Division of Computer Science, Hokkaido University, 2013.

14 Donald E. Knuth. *The Art of Computer Programming*, volume 4A: Combinatorial Algorithms, Part I. Addison-Wesley, 2011.

15 Neal Madras and Gordon Slade. *The Self-Avoiding Walk*. Birkhäuser Basel, 2011.

16 Jean-Christophe Madre and Jean-Paul Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *DAC*, pages 205–210, 1988. doi:10.1109/DAC.1988.14759.

17 Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC*, pages 272–277, 1993. doi:10.1145/157485.164890.

18 Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *DAC*, pages 52–57, 1990. doi:10.1145/123186.123225.

19 Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Compiling graph substructures into sentential decision diagrams. In *AAAI*, pages 1213–1221, 2017.

20 Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *IJCAI*, pages 3141–3148, 2015.

21 Héctor Palacios, Blai Bonet, Adnan Darwiche, and Héctor Geffner. Pruning conformant plans by counting models on compiled d-DNNF representations. In *ICAPS*, pages 141–150, 2005.

22 Tian Sang, Paul Beame, and Henry Kautz. Performing Bayesian inference by weighted model counting. In *AAAI*, pages 475–481, 2005.

23 Jonas Vlasselaer, Joris Renkens, Guy Van den Broeck, and Luc De Raedt. Compiling probabilistic logic programs into sentential decision diagrams. In *PLP*, 2014.

Engineering Fused Lasso Solvers on Trees

Elias Kuthe
Computer Science XI, TU Dortmund University, Germany
elias.kuthe@tu-dortmund.de

Sven Rahmann
Genome Informatics, Institute of Human Genetics, University of Duisburg-Essen, Essen, Germany
http://www.rahmannlab.de/people/rahmann
Sven.Rahmann@uni-due.de

Abstract

The graph fused lasso optimization problem seeks, for a given input signal $y = (y_i)$ on nodes $i \in V$ of a graph $G = (V, E)$, a reconstructed signal $x = (x_i)$ that is both element-wise close to y in quadratic error and also has bounded total variation (sum of absolute differences across edges), thereby favoring regionally constant solutions. An important application is denoising of spatially correlated data, especially for medical images.

Currently, fused lasso solvers for general graph input reduce the problem to an iteration over a series of "one-dimensional" problems (on paths or line graphs), which can be solved in linear time. Recently, a direct fused lasso algorithm for tree graphs has been presented, but no implementation of it appears to be available.

We here present a simplified exact algorithm and additionally a fast approximation scheme for trees, together with engineered implementations for both. We empirically evaluate their performance on different kinds of trees with distinct degree distributions (simulated trees; spanning trees of road networks, grid graphs of images, social networks). The exact algorithm is very efficient on trees with low node degrees, which covers many naturally arising graphs, while the approximation scheme can perform better on trees with several higher-degree nodes when limiting the desired accuracy to values that are useful in practice.

2012 ACM Subject Classification Theory of computation → Mathematical optimization; Theory of computation → Dynamic programming; Mathematics of computing → Trees

Keywords and phrases fused lasso, regularization, tree traversal, cache effects

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.23

Supplementary Material Source code: `https://github.com/eqt/treelas`

Funding *Sven Rahmann*: DFG SFB 876/C1.

1 Introduction

The Fused Lasso Signal Approximator (FLSA), also known as Total Variation denoising [22], was introduced by Tibshirani and colleagues [27]. A general formulation is as follows.

▶ **Problem 1.** *Let $G = (V, E)$ be an undirected graph. Let $y = (y_i)_{i \in V} \in \mathbb{R}^V$ be a signal measured in each node. Let $\mu = (\mu_i) \geq 0$ be node weights, and let $\lambda = (\lambda_{ij})_{\{i,j\} \in E} \geq 0$ be edge weights. The problem is to find a minimizer x^* of the convex function*

$$f(x) := \frac{1}{2} \sum_{i \in V} \mu_i \left(x_i - y_i\right)^2 + \sum_{ij \in E} \lambda_{ij} \left|x_i - x_j\right| , \tag{1}$$

i. e., an x^* that is both element-wise close to the observation y (in squared error) and bounded in total variation across edges.

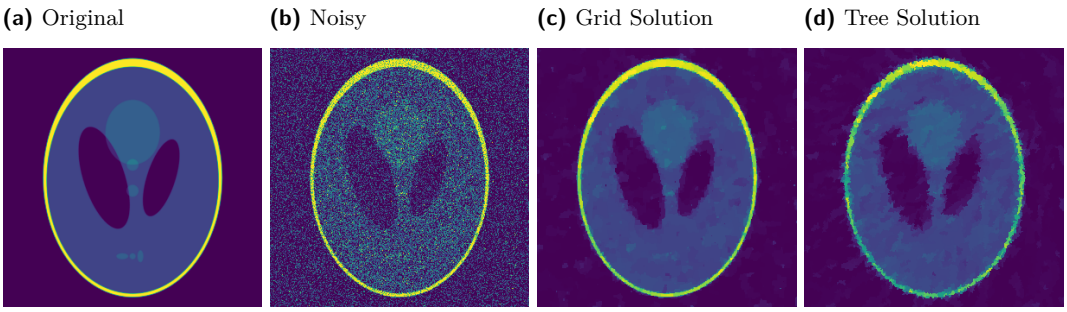


Figure 1 Image Denoising: The Shepp–Logan Phantom [23] (a standard test image), rendered at 300×300 (1a), with added Gaussian noise of standard deviation $\sigma = 0.25$ (1b). The fused lasso with parameter $\lambda = 0.2$ on a grid graph (1c) denoises while keeping most of the edges. The fused lasso on a random spanning tree (1d) is an approximation to the grid solution.

The node weights $\mu_i \geq 0$ represent the degree of confidence or precision of the observation y_i. We explicitly allow nodes i with weight $\mu_i = 0$, i. e. no observation y_i is available. Such nodes are called *latent* in contrast to *visible* nodes. The edge weights λ_{ij} represent our degree of belief that the signal does not change across edge $\{i, j\}$.

Important special cases of graphs for the FLSA are *line graphs*, where $V = \{1, \ldots, n\}$ and edges exist between i and $i + 1$ for $1 \leq i \leq n - 1$. Here one aims to reconstruct a signal x^* from observation y that is assumed to be piece-wise constant or of low total variation. For line graphs there exists a practically and theoretically fast algorithm by Johnson [15].

Other cases of practical importance are given by *grid graphs*, where $V = \{(i, j) : 1 \leq i \leq n,\ 1 \leq j \leq m\}$ with edges between (i, j) and (i', j') if and only if $|i - i'| + |j - j'| = 1$ (grid neighbors). With such graphs, one models pixels of images, such as magnetic resonance images (MRI) [12], and the FLSA is used for denoising these images without smoothing them, because sharp edges are preserved using FLSA, in contrast to the application of convolutions with kernels. Figure 1 shows an example using a common test image.

For general graphs (including grid graphs), so far only iterative approximations have been proposed, e. g. [3]. Often, an efficient line graph solver is used as a subroutine for these iterative methods [29], e. g. Consensus ADMM [25].

Trees (connected acyclic graphs) represent an interesting intermediate class of graphs. Recently, an exact $\mathcal{O}(n \log n)$ time dynamic programming algorithm was presented for trees [16]. To achieve this worst-case running time, it makes use of Fibonacci heaps [11], a complex data structure that has a reputation of being hard to implement, and being more efficient in theory than in practice. Perhaps this is also why we were unable to find any implementation of this method: There exists neither a link in the paper nor on the websites of V. Kolmogorov or T. Pock. We only found the code concerning their work on line graphs. Padilla and colleagues [20] suggested that this algorithm is slow in practice and proposed to use a heuristic instead:

> "Their algorithm [16] is theoretically very efficient, with $\mathcal{O}(n \log n)$ running time, but the implementation that achieves this running time (we have found) can be practically slow for large problem sizes, compared to dynamic programming on a chain graph. Alternative implementations are possible, and may well improve practical efficiency, but as far as we see it, they will all involve somewhat sophisticated data structures in the 'merge' steps in the forward pass of dynamic programming." [20]

So they propose to replace the tree by a line graph of the tree nodes in depth-first search (DFS) order and show that this yields a 2-approximation of the correct solution. We do not believe that such a sacrifice in accuracy should be made for the sake of efficiency.

Contributions. In this work we present the first public implementation of a fused lasso tree solver. We show that (except for adversarial generated instances) a carefully engineered tree solver is only 5 to 20 times slower than a line solver on the same number of nodes.

We describe two algorithms, an exact one that runs in $\mathcal{O}(nq \log q)$ time[1], and an approximation scheme that iteratively refines an approximate solution and closes the gap to the optimal x^*, with running time $\mathcal{O}(n \log(1/\delta))$ for a solution x with $\|x^* - x\|_\infty \leq \delta$. We compare under which conditions the exact tree algorithm and the approximation scheme yield better running time, up to double floating point precision, on different tree datasets with varying node degree distributions.

Our methods handle latent nodes and arbitrary node and edge weights, while other implementations use simplifying assumptions of $\mu_i = 1$ for all nodes $i \in V$ and $\lambda_{ij} = \lambda > 0$ for all edges $\{i, j\} \in E$.

The code, an optimized C++ implementation, is available at `github.com/eqt/treelas`. We provide bindings to Python using `pybind11` [14]. Correctness of solutions was verified via dual solutions on large graphs. The code was tested on Linux, Windows, and Mac OS to make it useful for others beyond a proof-of-concept implementation.

2 Definitions, Notation and Preliminaries

On Trees. We consider undirected simple graphs $G = (V, E)$. A tree is a connected acyclic undirected simple graph.

A tree can be rooted by selecting one node as the root r and directed by pointing all edges towards the root. The next node on the path from a node i to the root r is called the *parent* node $\pi(i)$ of i, and i is called a *child* of $\pi(i)$. The set of children of a node i is denoted as $C(i)$. A node without children is called a *leaf*. The subtree T_i rooted at node i (in the tree with root r) is the induced subgraph containing all nodes j such that i is on the path from j to r, including i itself. This set of nodes is denoted by $V(T_i)$.

A *post-order* on the nodes is any order that lists a parent after all of their children. A *pre-order* is any order that lists a parent before any of their children. As the root node r is treated in a special way, we exclude it from pre- and post-orders.

Merged Nodes. Let x^* be a solution of the fused lasso problem on a tree. We say that two nodes i and j are *merged* in the optimal solution x^* if they are in a region of equal values, i. e. $x^*_i = x^*_j = x^*_k$ for every k on the path from i to j.

The Clip Function and a Differentiability Lemma. The following *clip* function will be used frequently: For real numbers $a \leq b$, define the real-valued function

$$\mathrm{clip}_a^b(x) := \min\{b, \max\{a, x\}\}\,.$$

In other words, $\mathrm{clip}_a^b(x) = x$ is the identity for $x \in [a, b]$, and the value is "clipped" from below to a for $x < a$ and "clipped" from above to b for $x > b$.

[1] Here $q \leq 2n$ is an upper bound on the length of a certain priority queue used in the algorithm; it is a small two-digit constant in our experiments. This is further discussed below.

The following result shows that a certain function defined as a parameterized minimum is, perhaps surprisingly, differentiable in its parameter x. This is a fundamental result used in most fused lasso methods and sometimes called a "Min-Convolution".

▶ **Lemma 1** (Min-Convolution [9, 16]). *Let $g : \mathbb{R} \to \mathbb{R}$ be a convex differentiable function and $\lambda \geq 0$. Then the function*

$$h(x) := \min_z \left[g(z) + \lambda |z - x| \right]$$

is convex and differentiable in x with $h'(x) = \operatorname{clip}_{-\lambda}^{+\lambda}\left[g'(x)\right]$. Furthermore, for given x, we obtain the minimizing $z^* = \operatorname{clip}_a^b(x)$ with $a := \inf\{x \mid g'(x) \geq -\lambda\}$ and $b := \sup\{x \mid g'(x) \leq +\lambda\}$, where $\inf \emptyset = -\infty$ and $\sup \emptyset = +\infty$.

Proof. For the differentiability result, we refer to the literature [9, 16]. We derive the minimizing z^*: For z being minimal the subgradient has to contain 0, i. e.

$$0 \in g'(z^*) + \lambda \partial_z |z^* - x| \,. \tag{2}$$

Let us first assume $x < a$. As g is convex, the derivative g' is monotonically increasing, i. e. $g'(z) < -\lambda$ for all $z \leq x$. That is why the only option to make (2) true is $z^* = a$. Analogously for $x > b$ it follows $z^* = b$. For the case $x \in [a, b]$ we set $z^* = x$ and check $g'(z) = g'(x) \in [-\lambda, +\lambda]$ which shows that (2) is fulfilled. ◀

3 Algorithms

Before going into detail, we discuss the general idea of decomposing the problem on trees.

3.1 Dynamic Programming on Trees

We build solutions of subproblems (Problem 1 restricted to a subtree T_i with root i), while fixing (or assuming) a certain solution value $x_i = x$ in the subtree root.

Therefore, we define $f_i(x)$ as the minimal objective value when optimizing over the variables of the subtree T_i and fixing the subtree's root value to $x_i = x$. This way we obtain a series of functions $f_i(x)$ to be optimized by dynamic programming (bottom-up). Finally, we find the minimizer of f_r when r is the tree root.

For leaf nodes i we have $f_i(x) = \frac{1}{2}\mu_i(x - y_i)^2$ which is differentiable with $f_i'(x) = \mu_i(x - y_i)$. By induction over the tree we have the form

$$f_i(x) = \frac{1}{2}\mu_i(x - y_i)^2 + \sum_{j \in C(i)} \underbrace{\left(\min_{x_j} \lambda_j |x_j - x| + f_j(x_j) \right)}_{=: h_j(x)}, \tag{3}$$

where we abbreviated $\lambda_i := \lambda_{i,\pi(i)}$.

Minimizing along an edge term $h_j(x)$ is a form of "Min-Convolution". Applying Lemma 1 to Eq. (3), we obtain the main tool for the two algorithms described below.

▶ **Theorem 2.** *Let $f_i(x)$ be defined as in (3), as a function of the subtree root's value x. Then f_i is differentiable and*

$$f_i'(x) = \mu_i(x - y_i) + \sum_{j \in C(i)} \operatorname{clip}_{-\lambda_j}^{+\lambda_j}\left[f_j'(x)\right] \,. \tag{4}$$

The clip functions in Eq. (4) will be used as backtrace pointers in the dynamic programming algorithms, similarly to Johnson's algorithm on line graphs [15].

The *key observation* is the following one: While the derivatives $f_i' : \mathbb{R} \to \mathbb{R}$ are "infinite" objects, they have a finite representation, because they are *piecewise linear* (PWL) functions. (Because f_i is convex, f_i' is furthermore increasing.) This is seen by induction:

1. For a leaf node i, we minimize $f_i(x) = \frac{1}{2}\mu_i(x - y_i)^2$ which has the PWL derivative $f_i'(x) = \mu_i(x - y_i)$.
2. Assuming that all children's derivatives f_j' are PWL f_i' must also be PWL because the sum of PWLs is PWL, and clipping a PWL function again results in a PWL function.

Hence we can represent f_i' as an ordered sequence of knots where the slope s and intercept t of the PWL $x \mapsto sx + t$ changes. Both algorithms described below make use of this fact and the derivative recurrence in Eq. (4).

3.2 Exact Solver

The basic idea of the exact dynamic programming solver is to compute a suitable representation of the derivatives f_i'.

As all objective functions f_i, $i \in V$ are convex, the derivatives f_i' are monotonically increasing. This is why the "back-pointers" a_i and b_i with $f_i'(a_i) = -\lambda_i$ and $f_i'(b_i) = \lambda_i$ are of special interest: They define the interval $I_i := [a_i, b_i]$ of points x such that $f_i'(x)$ is not clipped, and, according to Lemma 1, the parent's value will be passed on (merged) later in the back-tracing step to determine the optimal value x_i^*. The *exact solver* first computes a complete representation of the derivative f_i' to determine the (exact) back-pointer $I_i = [a_i, b_i]$. As I_i suffices to compute the optimum x_i^*, the f_i' can be modified in place. In contrast, in the *approximation scheme*, we apply binary search to update an approximate back-pointer $[a_i^{(k)}, b_i^{(k)}]$ containing x_i^* at every iteration k.

Algorithm 1 shows the exact solver conceptually in pseudo-code; this is essentially Algorithm 2 from [16]. Each PWL derivative f_i' is represented by a priority queue containing information about its knots. There are two passes: first bottom-up (post-order) to compute back-pointers $[a_i, b_i]$ for each tree node i, then top-down (pre-order) to compute the optimal solution x^*.

Algorithm 1 TreeOpt. [16, Algorithm 2]

Input : Signal $y \in \mathbb{R}^n$, Weights $\mu \in \mathbb{R}^n$, $\lambda \in \mathbb{R}^n$, Tree T: Root r, Parent/child functions π, C
Output : Optimal solution $x^* \in \mathbb{R}^n$.

Initialize empty queues f_i' for each $i \in V$
Compute $\hat{\lambda}_i = \sum_{j \in C(i)} \lambda_i$ for each $i \in V$
for *each node i in post-order(T)* **do**
  $a_i \leftarrow \text{Clip}{\uparrow}\Big(f_i',\ \mu_i,\ -\mu_i y_i - \hat{\lambda}_i,\ -\lambda_i\Big)$
  $b_i \leftarrow \text{Clip}{\downarrow}\Big(f_i',\ \mu_i,\ -\mu_i y_i + \hat{\lambda}_i,\ +\lambda_i\Big)$
  $f_{\pi(i)}' \leftarrow f_{\pi(i)}' \cup f_i'$
$x_r^* \leftarrow \text{Clip}{\uparrow}\Big(f_r', \mu_r, -\mu_r y_r + \hat{\lambda}_r, 0\Big)$
for *each node i in pre-order(T)* **do**
  $x_i^* \leftarrow \text{clip}_{a_i}^{b_i}(x_{\pi(i)}^*)$

In terms of running time, most work is done in the Clip↑ function (Algorithm 2). In line 4 of Algorithm 1 we start with slope $s = \mu_i$ and intercept of $t = -\mu_i y_i - \hat{\lambda}_i$: This reflects the situations where the argument to the clip-function in Eq. (4) are when all summands $f'_j(x)$ evaluate to its minimum $-\lambda_j$. Then we step through the PWL f'_i, updating slope s and intercept t at every knot, until the position $\hat{x}$ is found where $f'_i(\hat{x}) = t_0 = s\hat{x} + t$.

Algorithm 2 Clip↑.

Input : Queue q, Slope s, Intercept t, Target t_0
Output : Position $\hat{x}$ (q is modified)
1 **while** q *not empty* **and** $s \times q.\min + t < t_0$ **do**
2 knot ← extract min from q
3 $s \leftarrow s +$ knot.s
4 $t \leftarrow t +$ knot.t
5 $\hat{x} \leftarrow (t_0 - t)/s$
6 Insert new knot with slope s, intercept t at position $\hat{x}$ into q

One has to be careful when processing latent nodes because in this case slope $s = 0$ can occur and Line 5 is undefined. In this case the solution of the current node is ambiguous and thus it is feasible to return $-\infty$ and not insert a new knot.

The algorithm for the counterpart Clip↓ works analogously: Iterate the loop in Line 1 while $s \times q.\max + t > t_0$ and subtract slope s and intercept t accordingly.

3.2.1 Double-Ended Mergable Queues

Algorithm 1 makes use of double-ended priority queues that are merged into the queues of their parent's queues (Line 6). For the theoretical worst-case running time the choice of the queue type defines the bottleneck.

▶ **Theorem 3.** *There is an $\mathcal{O}(n \log n)$ solver for fused lasso on trees.*

In their proof [16, Sec. 4.2] Kolmogorov et al. make use of two Fibonacci Heaps [11], a min-heap and a max-heap. The performance of Fibonacci Heaps in practical applications is controversially discussed [18, 5, 4]. Different types of double-ended priority queues as (Rank) Pairing Heaps [10, 24, 8, 13] or other sophisticated heap data structures are conceivable.

We believe that such complicated data structures are not necessary. Consider the algorithm processing node i: The decisive question for determining the running time is how many nodes are there in the queue in the forward pass when starting in Line 3? We found that the queues always contained not more than some hundred elements, independent of the number of nodes n.

▶ **Observation 4** (Queue Size). *The knot of a descendant $j \in V(T_i)$ is contained in the queue f'_i only if there exists an input $y_{\pi(i)}$ such that in the corresponding optimum $x^*(y)$ the nodes i and j are merged.*

Proof. According to Algorithm 1, nodes i and j can only be merged if none of the "back-pointers" $[a_k, b_k]$ on the way between them is clipped in Line 9. ◀

Note that a node j can be merged to i without having a knot in the priority queue f'_i, i. e. the observation gives only an upper bound for the number of elements in the queue.

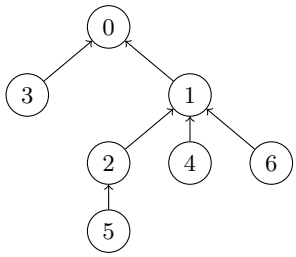


Figure 2 Queue layout: How to store the queues in memory such that every queue contains (the remaining) elements of its children's queues (separated by ":").

All in all we conjecture that in practice, the running time of the exact solver is more strongly affected by memory-related issues (e.g., cache effects) than by the actual queue data structure.

3.2.2 Implementation

We decided to replace the Fibonacci Heaps of [16], whose usage guarantees a $\mathcal{O}(n \log n)$ time algorithm, with simple plain sorted arrays. Sorting the queue in every step yields a (theoretically) suboptimal algorithm needing $\mathcal{O}(nq \log q)$ time, when q is an upper bound on the queue size. As the queue size could theoretically be $\Theta(n)$, we obtain an $\mathcal{O}\big(n^2 \log n\big)$ time algorithm in the worst case. However, our experiments show that in practice q behaves as a constant; giving an $\mathcal{O}(n)$ time algorithm in practice.

Observe that every node inserts exactly two knot elements, that define the new min and max, into the queue. This enclosing property makes it possible to pre-allocate the elements and store them in an order such that merging into the parent's queue (Line 6) is always possible and recently used elements are often in the cache (see Figure 2): For node i we allocate space for two elements at the position of its parenthesis in the parenthesis representation of the tree. To obtain the parenthesis form, walk the tree in depth-first search (DFS) [7]: Open a parenthesis when a node is discovered and close it when it is finished.

3.3 Approximation Scheme

The second algorithm we present is simpler but needs to be iterated to approximate the optimal solution. At every iteration k we refine for every node $i \in V$ the interval $[a_i^{(k)}, b_i^{(k)}]$, such that we can guarantee that the optimal solution x_i^* is contained.

The update is done by computing $\phi_i = f_i'(x_i)$ for a probing point $x_i \in [a_i^{(k)}, b_i^{(k)}]$: Recall that objective functions f_i are convex, hence f_i' are monotonically increasing. Because of Lemma 1 this means that if $\phi_i = f_i'(x_i) \geq -\lambda_i$ then $x_i^* \geq x$; analogously $\phi_i = f_i'(x_i) \leq +\lambda_i$ implies $x_i^* \leq x_i$. To compute ϕ_i for all $i \in V$ in linear time, we again apply the recursive Eq. (4). Algorithm 3 shows the pseudo-code.

It is best to set the next probing point as the middle point of the current interval, i. e. $x_i^{(k)} := (a_i^{(k)} + b_i^{(k)})/2$, such that

$$[a_i^{(k)}, b_i^{(k)}] = [x_i^{(k)} - \delta^{(k)}, x_i^{(k)} + \delta^{(k)}]$$

for some component-wise error $\delta^{(k)}$. So the next interval $[a_i^{(k+1)}, b_i^{(k+1)}]$ will be half the size $\delta^{(k+1)} = \delta^{(k)}/2$, independent of what happened in the loop in Lines 3–9.

Algorithm 3 TreeApx.

Input : Probe values $x_i \in I_i$, Bounds $I_i = [a_i, b_i]$ for $i \in V$; Tree T
Output : Improved bounds $[\hat{a}_i, \hat{b}_i] \subset [a_i, b_i]$
1 **for** *each node i in post-order(T)* **do**
2 $\quad \phi_i \leftarrow \mu_i(x_i - y_i) + \sum_{j \in C(i)} \operatorname{clip}_{-\lambda_j}^{+\lambda_j}[\phi_j]$
3 **for** *each node i in pre-order(T)* **do**
4 $\quad$ **if** $\phi_i \geq -\lambda_i$ **then**
5 $\quad\quad [\hat{a}_i, \hat{b}_i] \leftarrow [x_i, b_i]$
6 $\quad$ **else if** $\phi_i \leq +\lambda_i$ **then**
7 $\quad\quad [\hat{a}_i, \hat{b}_i] \leftarrow [a_i, x_i]$
8 $\quad$ **else**
9 $\quad\quad [\hat{a}_i, \hat{b}_i] \leftarrow [a_{\pi(i)}, b_{\pi(i)}]$

In the beginning, without apriori knowledge, we set δ_0 "big enough" such that it contains all possible solutions, e. g. $\delta_0 = \frac{1}{2}(\max_i y_i - \min_i y_i)$; further on we set the initial solutions all to the mean input signal $x_i^{(0)} = \overline{y}$. We obtain the following result.

▶ **Theorem 5.** *Algorithm 3 correctly updates the bounds $I_i^{(k)} = [a_i^{(k)}, b_i^{(k)}]$ such that after k iterations, one has $x_i^* \in I_i^{(k)}$ and $b_i^{(k)} - a_i^{(k)} = \delta_0\, 2^{-k}$ for every node $i \in V$.*

In other words, every interval I_i converges to the optimal solution x_i^* as $k \to \infty$. More precisely, to achieve component-wise accuracy of $\left\|x^* - x^{(k)}\right\|_\infty \leq \delta$ we need at least $k \geq \log_2 \delta_0 + \log_2 1/\delta$ iterations.

Engineering Memory Layout. Naively implemented, the approximation scheme (Algorithm 3) is not at all competitive to the exact solver (Algorithm 1): The reason is that walking a tree in pre- and post-order can cause a lot of memory cache misses as the next element is hard to predict for the hardware prefetcher.

To overcome this, we re-labeled the nodes such that memory layout is the pre-order (which is a reversed post-order). This way the next element will likely already be in the cache. The breadth first search (BFS) is a cache-friendly pre-order: Firstly the summands in Line 2 will all be adjacent in memory and secondly the parent's bounds in Line 9 will be accessed in a row for each child.

4 Experiments

We apply the exact algorithm TreeOpt and the $\mathcal{O}(n \log(1/\delta))$ approximation scheme TreeApx to differently structured tree datasets. The approximation is run for 20 iterations, for an an accuracy of $\delta = 2^{-20} < 10^{-6}$. In most practical applications probably fewer iterations are needed.

As there is no established library of test instances, we generate several ones. We use both randomly generated trees and random spanning trees of real-world datasets, as described in each section below. Table 1 gives an overview of the properties of different datasets.

For the sake of reproducibility the benchmarks are written as a Snakemake workflow [17] and included in the source code[2].

[2] `https://github.com/EQt/treelas/tree/master/data`

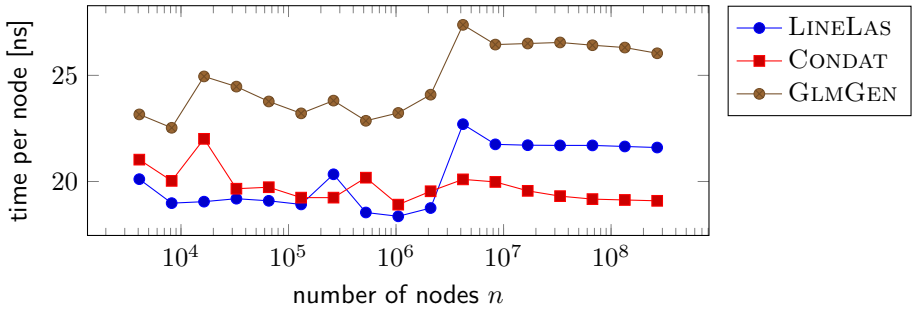


Figure 3 Running time per input node on line graphs of different size.

Table 1 Characteristics of the tree graphs used for experiments: Relevant for the experiments are the percentage of "leaf" nodes and the degree distribution $\deg_0$ of the non-leaf nodes; mean ± standard deviation are reported.

Name	n	leaf	mean $\deg_0$
Binary	100 000 000	50.0 %	3 ± 0
Europe	50 912 018	11.3 %	2.128 ± 0.367
HighDeg/LowDeg	50 000 000	37.2 %	2.592 ± 71.323
com-Orkut	3 072 441	48.6 %	2.944 ± 3.308
Phantom	1 000 000	33.6 %	2.506 ± 0.640

The benchmark system has an Intel® Core™ i7-5500U processor with cache sizes of 32 KB+32 KB (L1), 256 KB (L2) and 4096 KB (L3), and sufficient RAM (16 GB) to store each dataset plus auxiliary data structures in memory at all times. The code was compiled with GCC (`g++`) version 9.2.1, optimization level `-O3` with `-mtune=native`. All time measurements are averaged over 10 runs.

Input data $y = (y_i)$ is randomly drawn from a standard normal distribution $\mathcal{N}(0, 1)$ independently in each node. Node weights are set to $\mu_i = 1$ and edge weights are constant $\lambda_{ij} = \lambda$. To examine the effect of weak, intermediate and strong regularization, we run each algorithm with three different values $\lambda \in \{0.01, 0.1, 1\}$.

4.1 Line Graphs

For line graphs, efficient exact implementations exist. We here compare against the frequently used implementation GlmGen [1], which is algorithmically equivalent to our exact tree algorithm restricted to line graphs, and against an algorithm by Laurent Condat[3], an improvement of his earlier work [6], which is based on a dual formulation of the problem.

We measure running time per input node across a wide range of graph sizes between 5000 and 100 000 000 nodes. Overall, the time spent per node is in a narrow range between 18 and 28 nanoseconds, essentially independently of input size n. The small differences indicate that our implementation (LineLas) provides more optimization possibilities to the compiler than GlmGen, which uses the same algorithm. For large instances, Condat's algorithm is slightly faster. In 2016, Condat's implementation was considered to be the fastest one for line graphs [16]; for $n \leq 10^6$, our implementation is as fast.

[3] `https://lcondat.github.io/download/Condat_TV_1D_v2.c`; version v2, accessed January 2020

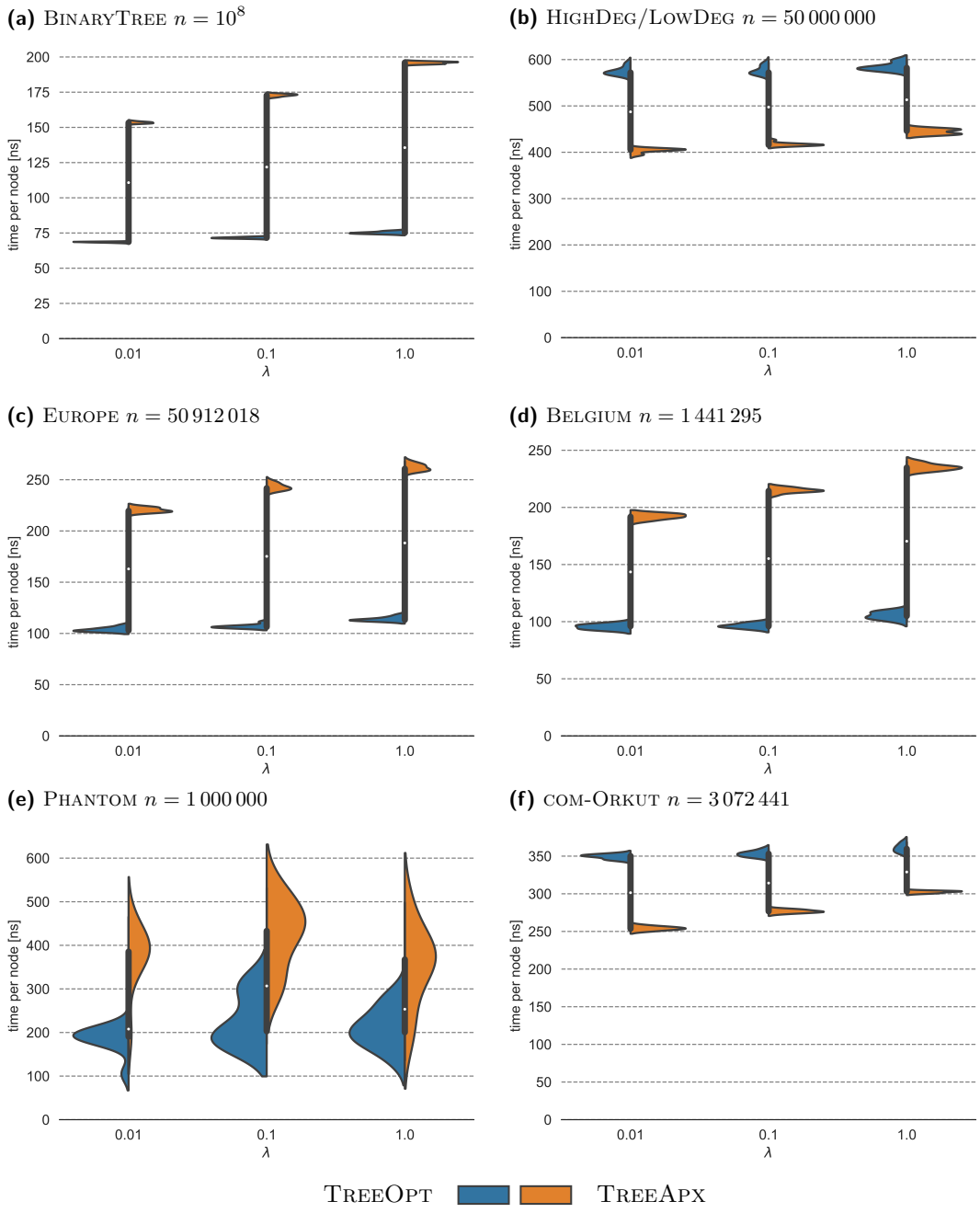


Figure 4 Violin plots of running times per node (nanoseconds) for the exact tree solver (blue) and the approximation scheme (orange) with $\delta = 2^{-20}$, averaged over 10 runs and 10 different random spanning trees when the data is a network.

4.2 Random Binary Trees

We started with a simple, regular tree: In a (full) binary tree every node has exactly two children except for the leaf nodes which constitute half of the nodes (see Table 1).

Neither algorithm is challenged by this instance (see Figure 4a): Even for $n = 10^8$ nodes the running time is roughly 4 times as slow as for a line graph.

4.3 Random High/Low Degree Trees

To challenge the exact solver, we included an instance HighDeg/LowDeg having an abundance of high-degree nodes because we expect a drop in performance, as the queue size obviously depends on the node degree. High degree nodes inevitably lead to more leaf nodes, i. e. low degree nodes. In the generated tree with $n = 5 \times 10^7$ nodes this is reflected in a relatively high standard deviation of the node degrees of 71 .

We generated the graph by means of a Prüfer sequence [21]: Thereto we find a sequence $S \in \{1, \ldots, n\}^{n-2}$ of length $n - 2$, whereby $S_i = j$ indicates that some node has parent j. From the Prüfer sequence we can reconstruct the tree in linear time [28]. We generated S as the sum of a normal distribution (high degree nodes), mixed with uniform distribution (some low degree nodes to make it more realistic):

$$S \sim |\mathcal{N}(0, 0.01)| + \mathcal{U}(1, n)$$

rounded to the closest integer in $\{1, \ldots, n\}$.

The results in Figure 4b show that indeed the approximation scheme performs better, as was expected.

4.4 Spanning Trees of Road Networks

There are a lot of applications of spatial data, like some noisy data on a map [26]. We include two graphs, Belgium (1.4 M nodes) and Europe (50 M nodes), generated out of OpenStreetMap data as provided by the "10th DIMACS Implementation Challenge: Street Networks" [2]. To obtain trees from the street graphs we computed 10 random spanning trees.

The experiments illustrate how efficiently the exact solver can process these low degree trees (see Figure 4c–4d).

4.5 Spanning Trees of Image Grid Graph

As a grid graph example we chose the Shepp–Logan Phantom [23] standard test image as shown in the introductory example (Figure 1). To be realistic as a model of a real photograph, we rendered the graph to $n = 1000 \times 1000$ nodes and added Gaussian noise of standard deviation $\sigma = 0.25$.

Interestingly the running time varies more than in the other cases (see Figure 4e). One explanation could be that this is the only instance that has a real input signal y.

4.6 Social Media Network

Cluster analysis becomes more important nowadays. Here we include the graph com-Orkut of a the free online social-media platform `www.orkut.com` which is part of the Stanford Network Analysis Project (SNAP) [19].

Although the tree with $n = 3\,072\,441$ nodes[4] is small, compared to the other graphs, the running time per node is larger as for e. g. EUROPE (see Figure 4f). With $\mathcal{O}(nq \log q)$ we would expect longer running times for larger trees. Again, the variety of node degrees (on average 2.94 ± 3.3) suits the approximation scheme better.

4.7 Summary of Observations

We see that either algorithm can be faster, depending on degree properties. As expected, the existence of nodes with high degree, i. e. high variation in node degrees, slows down the exact algorithm more than the approximation scheme. The number of nodes n is secondary for predicting the running time.

We further observe that the running times of both algorithms increase slightly with the weakening of regularization (i. e. increasing edge weight λ).

Compared to the times in Figure 3, we see that solving fused lasso on trees takes 5 times (PHANTOM) to 30 times (HIGHDEG/LOWDEG) longer, but the latter may be an unrealistic example in practice. On typical datasets, the factor is less than 20 (COM-ORKUT).

We found that about 40 % of the time of TREEAPX is used for initialization, i. e. to compute a child index and to permute the input memory. Without the memory re-ordering (re-labeling of the nodes), the running time of TREEAPX is about 10 to 20 times slower (excluding initialization).

In general both algorithms perform slightly slower with increasing tuning parameter λ: For TREEOPT this reflects Observation 4 that the sizes of the queues depends on the (potential) size of segments size. For TREEAPX the increase in running time stems from the fact that if i and $\pi(i)$ are separated in the solution, i. e. $x^*_i \neq x^*_{\pi(i)}$, the computation can be done independently.

5 Conclusion and Discussion

We presented the first implementations of fused lasso solvers on trees. We engineered both, an exact algorithm that runs in $\mathcal{O}(nq \log q)$ time and an approximation scheme that runs in $\mathcal{O}(n \log(1/\delta))$ time for a given target accuracy δ, such as $\delta = 2^{-20}$. Depending on the node degree distribution, either algorithm can be faster.

Instead of theoretically optimal data structures like Fibonacci heaps for the central priority queue in the algorithm, in practice we find that their length q is short on most natural datasets, and considerations about cache efficiency and memory layout are more important for the running time than asymptotic considerations.

Our next goal is to examine whether an iterative fused lasso solver for general graphs can be developed efficiently on the basis of one of the tree solvers presented here. Currently, general graph solvers partition the graph into overlapping paths (line graphs), solve many line graph problems and iteratively integrate the obtained solutions until they converge towards a global solution. It is reasonable to expect that by using spanning trees instead of random paths as subproblems, much fewer iterations may be necessary, which may lead to general graph solvers that converge much faster, even though each iteration on trees takes longer than an iteration on lines.

[4] Before processing the graph we had to exclude 185 nodes which are not connected to the largest connected component.

References

1 Taylor Arnold, Veeranjaneyulu Sadhanala, and Ryan J. Tibshirani. GLMGEN: Fast generalized lasso solver, 2019. URL: https://github.com/glmgen/glmgen.

2 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*, volume 588. American Mathematical Society, 2013.

3 Amir Beck and Marc Teboulle. Fast gradient-based algorithms for constrained total variation image denoising and deblurring problems. *Image Processing, IEEE Transactions on*, 18(11):2419–2434, November 2009. doi:10.1109/TIP.2009.2028250.

4 Gerth S. Brodal. Worst-case efficient priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, pages 52–58, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

5 Gerth S. Brodal. A survey on priority queues. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *Lecture Notes in Computer Science*, pages 150–163. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-40273-9_11.

6 Laurent Condat. A direct algorithm for 1-d total variation denoising. *Signal Processing Letters, IEEE*, 20(11):1054–1057, November 2013. doi:10.1109/LSP.2013.2278339.

7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

8 Amr Elmasry. Pairing heaps with costless meld. In Mark de Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010*, volume 6347 of *Lecture Notes in Computer Science*, pages 183–193. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-15781-3_16.

9 Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Distance transforms of sampled functions. *Theory of computing*, 8(1):415–428, 2012.

10 Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129, 1986. doi:10.1007/BF01840439.

11 Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

12 Alexandre Gramfort, Bertrand Thirion, and Gaël Varoquaux. Identifying predictive regions from fmri with tv-l1 prior. In *Pattern Recognition in Neuroimaging (PRNI), 2013 International Workshop on*, pages 17–20. IEEE, 2013.

13 Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. Rank-pairing heaps. *SIAM Journal on Computing*, 40(6):1463–1485, 2011. doi:10.1137/100785351.

14 Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – Seamless operability between C++11 and Python, 2017. URL: https://github.com/pybind/pybind11.

15 Nicholas Johnson. A dynamic programming algorithm for the fused lasso and L_0-segmentation. *Journal of Computational and Graphical Statistics*, 22(2):246–260, 2013. doi:10.1080/10618600.2012.681238.

16 Vladimir Kolmogorov, Thomas Pock, and Michal Rolinek. Total variation on a tree. *SIAM J. Imaging Sciences*, 9(2):605–636, 2016.

17 Johannes Köster and Sven Rahmann. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.

18 Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. A back-to-basics empirical study of priority queues. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 61–72. SIAM, 2014.

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

20 Oscar H. M. Padilla, James Sharpnack, and James G Scott. The DFS fused lasso: Linear-time denoising over general graphs. *The Journal of Machine Learning Research*, 18(1):6410–6445, 2017.

21 Heinz Prüfer. Neuer Beweis eines Satzes über Permutationen. *Arch. Math. Phys.*, 27:742–744, 1918.

22 Leonid I. Rudin. *Images, numerical analysis of singularities and shock filters*. Dissertation (Ph. D.), California Institute of Technology, 1987.

23 Lawrence A. Shepp and Benjamin F. Logan. The Fourier reconstruction of a head section. *IEEE Transactions on nuclear science*, 21(3):21–43, 1974.

24 John T. Stasko and Jeffrey S. Vitter. Pairing heaps: Experiments and analysis. *Commun. ACM*, 30(3):234–249, March 1987. doi:10.1145/214748.214759.

25 Wesley Tansey and Jeffrey G Scott. A fast and flexible algorithm for the graph-fused lasso, May 2015. arXiv:1505.06475.

26 Wesley Tansey, Jesse Thomason, and James G. Scott. Maximum-variance total variation denoising for interpretable spatial smoothing. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, 2018.

27 Robert Tibshirani, Michael Saunders, Saharon Rosset, Ji Zhu, and Keith Knight. Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistics Society: Series B*, 67(1):91–108, 2005. doi:10.1111/j.1467-9868.2005.00490.x.

28 Xiaodong Wang, Lei Wang, and Yingjie Wu. An optimal algorithm for Prufer codes. *Journal of Software Engineering and Applications*, 2(02):111, 2009.

29 Álvaro Barbero and Suvrit Sra. Modular proximal optimization for multidimensional total-variation regularization, 2014. arXiv:1411.0589.

Finding Structurally and Temporally Similar Trajectories in Graphs

Roberto Grossi
Dipartimento di Informatica, Università di Pisa, Italy
grossi@di.unipi.it

Andrea Marino
Dipartimento di Statistica, Informatica, Applicazioni "G. Parenti", Università di Firenze, Italy
andrea.marino@unifi.it

Shima Moghtasedi
Dipartimento di Informatica, Università di Pisa, Italy
shima.moghtasedi@di.unipi.it

Abstract

The analysis of similar motions in a network provides useful information for different applications like route recommendation. We are interested in algorithms to efficiently retrieve trajectories that are similar to a given query trajectory. For this task many studies have focused on extracting the geometrical information of trajectories. In this paper we investigate the properties of trajectories moving along the paths of a network. We provide a similarity function by making use of both the temporal aspect of trajectories and the structure of the underlying network. We propose an approximation technique that offers the top-k similar trajectories with respect to a query trajectory in an efficient way with acceptable precision. We investigate our method over real-world networks, and our experimental results show the effectiveness of the proposed method.

2012 ACM Subject Classification Information systems → Similarity measures; Information systems → Nearest-neighbor search

Keywords and phrases Graph trajectory, approximated similarity, top-k similarity query

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.24

Acknowledgements We are in debt with Ioanna Miliou for helping us with the Milan GPS dataset.

1 Introduction

Many papers in the literature have focused on extracting similarity information from sets of trajectories [1, 4, 3, 5, 7, 9, 16, 11, 12, 13, 14, 8, 15, 16]. Looking at the trajectories as sequences of nodes, the similarity among them can be related to the similarity among sequences. In this paper, we study how to retrieve similar trajectories constrained to follow paths in the graphs by taking into account both time and place. We aim at exploiting the topology of the network, assessing that two trajectories are similar if they pass through nearby nodes at roughly the same time. While there are measures such as the Fréchet distance for the plane, not much has been done for graphs. Indeed distances on the plane are fast to compute as we need to know just the coordinates of the points, so that measures taking into account both time and place can be computed in a reasonable time.

On the other hand, when trajectories are topologically constrained, as it happens in graphs, the distance computation between nodes is more challenging and the similarity function can turn easily into measures that are costly to compute. For this reason, the measures in the known literature that consider both time and place require that similar trajectories should pass through the same nodes. Table 1 summarizes the state of the art for the similarities of trajectories in graphs. As it can be noted, most of them require

Table 1 Graph-based similarity measures for trajectories of ℓ nodes.

	Temporal	Proximity	Properties	Input	Complexity time
[15]	×	×	Jaccard similarity based	Two strings as trajectories	$O(\ell^2)$
[16]	✓	×	Jaccard similarity based on the edit distance	Two strings as trajectories	$O(\ell^2)$
[8]	✓	✓	Pair to pair distance computation only at **specific predefined points**	Two trajectories with **the same length**, implicitly	$\boldsymbol{O(\ell)}$
[12]	✓	✓	Spatial and temporal distance computation in separate way (Liner combination)	Two trajectories with **the same length**	$\boldsymbol{O(\ell)}$
[11]	✓	✓	LCSS based Liner combination of spatial and temporal distance	Two trajectories	$O(\ell^2)$
[13]	✓	✓	Linear combination of spatial and temporal distance	A set of query points and a trajectory	$O(\ell^2)$
This paper	✓	✓	Temporal and spatial aspect of trajectories is combined in one single distance	Two trajectories with **different length**	$\boldsymbol{O(\ell)}$

quadratic time and those requiring linear time [1] have limitations: Hwang et al. [8] define the spatial-temporal similarity function so that two trajectories are similar if they pass through the same nodes at the same time. So they do not take into account the proximity of the trajectories and the closure in time of the intervals. Tikas et al. [12] remove this limitation, proposing a new similarity definition. However, their similarity works only when trajectories have the same length. As far as we know, there is no linear-time similarity that considers trajectories of different lengths and with a flexible notion of proximity in time and place.

In this paper, we show that it is possible to consider both time/structure for the trajectories on the graph, thus overcoming the limitations of the current literature and achieving a good compromise between time and precision. After defining our similarity function, we propose an algorithm to find the topmost k trajectories, having the highest similarity with respect to a given one. We do not put any constraint on the trajectories, and our similarity function can be computed in linear time once the pairwise node distances are given.

Our method is based on an indexing structure by using interval trees [2] to quickly find the top-k similar trajectories. To further speed up the computations, we propose an approximated similarity measure that shrinks the trajectories using the centers of the Voronoi diagram for graphs [6]. Shrinking using centers is natural as, for example, a long trajectory that goes along a road network can be reasonably represented with the most famous visited places. Similarly, the trajectories on the communication networks could be represented with respect to the nodes having more traffic.

We validate our measures and algorithms in the experimental part of the paper. Due to the lack of competitors, as there are no linear-time similarities on trajectories for graphs that use flexible proximity, we design a baseline approach to compare with. The approximation methods have a good precision while the time needed to answer the query reduces significantly.

[1] We are assuming for all of them that the pairwise distance among nodes is already given.

2 Preliminaries

Consider a network represented by a graph $G(V, E)$, where V is the set of nodes and E is the set of edges. Let $\mathcal{T}$ be a set of trajectories in G, where each trajectory is defined as follows. We say that two intervals $t_i = [s_i, e_i]$ and $t_{i+1} = [s_{i+1}, e_{i+1}]$, with integer endpoints, are *consecutive* if $s_i \leq e_i < s_{i+1} \leq e_{i+1}$ and $e_i + 1 = s_{i+1}$.

▶ **Definition 1.** *A trajectory $T \in \mathcal{T}$ is a sequence $T = \langle (v_1, t_1), (v_2, t_2), \cdots, (v_l, t_l) \rangle$ such that for each $1 \leq i \leq l-1$, we have that $(v_i, v_{i+1}) \in E$ and t_i and t_{i+1} are two consecutive time intervals. We call $|T| = l$ the length of T, which corresponds to the number of (non-distinct) nodes traversed by T. Letting $t_1 = [s_1, e_1]$ and $t_l = [s_l, e_l]$, we refer to s_1 and e_l as the starting time and ending time of T.*

Given a trajectory $T \in \mathcal{T}$, we denote by $t_i = [s_i, e_i]$ the i-th time interval of T. Note that a trajectory can pass through a node more than once.

For a trajectory T, let s and e be the starting and ending time of T. Given a time instant $i \in [s, e]$, the notation $T(i)$ indicates the unique node $u \in V$ such that there exists a pair $(u, t) \in T$ with $i \in t$. In the following, we use the standard notation for graphs. Given an undirected graph $G = (V, E)$, we denote by n and m the number of its nodes and edges. We denote by D_G the diameter of G and by $d(u, v)$ the shortest path distance between nodes u and v.

2.1 Trajectory Similarity Measure

This section is devoted to introducing our similarity function. It is arguably natural to assess that two trajectories are *similar* if they are close to each other without necessarily sharing common nodes or having the same length. Since the motion of trajectories in this paper is constrained by the network, the similarity function will use the proximity between trajectories. As the Euclidean distance is not appropriate to measure the distance between the nodes on the graph, it is important to use the graph distance metric instead. Therefore, the distance between nodes will be combined with the time intervals at which these have been traversed. In this sense, our similarity measure will consider both aspects of the trajectories: the *temporal* aspect and the location of trajectories over the graph, i.e the *structural* aspect.

In order to define the building blocks of our similarity measure, we need first to restrict trajectories within a time interval, as shown in the following definition.

▶ **Definition 2** (Time restricted Trajectory)**.** *Given a trajectory T and a time interval $t = [s, e]$, the time restricted trajectory $T[t]$ is the sequence of pairs $(v_i, t_i) \in T$ such that $t_i = [s_i, e_i]$ has overlap with $t = [s, e]$ (i.e. $t_i \cap t \neq 0$).*

Without loss of generality, we assume that $\sum_{(v_i, t_i) \in T[t]} |t_i| = |t|$. We define the distance between a *node* v and a *trajectory* T within a *time interval* t as follows:

$$dist(v, T, t) = \frac{\min_{(v_i, t_i) \in T[t]} d(v_i, v)}{D_G} \tag{1}$$

▶ **Proposition 3.** *The distance function $dist(v, T, t)$ is always in the interval $[0, 1]$.*

We observe that the extreme values are achieved in the following cases.

- $dist(v, T, t) = 0$ if and only if there exists at least one time instant $i \in t$ such that node $T(i) = v$.

- $dist(v, T, t) = 1$ if and only if for each time instant $i \in t$, node $T(i)$ is at distance D_G from v. This corresponds to the case where T spends the whole time interval t on nodes of G that are maximum distance from v.

We are now ready to introduce our similarity measure, using the distance function defined in Equation 1. Taking inspiration from [5], we aim at assigning a larger contribution to those parts of trajectories that are close for sufficiently long time intervals (while assigning lower contribution to farther parts). These desired properties are satisfied as follows.

▶ **Definition 4** (Similarity Function). *Given a query trajectory Q, a target trajectory T, and a time interval t, the similarity of T with respect to Q within t is*

$$Sim(Q, T, t) = \frac{\sum_{(v_i,t_i)\in Q[t]} |t_i| \times e^{-dist(v_i,T[t_i])}}{|t|} \tag{2}$$

▶ **Lemma 5.** *The similarity function $Sim(Q, T, t)$ is always in the interval $(0, 1]$.*

Proof. By Proposition 3, for each $(v_i, t_i) \in Q[t]$ we have that $0 \leq dist(v_i, T[t_i]) \leq 1$, and thus $1 \geq e^{-dist(v_i,T[t_i])} \geq e^{-1} > 0$. If we multiply by $|t_i|$, we get $0 < |t_i| \times e^{-dist(v_i,T[t_i])} \leq |t_i|$. By summation over each pair $(v_i, t_i) \in Q[t]$ we get

$$0 < \sum_{(v_i,t_i)\in Q[t]} |t_i| \times e^{-dist(v_i,T[t_i])} \leq \sum_{(v_i,t_i)\in Q[t]} |t_i| \tag{3}$$

By assuming $\sum_{(v_i,t_i)\in Q[t]} |t_i| = |t|$ and dividing Equation 3 by $|t|$ we get

$$0 < \frac{\sum_{(v_i,t_i)\in Q[t]} |t_i| \times e^{-dist(v_i,T[t_i])}}{|t|} \leq 1. \qquad \blacktriangleleft$$

It is worth remarking that whenever $Q[t] = T[t]$ we have $Sim(Q, T, t) = 1$, where $Q[t] = T[t]$ means that for each $i \in t$ we have $Q(i) = T(i)$.

▶ **Lemma 6.** *Given two trajectories Q and T, and a time interval t, where $|Q[t]| = \ell_1$ and $|T[t]| = \ell_2$, computing $Sim(Q, T, t)$ requires $O(\ell_1 + \ell_2)$ time and pairwise node distances.*

Proof. Looking at equations (1) and (2), it seems that $O(\ell_1 \times \ell_2)$ time is needed. The cost is instead $O(\ell_1 + \ell_2)$ if we realize that the computation is conceptually a nested loop in which the nodes in Q and T are scanned forward when a pairwise distance $d(v_i, v)$ is needed: in each iteration at least one node is scanned, thus the total cost is $O(\ell_1 + \ell_2)$. ◀

2.2 Top-k Most Similar Trajectories

We define the problem of retrieving, in a given set of trajectories $\mathcal{T}$, the top-k similar trajectories to a query in a specific time interval. More formally, this desires a set of trajectories, referred to as k-MsTraj, corresponds to the following one.

▶ **Definition 7** (k-Most Similar Trajectory (k-MsTraj)). *Given a set of trajectories $\mathcal{T}$, a query trajectory Q, and a query time interval t, let k-MsTraj be the set $\mathcal{T}' \subseteq \mathcal{T}$ with $|\mathcal{T}'| = k$, such that $Sim(Q, S, t) \geq Sim(Q, T, t)$ for each trajectory $S \in \mathcal{T}'$ and $T \in \mathcal{T} - \mathcal{T}'$.*

To present a good intuition of our proposed similarity function, we provide an illustrative example of four trajectories that are randomly chosen from a dataset of trajectories moving in Milan (see Section 5.1 for more details about this dataset). By the similarity function in Definition 4, using the red trajectory in Figure 1 as query trajectory, the green trajectory

has the maximum similarity among all the others, meaning that it is a solution for the k-MsTraj problem with $k = 1$. Note that the trajectories having color red, green, yellow and violet in Figure 1, start to move at time instances (in msec) 37237, 45964, 57354 and 26430, and stop the movement at 582313, 331565, 57872 and 564740, respectively.

Figure 1 An example including 4 random trajectories in a dataset of trajectories moving in Milan. The trajectory with the red color is a query. The green trajectory is the most similar one by Definition 4.

A straightforward approach to find k-MsTraj is to compute the similarity score for each trajectory $T \in \mathcal{T}$ and Q, reporting the k trajectories with maximum scores. Clearly we only consider those trajectories that are defined for all instants $i \in t$. This approach is inefficient as it requires $O(|\mathcal{T}| \times \max\{|y|_{max}, |Q|\})$ shortest path (precomputed) distances, where $|\mathcal{T}|$ and $|y|_{max}$ are respectively the number of trajectories and the maximum length of trajectories in $\mathcal{T}$.

In the next two sections, we discuss how to accelerate this method and how to estimate similarities.

3 Baseline: Exact Computation of k-MsTraj

In this section, we introduce a baseline method to solve exactly the k-MsTraj problem. This is based on an indexing phase, described in Section 3.1, which aims at accelerating the query processing, as described in Section 3.2. We will use this method as a baseline in the experimental evaluation of our proposed methods.

3.1 NTrajI Indexing

The *Neighborhood Trajectory Indexing* (NTrajI) described here efficiently finds the closest trajectories with respect to each node of the query and its corresponding time interval.

We use an interval tree, which is a binary tree storing a set of intervals based on the median of the endpoints of the intervals. In this structure, all the intervals that intersect the median point are stored in the root of the tree. The intervals lying completely to the left and the right of the median point are, respectively, stored in the left subtree and the right subtree of the root. The subtrees are constructed recursively in the same way. By using this structure, we are able to find efficiently all intervals that overlap with any given interval or point using the following well-known result.

▶ **Theorem 8** ([2]). *Given a set of n intervals, an interval tree uses $O(n)$ space, can be built in $O(n \cdot \log n)$ time and can report all intervals that overlap a query interval or point in $O(\log n + k)$ time, where k is the number of reported intervals.*

In NTrajI, we build an Interval Tree IT_u for each node $u \in V$. Each IT_u stores the time intervals of the trajectories in $\mathcal{T}$ spent in either u or the neighbors of u, and maintains the corresponding trajectories. Specifically, we have the following.

▶ **Definition 9** (Node Projection Set). *The projection set S_u of a node u stores the pairs (t, T) of all trajectories $T \in \mathcal{T}$ that pass through the nodes in $\{u\} \cup N(u)$ during interval t, namely, $S_u = \{(t, T) \mid (v, t) \in T$ and $v \in \{u\} \cup N(u)$ and $T \in \mathcal{T}\}$, where $N(u)$ is the set of neighbors of u in the graph.*

The Interval Tree IT_u maintains all the pairs $(t, T) \in S_u$ for each node $u \in V$. Each entry of IT_u is of the form $\langle t, id \rangle$, where id is the trajectory identifier and t is the time interval spent in $\{u\} \cup N(u)$ by the trajectory id. Note that there can be more than one pair associated with node u and the same trajectory id, since each trajectory can traverse u multiple times.

By Theorem 8, we can derive that NTrajI uses $O(|\mathcal{T}| \times |y|_{max} \times \Delta)$ space, where Δ denotes the maximum degree of G. For a given node u and a time interval t, let $\Gamma_{(u,t)}$ denote the trajectories that traverse either u or $N(u)$ within t.By searching over the NTrajI, we are able to find $\Gamma_{(u,t)}$ efficiently by taking $O(\log |\Gamma_u| + |\Gamma_{(u,t)}|)$ time, where $|\Gamma_u|$ is the size of IT_u and $|\Gamma_{(u,t)}|$ is the number of reported trajectories.

Algorithm 1 Baseline.

Input: Graph G, set of trajectories $\mathcal{T}$, query trajectory Q, time interval $t = [a, b]$, integer k
Result: Top-k trajectories k-MsTraj

```
Heap H
for each (v_i, t_i) ∈ Q[t] do
    Γ_(v_i,t_i) ← NTrajI-search(v_i, t_i)
end for
Γ = ⋃_{(v_i,t_i)∈Q_t} Γ_(v_i,t_i)
H ← the Sim scores for all trajectories in Γ
k-MsTraj ← top-k trajectories in H
```

3.2 Query Processing

We introduce a pruning technique as the baseline method to the k-MsTraj problem (see Algorithm 1). The baseline method explores the set Γ of trajectories that are most promising to be k-MsTraj. They are discovered by searching through the NTrajI index.

▶ **Proposition 10.** *By construction, k-MsTraj $\subseteq \Gamma \subseteq \mathcal{T}$.*

By exploiting Γ, the baseline method computes the similarity score of each trajectory in Γ and finds the trajectories having the highest similarity with Q within the time interval t.

Therefore, the main task is to construct the *candidate set* Γ using NTrajI. In particular, for a given query trajectory Q, we first restrict query Q within the time query t, obtaining $Q[t]$. Then, for each $(v_i, t_i) \in Q[t]$, we aim at finding the trajectories that are close to v_i within

t_i. To this aim, for each $(v_i, t_i) \in Q[t]$, we search through IT_{v_i}, by NTrajI-search(v_i, t_i) in Algorithm 1, to build $\Gamma_{(v_i,t_i)}$. So we have $\Gamma = \bigcup_{(v_i,t_i)\in Q[t]} \Gamma_{(v_i,t_i)}$. We compute the similarity score with respect to the function in Definition 4, for each trajectory in Γ. In order to maintain the k trajectory ids with the highest similarity score during the search process, we use a heap H, whose top-k entries represent k-MsTraj.

4 Approximated Computation of k-MsTraj

The baseline method described in Section 3 is costly when the number of trajectories in $\mathcal{T}$ is large. To accelerate the searching process, we propose some approximated methods with two-phase preprocessing as discussed in Section 4.1.

- First, we partition G into disjoint groups of nodes, precomputing the distances among the centers of each group. As the similarity function in Definition 4 uses the shortest path distance between nodes of the graph, we aim at approximating distances inside the graph using the distances from the centers of the groups.
- Second, we adapt the NTrajI indexing so that we maintain trajectories among the groups in a structure called VoTrajI.

For the query processing, given a query trajectory, we show how to estimate the similarity scores for the trajectories in $\mathcal{T}$ using the partitioning and the new VoTrajI index in Section 4.2

4.1 Two-phase Preprocessing

The partitioning takes into account node popularity by choosing the nodes having a higher number of trajectories passing through them as the centers of the groups. To do this, it uses Voronoi Diagrams for graphs (VDG), as explained next.

The VDG is a generalization of the classic Voronoi diagram. For graph $G = (V, E)$ and a set of trajectory $\mathcal{T}$, let $\mathcal{C} = \{c_1, c_2, \dots c_h\}$ be a set of h (most popular) nodes in V, called Voronoi sites i.e. center nodes. The VDG over the nodes in $\mathcal{C}$ is defined as a partition of V into h groups $g_1, g_2, \cdots, g_h$, one for each center in $\mathcal{C}$. Node $u \in V$ is in group g_i with center c_i (i.e. $g_i.\mathcal{C} = c_i$) iff $d(u, c_i) \leq d(u, c_j)$ for each $c_j \in \mathcal{C}$ with $i \neq j$ (ties are broken arbitrarily). We can divide G into h Voronoi groups in $O(n \log n)$ time when $h = O(n^\epsilon)$ for a positive constant $\epsilon < 1$ [6].

Once the Voronoi groups have been computed, we precompute and store the pairwise distances among the center nodes of the groups. Our aim is using these distances as an approximation for the distances required by the similarity function in Definition 4. By running one BFS for each center node, we compute the distance between each pair $c_i, c_j \in \mathcal{C}$ in $O(m \cdot n^{1/2})$ time, where we set $h = n^{1/2}$. As a result, we obtain the following lemma.

▶ **Lemma 11.** *Graph partitioning and centers distance precomputation require $O(m \cdot n^{1/2})$ time. The space required by the centers distance table is $O(n)$ space.*

We now discuss how to build the *Voronoi Trajectory Indexing* (VoTrajI) by adapting the NTrajI data structure. We use an interval tree IT_c for each $c \in \mathcal{C}$. Interval tree IT_c stores the time intervals of trajectories in $\mathcal{T}$ spent within the nodes in g, when $g.\mathcal{C} = c$. The VoTrajI maintains the corresponding trajectory ids of the time intervals. By modifying Definition 9, we have:

▶ **Definition 12** (Group Projection Set). *The projection set S_c of a center node $c \in \mathcal{C}$ stores the pairs (t, T) of all trajectories $T \in \mathcal{T}$ that pass through the nodes in g, when $g.\mathcal{C} = c$, namely, $S_c = \{(t, T) \mid (v, t) \in T$ and $v \in g$ and $g.\mathcal{C} = c$ and $T \in \mathcal{T}\}$.*

The Interval Tree IT_c for each node $c \in \mathcal{C}$ maintains all pairs $(t, T) \in S_c$. Each entry of IT_c is the form of $\langle t, id\rangle$, where id is the trajectory id, and t is the time interval that the trajectory id spent at $v \in g$, where $g.\mathcal{C} = c$.

To reduce the storage space used by IT_c, for each $c \in \mathcal{C}$, we consider a sequence of consecutive time intervals with the same trajectory id in S_c as a single time interval with the corresponding trajectory id.

▶ **Lemma 13.** *The two-phase preprocessing takes $O(m \cdot n^{1/2})$ time and $O(n)$ space.*

4.2 Query Processing

Consider how a trajectory $T \in \mathcal{T}$ is represented with respect to the center nodes of the Voronoi diagram. Let $(v_i, t_i) \in T$ and $v_i \in g$, where g is a Voronoi group of G. We represent $(v_i, t_i) \in T$ as (c, t_i) where $g.\mathcal{C} = c$. We obtain a new trajectory T' as a sequence of center nodes and the corresponding time intervals. Note that T' can traverse a sequence of the nodes belonging to the same Voronoi group within consecutive time intervals. To avoid the duplication of nodes for consecutive time intervals, we define *shrunk trajectories*.

Given a trajectory $T' = \langle (c_1, t_1), \ldots, (c_l, t_l)\rangle$, consider the operator $\textsc{shrink}(T')$, which recursively merges any pair $(c_i, t_i), (c_{i+1}, t_{i+1}) \in T'$ as $(c_i, t_i + t_{i+1})$ when $c_i = c_{i+1}$ and t_i, t_{i+1} are two consecutive time intervals (here operation $t_i + t_{i+1}$ gives $[s_i, e_{i+1}]$).

▶ **Definition 14** (Shrunk Trajectory). *Let $T = \langle (v_1, t_1), \ldots, (v_l, t_l)\rangle$ be a trajectory in $\mathcal{T}$. Consider the corresponding sequence $T' = \langle (c_1, t_1), \ldots, (c_l, t_l)\rangle$ with respect to the Voronoi groups. We define the shrunk trajectory of T as $\hat{T} = \textsc{shrink}(T')$.*

Note that it takes $O(l)$ time to obtain $\hat{T}$, and that $|\hat{T}| \leq |T|$. At this point, we consider two variants for estimating k-MSTraj.

SHQ Shrunk Query: Shrinking trajectory Q during query time.

SHQT Shrunk Query and Target: Shrinking each trajectory in $\mathcal{T}$ during the preprocessing and shrinking trajectory Q during query time.

Both variants perform a search on the VoTrajI index using the shrunk query trajectory $\hat{Q}$. The outcome of that search is a set $\tilde{\Gamma}$, which is defined as Γ in Section 3, except that we use VoTrajI in place of NTrajI. This makes a difference, as the property in Proposition 10 does not necessarily hold anymore. Indeed there could be a trajectory $T \in$ k-MsTraj such that $T \notin \tilde{\Gamma}$ (whereas surely $T \in \Gamma$). This approximated version has the advantage of speed, which motivates this study.

4.2.1 Variant SHQ

In this variant we compute the similarity scores for each trajectory $T \in \tilde{\Gamma}$ with respect to the shrunk query $\hat{Q}$. In particular, we make an estimate of $Sim(Q, T, t)$ as $Sim(\hat{Q}, T, t)$, and report the top-k trajectories with the highest estimated similarity score. To measure the precision ratio of this estimation, the similarity function makes an estimate of $d = d(v, u)$ as $\bar{d} = d(c, u)$, when $v \in Q$, $u \in T$, and c is the center node of a group that includes v.

▶ **Lemma 15.** *For any given $v, u \in V$, $u \neq v$, the ratio between $d = dist(v, u)$ and $\bar{d} = dist(c, u)$, where c is the center of Voronoi group containing v, is bounded as $1 \leq d/\bar{d} \leq 3$.*

Proof. Let $dist(c, v) = r$. We have two possibilities. If $r \leq 2\bar{d}$, then by triangle inequality $d \leq r + \bar{d}$ and thus $d \leq 3\bar{d}$. Else if $2\bar{d} < r$, then by triangle inequality $r \leq d + \bar{d}$ and thus $\bar{d} < d$. ◀

Although we reduce the number of nodes in the query trajectory which needs to be processed, the number of distances involved is still large. As mentioned earlier, the cost of distance computation depends on the length of the trajectories within the query time interval t. In order to reduce this cost, we consider our second variant SHQT.

4.2.2 Variant SHQT

In this variant, we estimate $Sim(Q, T, t)$ as $Sim(\hat{Q}, \hat{T}, t)$. Specifically, the similarity function makes an estimate of $d = d(v, u)$ as $\tilde{d} = d(c_i, c_j)$, when $v \in Q$, $u \in T$, and c_i, c_j are the center nodes of the groups that include $v \in Q$ and $u \in T$, respectively.

▶ **Lemma 16.** *For any two distinct nodes v, u belonging to Voronoi groups with center nodes c_i, c_j, respectively, the ratio between $\tilde{d} = dist(c_i, c_j)$ and $\bar{d} = dist(c_i, u)$ is bounded as $\tilde{d}/\bar{d} \leq 2$.*

Proof. As illustrated in Figure 2, let $dist(v, c_i) = r$ and $dist(u, c_j) = r'$. We consider the groups g_i, g_j containing the two centers c_i, c_j, respectively. By triangle inequality we have $\tilde{d} \leq r' + \bar{d}$. Since $u \in g_j$ and $u \notin g_i$ then $r' \leq \bar{d}$. Thus, $\tilde{d} \leq 2\bar{d}$. ◀

Using Lemma 15 and 16, we are able to conclude that $\tilde{d} \leq 2d$ when $r > 2\bar{d}$. If $r > 2\bar{d}$ by triangle inequality we have $\bar{d} < d$. Since $\tilde{d} \leq 2\bar{d}$, we can obtain that $\tilde{d} \leq 2d$.

The similarity function in Definition 4 assigns a larger contribution to those nodes of the trajectories that are closer to each other, rather than the farther ones. Thus, by Lemma 15 and 16, we expect that the estimated similarity score in both variants is larger than the exact similarity score. We will evaluate this idea in our experiments.

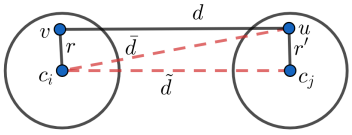


Figure 2 $c_i \in g_i$ and $c_j \in g_j$ such that $c_i, c_j \in \mathcal{C}$.

Table 2 Summary of Datasets. Recall that D_G is the diameter of G.

Dataset Name	trajectories	nodes	edges	D_G
Facebook	1000	4039	88234	8
Milan	16166	3000	130071	5
Rome	7755	473	10524	6

5 Experimental Evaluation

This section is devoted to comparing the performances of SHQ and SHQT with respect to the baseline method, hereafter called BASE. The evaluation aims at providing a response to the following questions.

Q1: How fast is getting the answer for a query, i.e. how much is the query time?

Q2: How fast is the preprocessing phase?

Q3: How good is the quality of the solution found if compared with the exact solution?

Table 3 The average time for answering a query for each method (in sec).

Datasets	BASE	SHQ	SHQT
Facebook	1.09	0.74	0.43
Milan	380.03	376.15	85.48
Rome	26.19	19.42	15.83

Table 4 The average number of trajectories in the candidate set in each method. For the Facebook dataset, refer to Figure 4.

Datasets	BASE	SHQ	SHQT
Milan	9786.39	9968.98	9968.98
Rome	7504.37	6569.84	6569.84

We will respond to these questions by evaluating the performance of each method for different value of k. In particular, we set k as 2^i for $i = 0, \ldots, 6$. Each experiment requires a graph, a trajectory set, and a query trajectory. For each experiment, we choose 100 trajectories as query trajectories, randomly.

Our computing platform is a machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40GHz, 24 virtual cores, 128 Gb RAM, running Ubuntu Linux version 4.4.0-22-generic. The source code has been written in `Python3`.

5.1 Datasets

We conduct our experiments on real-world graphs, using synthetic and real trajectories, whose properties are shown in Table 2.

Facebook Synthetic trajectories from Facebook social network. [2]

Milan Dataset based on GPS tracks of private cars in Milan. First, we build the graph by making use of the GPS trajectories. Then, we cluster the close nodes with k-Means. There is an edge between two clusters i and j if there exists at least one trajectory going through i, j, consecutively. [3]

Rome Dataset of Flickr geo-tagged photos provided by [10], containing tourist trajectories covering Rome. We build the graph of the Points of Interest (in short, PoIs).

5.2 Query Time

In the following, we compare the query time of the three methods. Table 3 reports our results, showing the average query time over 100 queries in each dataset. As it can be seen, both SHQ and SHQT variants outperform BASE. The most evident benefit can be seen for the biggest dataset that we considered, i.e. Milan dataset. In this case, SHQT spends less than 23% of the time needed by BASE and SHQ.

We report in Table 4 the number of candidate trajectories for all the methods (i.e. $|\Gamma|$ and $|\tilde{\Gamma}|$). The table shows that SHQ and SHQT select more candidates than BASE. This is not an issue as, even if we have to process these extra trajectories, they compute much fewer query distances and consecutively spend less time than BASE, as shown in Table 3.

For the sake of completeness, we also analyzed the behavior of our method when the length of the queried trajectories varies for the case of the Facebook dataset. As we can see in Figure 3, SHQ and SHQT outperform BASE. Actually SHQT significantly outperforms the other two, and the improvement becomes even more evident when the length of the query trajectory increases. As the length of the query goes up to 80, the time needed by BASE

[2] `https://snap.stanford.edu/data/ego-Facebook.html`
[3] `https://sobigdata.d4science.org/catalogue-sobigdata?path=/dataset/gps_track_milan_italy`

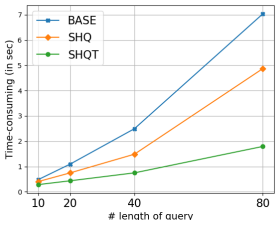


Figure 3 Average time vs length of queried trajectories in Facebook dataset.

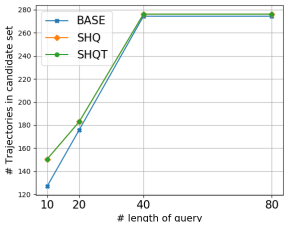


Figure 4 The average number of trajectories in candidate set in each method vs length of query in Facebook dataset. For both SHQ and SHQT the candidate set $\tilde{\Gamma}$ is the same.

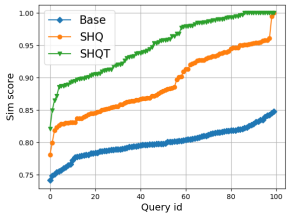


Figure 5 The comparison between similarity scores of each method.

and SHQ increases faster than the SHQT. This observation confirms that shrinking both target and query trajectories reduces the number of distance computations and thus the time reduced greatly.

Figure 4 shows the average number of trajectories in the candidate set for each method. Note that the number of trajectories in the candidate set for both methods SHQ and SHQT is the same since we use the same approach for specifying $\tilde{\Gamma}$. As it can be seen, by increasing the length of the query up to 40, the number of trajectories in the candidate set for each method increases quickly to more than 240, and then becomes the same for all of them. This confirms the role of the precomputed distances among Voronoi centers to accelerate query processing. The time cost of this precomputation is negligible. Moreover, by Lemmas 15 and 16, we would expect that the similarity score would be larger when we shrink trajectories: looking at Figure 5, we observe that the similarity scores by shrinking trajectories behave as we expected.

5.3 Preprocessing Time

As we mentioned before, between SHQT and SHQ, only the former uses the precomputed distances. The cost of the precomputation is shown in Table 5.We also report the time needed to index and shrink trajectories. In particular, columns NTrajI and VoTrajI report the time needed for building respectively the indexing structures NTrajI and VoTrajI. As expected, the time needed for building NTrajI over the trajectories in the Facebook dataset is larger than the one required by other datasets. This is due to the presence of longer trajectories with respect to other datasets. The column Distance Precomputing shows that the time needed to precompute the distances is negligible. Finally, we can observe that the time needed to perform the Voronoi partitioning and shrinking trajectories in the last column is also negligible in comparison with the time needed for building NTrajI, which clearly dominates the cost.

5.4 Quality and Evaluation Metrics

We evaluate the quality of the solution produced by the SHQ and SHQT methods with respect to BASE. The effectiveness of the methods is assessed by means of the metrics that we describe next, where the values close to 1 are more desirable. Let γ_1 and γ_2 be two output sets containing top-k trajectories, e.g. the exact and approximated solutions.

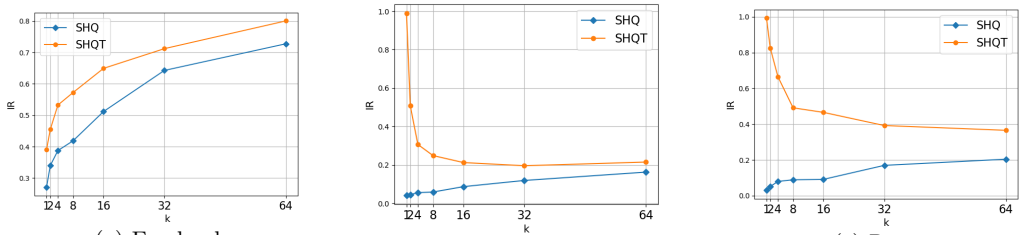
Table 5 Preprocessing time (in sec).

Dataset	NTrajI	VoTrajI	Distance precomputing	Shrinking trajectories and building Voronoi diagram
Facebook	185.19	0.51	0.48	0.62
Milan	1716.44	13.50	0.27	10.21
Rome	69.25	0.24	0.005	0.56

1. We define the *similarity score ratio* as the ratio of the average similarity scores of trajectories in γ_1 and γ_2. Namely, $\text{SSR}(\gamma_1, \gamma_2) = \frac{\sum_{T \in \gamma_1} Sim(Q,T,t)}{\sum_{S \in \gamma_2} Sim(Q,S,t)}$.
2. We define the *intersection ratio* as $\text{IR}(\gamma_1, \gamma_2) = \frac{|\gamma_1 \cap \gamma_2|}{k}$.

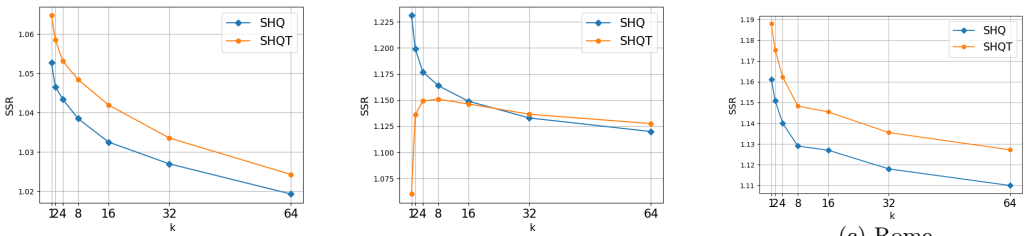
It is worth remarking that the running time of the methods does not change for the different values of k.

Our results are shown in Figures 6 and 7, where the IR and SSR ratios are reported as a function of k. In particular, Figure 6 represents the IR ratio for increasing k values in the three datasets. The IR ratio goes up to more than 0.80 quickly, by increasing the value of k on the Facebook network. On the other hand, this value in Milan and Rome networks becomes close to 0.3. However, we observe that the lower values of IR correspond to SSR values that are close to 1. Indeed, Figure 7 shows SSR which is almost always very close to 1 and that gets more close to 1, while increasing k.



(a) Facebook (b) Milan (c) Rome

Figure 6 The quality of the results returned by the competitors in terms of IR ratio vs k.



(a) Facebook (b) Milan (c) Rome

Figure 7 The quality of the results returned by the competitors in terms of SSR ratio vs k.

References

1 Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *International conference on foundations of data organization and algorithms*, pages 69–84. Springer, 1993.

2 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational geometry: algorithms and applications*. Springer-Verlag TELOS, 2008.

3 Lei Chen and Raymond Ng. On the marriage of Lp-norms and edit distance. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 792–803. VLDB Endowment, 2004.

4 Lei Chen, M Tamer Özsu, and Vincent Oria. Robust and fast similarity search for moving object trajectories. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 491–502. ACM, 2005.

5 Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, Yu Zheng, and Xing Xie. Searching trajectories by locations: an efficiency study. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 255–266. ACM, 2010.

6 Martin Erwig. The graph Voronoi diagram with applications. *Networks: An International Journal*, 36(3):156–163, 2000.

7 Christos Faloutsos, Mudumbai Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of ACM SIGMOD*, pages 419–429, Minneapolis, MN, 1994.

8 Jung-Rae Hwang, Hye-Young Kang, and Ki-Joune Li. Searching for similar trajectories on road networks using spatio-temporal similarity. In *East European Conference on Advances in Databases and Information Systems*, pages 282–295. Springer, 2006.

9 Eamonn Keogh and Chotirat Ann Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and information systems*, 7(3):358–386, 2005.

10 Cristina Ioana Muntean, Franco Maria Nardini, Fabrizio Silvestri, and Ranieri Baraglia. On learning prediction models for tourists paths. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 7(1):1–34, 2015.

11 Shuo Shang, Ruogu Ding, Kai Zheng, Christian S Jensen, Panos Kalnis, and Xiaofang Zhou. Personalized trajectory matching in spatial networks. *The VLDB Journal*, 23(3):449–468, 2014.

12 Eleftherios Tiakas, Apostolos N Papadopoulos, Alexandros Nanopoulos, Yannis Manolopoulos, Dragan Stojanovic, and Slobodanka Djordjevic-Kajan. Trajectory similarity search in spatial networks. In *null*, pages 185–192. IEEE, 2006.

13 Eleftherios Tiakas and Dimitrios Rafailidis. Scalable trajectory similarity search based on locations in spatial networks. In *Model and Data Engineering*, pages 213–224. Springer, 2015.

14 Michail Vlachos, George Kollios, and Dimitrios Gunopulos. Discovering similar multidimensional trajectories. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 673–684. IEEE, 2002.

15 Jung-Im Won, Sang-Wook Kim, Ji-Haeng Baek, and Junghoon Lee. Trajectory clustering in road network environment. In *Computational Intelligence and Data Mining, 2009. CIDM'09. IEEE Symposium on*, pages 299–305. IEEE, 2009.

16 Ying Xia, Guo-Yin Wang, Xu Zhang, Gyoung-Bae Kim, and Hae-Young Bae. Spatio-temporal similarity measure for network constrained trajectory data. *International Journal of Computational Intelligence Systems*, 4(5):1070–1079, 2011.

Zipping Segment Trees

Lukas Barth
Karlsruhe Institute of Technology, Germany
lukas.barth@kit.edu

Dorothea Wagner
Karlsruhe Institute of Technology, Germany
dorothea.wagner@kit.edu

Abstract

Stabbing queries in sets of intervals are usually answered using segment trees. A dynamic variant of segment trees has been presented by van Kreveld and Overmars [14], which uses red-black trees to do rebalancing operations. This paper presents *zipping segment trees* – dynamic segment trees based on zip trees, which were recently introduced by Tarjan et al. [13]. To facilitate zipping segment trees, we show how to uphold certain segment tree properties during the operations of a zip tree. We present an in-depth experimental evaluation and comparison of dynamic segment trees based on red-black trees, weight-balanced trees and several variants of the novel zipping segment trees. Our results indicate that zipping segment trees perform better than rotation-based alternatives.

2012 ACM Subject Classification Theory of computation → Sorting and searching; Information systems → Point lookups

Keywords and phrases Segment Trees, Dynamic Segment Trees, Zip Trees, Data Structures

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.25

Related Version A full version of the paper is available at `https://arxiv.org/abs/2004.03206`.

Supplementary Material `https://github.com/tinloaf/ygg/releases/tag/version_sea2020`

Funding *Lukas Barth*: This work was partially funded by the Helmholtz Program "Enery System Design", Topic 2 "Digitalization and System Technology".

1 Introduction

A common task in computational geometry, but also many other fields of application, is the storage and efficient retrieval of segments (or more abstractly: intervals). The question of which data structure to use is usually guided by the nature of the retrieval operations, and whether the data structure must be dynamic, i.e., support updates. One very common retrieval operation is that of a *stabbing query*, which can be formulated as follows: Given a set of intervals on $\mathbb{R}$ and a query point $x \in \mathbb{R}$, report all intervals that contain x.

For the static case, a *segment tree* is the data structure of choice for this task. It supports stabbing queries in $\mathcal{O}(\log n)$ time (with n being the number of intervals). Segment trees were originally introduced by Bentley [3]. While the segment tree is a static data structure, i.e., is built once and would have to be rebuilt from scratch to change the contained intervals, van Kreveld and Overmars present a dynamic version [14], called *dynamic segment tree* (DST).

Dynamic segment trees are applied in many fields. Solving problems from computational geometry certainly is the most frequent application, for example for route planning based on geometry [7] or labeling rotating maps [8]. However, DSTs are also useful in other fields, for example internet routing [4] or scheduling algorithms [1].

In this paper, we present an adaption of dynamic segment trees, so-called *zipping segment trees*. Our main contribution is replacing the usual red-black-tree base of dynamic segment trees with zip trees, a novel form of balancing binary search trees introduced recently by

Tarjan et al. [13]. On a conceptual level, basing dynamic segment trees on zip trees yields an elegant and simple variant of dynamic segment trees. Only few additions to the zip tree's rebalancing methods are necessary. On a practical level, we can show that zipping segment trees outperform dynamic segment trees based on red-black trees in our experimental setting.

2 Preliminaries

A concept we need for zip trees are the two *spines* of a (sub-) tree. We also talk about the spines of a node, by which we mean the spines of the tree rooted in the respective node. The *left spine* of a subtree is the path from the tree's root to the previous (compared to the root, in tree order) node. Note that if the root (call it v) is not the overall smallest node, the left spine exits the root left, and then always follows the right child, i.e., it looks like $(v, L(v), R(L(v)), R(R(L(v))), \dots)$. Conversely, the *right spine* is the path from the root node to the next node compared to the root node. Note that this definition differs from the definition of a spine by Tarjan et al. [13].

2.1 Union-Copy Data Structure

Dynamic segment trees in general carry annotations of sets of intervals at their vertices or edges. These set annotations must be stored and updated somehow. To achieve the run times in [14], van Kreveld and Overmars introduce the *union-copy* data structure to manage such sets. Sketching this data structure would be out of scope for this paper. It is constructed by intricately nesting two different types of union-find data structures: a textbook union-find data structure using union-by-rank and path compression (see for example Seidel and Sharir [11]) and the $UF(i)$ data structure by La Poutré [9].

For this paper, we just assume this union-copy data structure to manage sets of items. It offers the following operations[1]:

createSet() Creates a new empty set in $\mathcal{O}(1)$.

deleteSet() Deletes a set in $\mathcal{O}(1 + k \cdot F_N(n))$, where k is the number of elements in the set, and $F(n)$ is the time the *find* operation takes in one of the chosen union-find structures.

copySet(A) Creates a new set that is a copy of A in $\mathcal{O}(1)$.

unionSets(A,B) Creates a new set that contains all items that are in A or B, in $\mathcal{O}(1)$.

createItem(X) Creates a new set containing only the (new) item X in $\mathcal{O}(1)$.

deleteItem(X) Deletes X from *all* sets in $\mathcal{O}(1 + k)$, where k is the number of sets X is in.

2.2 Dynamic Segment Trees

This section recapitulates the workings of dynamic segment trees as presented by van Kreveld and Overmars [14] and outlines some extensions. Before we describe the dynamic segment tree, we briefly describe a classic static segment tree and the *segment tree property*. For a more thorough description, see de Berg et al. [5, 10.2]. Segment trees store a set $\mathcal{I}$ of n intervals. Let $x_1, x_2, \dots x_{2n}$ be the ordered sequence of interval end points in $\mathcal{I}$. For the sake of clarity and ease of presentation, we assume that all interval borders are distinct, i.e., $x_i > x_{i+1}$. We also assume all intervals to be closed. Lifting these two restrictions is straightforward.

[1] The data structure presented by Kreveld and Overmars provides more operations, but the ones mentioned here are sufficient for this paper.

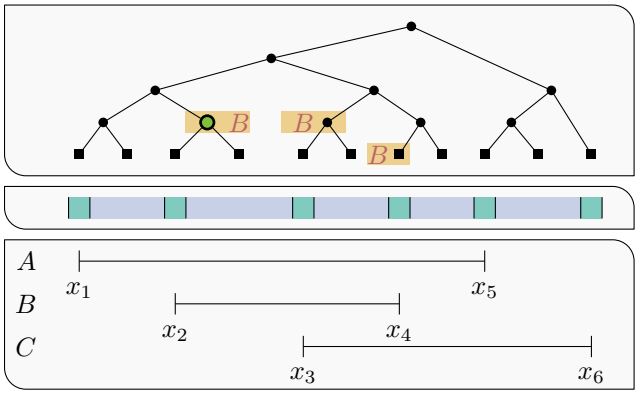


Figure 1 A segment tree (top) for three intervals (bottom). The middle shows the elementary intervals. Note that the green intervals do actually contain just one point and are only drawn fat so that they can be seen. The nodes marked with B are the nodes that carry the annotation for interval B.

In the first step, we forget whether an x_i is a start or an end of an interval. The intervals

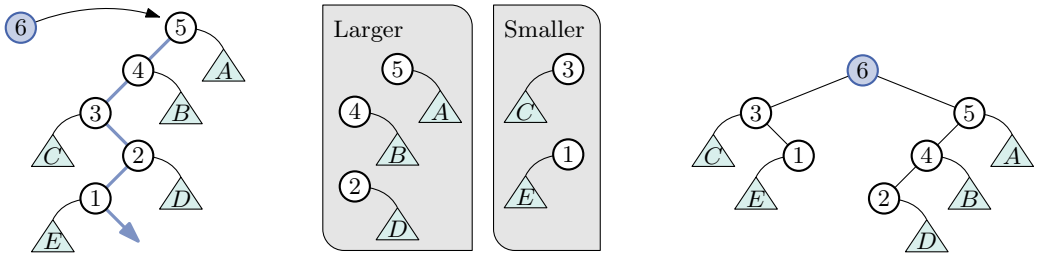
$$(-\infty, x_1), [x_1, x_1], (x_1, x_2), [x_2, x_2], \dots (x_{2n-1}, x_{2n}), [x_{2n}, x_{2n}], (x_{2n}, \infty)$$

are called the *elementary intervals* of $\mathcal{I}$. To create a segment tree, we create a leaf node for every elementary interval. On top of these leaves, we create a binary tree. The exact method of creating the binary tree is not important, but it should adhere to some balancing guarantee to provide asymptotically logarithmic depths of all leaves.

Such a segment tree is outlined in Figure 1. The lower box indicates the three stored intervals and their end points $x_1, \dots x_6$. The middle box contains a visualization of the elementary intervals, where the green intervals are the $[x_i, x_i]$ intervals (note that while of course they should have no area, we have drawn them "fat" to make them visible) while the blue intervals are the (x_i, x_{i+1}) intervals. The top box contains the resulting segment tree, with the square nodes being the leaf nodes corresponding to the elementary intervals, and the circular nodes being the inner nodes.

We associate each inner node v with the union of all the intervals corresponding to the leaves in the subtree below v. In Figure 1, that means that the larger green inner node is associated with the intervals $[x_2, x_3)$, i. e., the union of $[x_2, x_2]$ and (x_2, x_3), which are the two leaves beneath it. Recall that a segment tree should support fast stabbing queries, i.e., for any query point q, should report which intervals contain q. To this end, we annotate the nodes of the tree with sets of intervals. For any interval I, we annotate I at every node v such that the associated interval of v is completely contained in I, but the associated interval of v's parent is not. In Figure 1, the annotations for B are shown. For example, consider the larger green node. Again, its associated interval is $[x_2, x_3)$, which is completely contained in $B = [x_2, x_4]$. However, its parent is associated with $[x_1, x_3)$, which is not contained in B. Thus, the large green node is annotated with B.

A segment tree constructed in such a way is semi-dynamic. Segments cannot be removed, and new segments can be inserted only if their end points are already end points of intervals in I. To provide a fully dynamic data structure with the same properties, van Kreveld and Overmars present the dynamic segment tree [14]. It relaxes the property that intervals are always annotated on the topmost nodes the associated intervals of which are still completely contained in the respective interval. Instead, they propose the *weak segment tree property*:



(a) Before unzipping. **(b)** Unzipped parts. **(c)** After reassembly.

Figure 2 Illustration of the process of unzipping a path in zip trees. Nodes' names are simultaneously their ranks. Node keys are not shown.

For any point q and any interval I that contains q, the search path of q in the segment tree contains exactly one node that is annotated with I. For any q and any interval J that does not contain q, no node on the search path of q is annotated with J. Thus, collecting all annotations along the search path of q yields the desired result, all intervals that contain q. It is easy to see that this property is true for segment trees: For any interval I that contains q, some node on the search path for q must be the first node the associated interval of which does not fully contain I. This node contains an annotation for I.

Dynamic segment trees also remove the distinction between leaf nodes and inner nodes. In a dynamic segment tree, every node represents an interval border. To insert a new interval, we insert two nodes representing its borders into the tree, adding annotations as necessary. To delete an interval, we remove its associated nodes. If the dynamic segment tree is based on a classic red-black tree, both operations require rotations to rebalance. Performing such a rotation without adapting the annotations would cause the weak segment tree property to be violated. Also, the nodes removed when deleting an interval might have carried annotations, which also potentially violates the weak segment tree property.

We must thus fix the weak segment tree property during rotations. We must also make sure that any deleted node does not carry any annotations, and we must specify how we add annotations when inserting new intervals.

3 Zipping Segment Trees

In Section 2.2 we have described a variant of the dynamic segment trees introduced by van Kreveld and Overmars [14]. These are built on top of a balancing binary search tree, for which van Kreveld and Overmars suggested using red-black trees. The presented technique is able to uphold the weak segment tree property during the red-black tree's operations: node rotations, node swaps, leaf deletion and deletion of vertices of degree one. These are comparatively many operations that must be adapted to dynamic segment trees. Also, each each operation incurs a run time cost for the repairing of the weak segment tree property.

Thus it stands to reason to look at different underlying trees which either reduce the number of necessary balancing operations. One such data structure are *zip trees* introduced by Tarjan et al. [13]. Instead of inserting nodes at the bottom of the tree and then rotating the tree as necessary to achieve balance, these trees determine the height of the node to be inserted before inserting it in a randomized fashion by drawing a *rank*. The zip tree then forms a *treap*, a combination of a search tree and a heap: While the key of $L(v)$ (resp. $R(v)$) must always be smaller or equal (resp. larger) to the key of v, the ranks of both $L(v)$ and $R(v)$ must also be smaller or equal to the rank of v. Thus, any search path always sees

nodes' ranks in a monotonically decreasing sequence. The ranks are chosen randomly in such a way that we expect the result to be a balanced tree. In a balanced binary tree, half of the nodes will be leaves. Thus, we assign rank 0 with probability 1/2. A fourth of the nodes in a balanced binary tree are in the second-to-bottom layer, thus we assign rank 1 with probability 1/4. In general, we assign rank k with probability $(1/2)^{k+1}$, i.e., the ranks follow a geometric distribution with mean 1. With this, Tarjan et al. show that the expected length of search paths is in $\mathcal{O}(\log n)$, thus the tree is expected to be balanced.

Zip trees do not insert nodes at the bottom or swap nodes down into a leaf before deletion. If nodes are to be inserted into or removed from the middle of a tree, other operations than rotations are necessary. For zip trees, these operations are *zipping* and *unzipping*. In the remainder of this section, we examine these two operations of zip trees separately and explain how to adapt them to preserve the weak segment tree property. For a more thorough description of the zip tree procedures, we refer the reader to [13].

3.1 Insertion and Unzipping

Figure 2 illustrates the unzipping operation that is used when inserting a node. Note that we use the numbers 1 through 6 as nodes' names as well as their ranks in this example. The triangles labeled A through E represent further subtrees. The node to be inserted is ⑥, the fat blue path is its search path (i.e., its key is smaller than the keys of ⑤, ④ and ②, but larger than the keys of ③ and ①). Since ⑥ has the largest rank in this example, the new node needs to become the new root. To this end, we unzip the search path, splitting it into the parts that are – in terms of nodes' keys – larger than ⑥ and parts that are smaller than ⑥. In other words: We group the nodes on the search path by whether we exited them to the left (a larger node) or to the right (a smaller node). Algorithm 1, when ignoring the highlighted parts, provides pseudocode for the unzipping operation.

We remove all edges on the search path (Step 1 in Algorithm 1). The result is depicted in the two gray boxes in Figure 2b: several disconnected parts that are either larger or smaller than the newly inserted node. Taking the new node ⑥ as the new root, we now reassemble these parts below it. The smaller parts go into the left subtree of ⑥, stringed together as each others' right children (Step 3 in Algorithm 1). Note that all nodes in the "smaller" set must have an empty right subtree, because that is where the original search path exited them – just as nodes in the "larger" set have empty left subtrees. The larger parts go into the right subtree of ⑥, stringed together as each others' left children. This concludes the unzipping operation, yielding the result shown in Figure 2c. With careful implementation, the whole operation can be performed during a single traversal of the search path.

To insert a segment into a dynamic segment tree, we need to do two things: First, we must correctly update annotations whenever a segment is inserted. Second, we must ensure that the tree's unzipping operation preserves the weak segment tree property.

We will not go into detail on how to achieve step one. In fact, we add new segments in basically the same fashion as red-black-tree based DSTs do. We first insert the two nodes representing the segment's start and end. Take the path between the two new nodes. The nodes on this path are the nodes at which a static segment tree would carry the annotation of the new segment. Thus, annotating these nodes (resp. the appropriate edges) repairs the weak segment tree property for the new segment.

In the remainder of this section, we explain how to adapt the unzipping operations of zip trees to repair the weak segment property. Let the annotation of an edge e before unzipping be $S(e)$, and let the annotation after unzipping be $S'(e)$. As an example how to fix the annotations after unzipping, consider in Figure 2 a search path that descends into subtree D before unzipping. It picks up the annotations on the unzipped path from ⑤ up to ②, i.e., $S\big(\vec{L}(5)\big)$, $S\big(\vec{L}(4)\big)$, $S\big(\vec{R}(3)\big)$, and on the edge going into D, i.e., $S\big(\vec{R}(2)\big)$. After unzipping, it

Algorithm 1 Unzipping routine. This inserts *new* into the tree at the position currently occupied by v by first disassembling the search path below v, and then reassembling the different parts as left and right spines below *new*. The highlighted parts are used to repair the dynamic segment tree's annotations. Note that in an efficient implementation, one would interleave all four steps.

```
   Input : new: Node to be inserted
   Input : v: Node to be replaced by new
 1 cur ← v;
 2 oldParent ← P(v);
 3 smaller ← newList();
 4 larger ← newList();
 5 collected ← createSet();
   /* Step 1:  Remove edges along search path.                          */
 6 while cur ≠ ⊥ do
 7     if new < cur then
 8         larger.append(cur);
 9         next ← L(cur);
10         S(R⃗(cur)) ← unionSets(S(R⃗(cur)), collected);
11         collected ← unionSets(collected, S(L⃗(cur)));
12         next ← L(cur);
13         L(cur) ← ⊥;
14         cur ← next;
15     else
           /* Omitted, symmetric to the case new < cur. Collect parts in
              smaller.                                                  */
   /* Step 2:  Insert new.                                             */
16 if L(oldParent) = v then
17     L(oldParent) ← new;
18 else
19     R(oldParent) ← new;
   /* Step 3:  Reassemble left spine from parts smaller than new       */
20 parent ← new;
21 for n ∈ smaller do
22     if parent = new then
23         L(parent) ← n;
24         deleteSet(S(L⃗(parent))); S(L⃗(parent)) ← createSet();
25     else
26         R(parent) ← n;
27         deleteSet(S(R⃗(parent))); S(R⃗(parent)) ← createSet();
   /* Step 4:  Reassemble right spine.  This is symmetric to the left
      spine and thus omitted.                                          */
```

picks up the annotations on all the new edges on the path from ⑥ to ② plus $S'(\vec{R}(2))$. We set the annotations on all newly inserted edges to $\emptyset$ after unzipping. Thus, we need to add the annotations before unzipping, i.e., $S(\vec{L}(5)) \cup S(\vec{L}(4)) \cup S(\vec{R}(3))$, to the edge going into D. We therefore set $S'(\vec{R}(2)) = S(\vec{R}(2)) \cup S(\vec{L}(5)) \cup S(\vec{L}(4)) \cup S(\vec{R}(3))$ after unzipping.

In Algorithm 1, the blue highlighted parts are responsible for repairing the annotations. While descending the search path to be unzipped, we incrementally collect all annotations we see on this search path (line 11), and at every visited node add the previously collected annotations to the other edge (line 10), i. e., the edge that is not on the search path. By setting the annotations of all newly created edges to the empty set (lines 24 and 27), we make sure that after reassembly, every search path descending into one of the subtrees attached to the reassembled parts picks up the same annotations on the edge into that subtree as it would have picked up on the path before disassembly.

3.2 Deletion and Zipping

Deleting segments again is a two-staged challenge: We need to remove the deleted segment from all annotations, and must make sure that the *zipping* operation employed for node deletion in zip trees upholds the weak segment tree property. Removing a segment from all annotations is trivial when using the union-copy data structure outlined in Section 2.1: The *deleteItem()* method does exactly this.

We now outline the *zipping* procedure and how it can be amended to repair the weak segment tree property. Zipping two paths in the tree works in reverse to unzipping. Pseudocode is given in Algorithm 2. Again, the pseudocode without the highlighted parts is the pseudocode for plain zipping, independent of any dynamic segment tree. Assume that in the situation Figure 2c, we want to remove ⑥, thus we want to arrive at the situation in Figure 2a. The zipping operation consist of walking down the left spine (consisting of ③ and ① in the example) and the right spine (consisting of ⑤, ④ and ② in the example) simultaneously and zipping both into a single path. This is done by the loop in line 7. At every point during the walk, we have a current node on both spines, call it the *current left* node l and the *current right* node r. Also, there is a *current parent* p, which is the bottom of the new zipped path being built. In the beginning, the current parent is the parent of the node being removed.[2] In each step, we select the current node with the smaller rank, breaking ties arbitrarily (line 8). Without loss of generality, assume the current right node is chosen (the branch starting in line 20). We attach the chosen node to the bottom of the zipped path (p), and then r itself becomes p. Also, we walk further down on the right spine.

Note that the choice whether to attach left or right to the bottom of the zipped path (made via *attachRight* in Algorithm 2) is made in such a way that the position in which we attach previously was part of one of the two spines being zipped. For example, if p came from the right spine, we attach left to it. However, $\vec{L}(p)$ comes from the right spine. This method of attaching nodes always upholds the search tree property: When we make a node from the right spine the new parent (line 29), we know that the new p is currently the largest remaining nodes on the spines. We always attach left to this p (line 31). Since all other nodes on the spine are smaller than p, this is valid. The same argument holds for the left spine.

We now explain how the edge annotations can be repaired so that the weak segment tree property is upheld. Assume that for an edge e, $S(e)$ is the annotation of e before zipping, and $S'(e)$ is the annotation of e after zipping. Again, we argue via the subtrees that search paths can descend into. A search path descending into a subtree on the right of a node on the right spine, e.g., subtree B attached to ④ in Figure 2c, will before zipping pick up the annotation on the right edge of the node being removed plus all annotations on the spine up to the respective node, e.g., $S\big(\vec{R}(6)\big) \cup S\big(\vec{L}(5)\big)$, before descending into the respective subtree (B in the example). To preserve these picked up annotations, we again push them down onto the edge that actually leads away from the spine into the respective subtree.

[2] If the root is being removed, pretend there is a pseudonode above the root.

Algorithm 2 Zipping routine. This removes v from the tree, zipping the left and right spines of v. The highlighted parts are used to repair the dynamic segment tree's annotations.

Input : n: Node to be removed

$l \leftarrow L(n)$; // Current node descending v's left spine
$r \leftarrow R(n)$; // Current node descending v's right spine
$p \leftarrow P(n)$; // Bottom of the partially zipped path
$attachRight \leftarrow R(P(n)) = v$;
$collected_l = \text{copySet}(S(\vec{L}(n)))$;
$collected_r = \text{copySet}(S(\vec{R}(n)))$;
while $l \neq \bot \vee r \neq \bot$ **do**
  if $(l \neq \bot) \wedge ((r = \bot) \vee (rank(l) > rank(r)))$ **then**
    if $attachRight$ **then**
      $R(p) \leftarrow l$;
      $\text{deleteSet}(S(\vec{R}(p)))$; $S(\vec{R}(p)) \leftarrow \text{createSet}()$;
    else
      $L(p) \leftarrow l$;
      $\text{deleteSet}(S(\vec{L}(p)))$; $S(\vec{L}(p)) \leftarrow \text{createSet}()$;
    $S(\vec{L}(l)) \leftarrow \text{unionSets}(S(\vec{L}(l)), collected_l)$;
    $collected_l \leftarrow \text{unionSets}(collected_l, S(\vec{R}(l)))$;
    $p \leftarrow l$;
    $l \leftarrow R(l)$;
    $attachRight \leftarrow \text{true}$;
  else
    if $attachRight$ **then**
      $R(p) \leftarrow r$;
      $\text{deleteSet}(S(\vec{R}(p)))$; $S(\vec{R}(p)) \leftarrow \text{createSet}()$;
    else
      $L(p) \leftarrow r$;
      $\text{deleteSet}(S(\vec{L}(p)))$; $S(\vec{L}(p)) \leftarrow \text{createSet}()$;
    $S(\vec{R}(r)) \leftarrow \text{unionSets}(S(\vec{R}(r)), collected_r)$;
    $collected_r \leftarrow \text{unionSets}(collected_r, S(\vec{L}(r)))$;
    $p \leftarrow r$;
    $r \leftarrow L(r)$;
    $attachRight \leftarrow \text{false}$;

Formally, during zipping, we keep two sets of annotations, one per spine. In Algorithm 2, these are $collected_l$ and $collected_r$, respectively. Let n be the node to be removed. Initially, we set $collected_l = S\big(\vec{L}(n)\big)$ and $collected_r = S\big(\vec{R}(n)\big)$. Then, whenever we pick a node c from the left (resp. right) spine as new parent, we set $S'\big(\vec{L}(c)\big) = S\big(\vec{L}(c)\big) \cup collected_l$ (resp. $S'\big(\vec{R}(c)\big) = S\big(\vec{R}(c)\big) \cup collected_r$). This pushes down everything we have collected to the edge leading away from the spine at c. Then, we set $collected_l = collected_l \cup S\big(\vec{R}(c)\big)$ and $S'\big(\vec{R}(c)\big) = \emptyset$ (resp. $collected_r = collected_r \cup S\big(\vec{L}(c)\big)$ and $S'\big(\vec{L}(c)\big) = \emptyset$). This concludes the techniques necessary to use zip trees as a basis for dynamic segment trees, yielding *zipping segment trees*.

3.3 Complexity

Zip trees are randomized data structures, therefore all bounds on run times are expected bounds. In [13, Theorem 4], Tarjan et al. claim that the expected number of pointers changed during a zip or unzip is in $\mathcal{O}(1)$. However, they actually even show the stronger claim that the number of nodes on the zipped (or unzipped) paths is in $\mathcal{O}(1)$. Observe that the loops in lines 6 and 21 of Algorithm 1 as well as line 7 of Algorithm 2 are executed at most once per node on the unzipping (resp. zipping) path. Inside each of the loops, a constant number of calls are made to each of the *copySet*, *createSet*, *deleteSet* and *unionSets* operations. Thus, the rebalancing operations incur expected constant effort plus a constant number of calls to the union-copy data structure.

When inserting a new segment, we add it to the sets annotated at every vertex along the path between the two nodes representing the segment borders. Since the depth of every node is expected logarithmic in n, this incurs expected $\ln(n)$ calls to *unionSets*. The deletion of a segment from all annotations costs exactly one call to *deleteItem*.

All operations but *deleteSet* and *deleteItem* are in $\mathcal{O}(1)$ if the union-copy data structure is appropriately built. The analysis for the two deletion functions is more complicated and involves amortization. The rough idea is that every non-deletion operation can increase the size of the union-copy's representation only by a limited amount. On the other hand, the two deletion operations each decrease the representation size proportionally to their run time.

The red-black-tree-based DSTs by van Kreveld and Overmars [14] also need $\Omega(\ln n)$ calls to *copySet* during the insertion operation, and at least a constant number of calls during tree rebalancing and deletion. Therefore, for every operation on zipping segment trees, the (expected) number of calls to the union-copy data structure's functions is no larger than the number of calls in the red-black-tree-based implementation and we achieve the same (but only expected) run time guarantees, which are $\mathcal{O}(\log n)$ for insertion, $\mathcal{O}(\log n \cdot a(i, n))$ for deletion (with $a(i, n)$ being the row-inverse of the Ackermann function, for some constant i) and $\mathcal{O}(\log n + k)$ for stabbing queries, where k is the number of reported segments.

3.4 Generating Ranks

Nodes' ranks play a major role in the rebalancing operations of zip trees. In Section 3, we already motivated why nodes' ranks should follow a geometric distribution with mean 1; it is the distribution of the node depths in a perfectly balanced tree.

A practical implementation needs to somehow generate these values. The obvious implementation would be to somehow generate a (pseudo-) random number and determine the position of the first 1 in its binary representation. The rank generated in this way is then stored at the respective node.

Storing the rank at the node can be avoided if the rank is generated in a reproducible fashion. Tarjan et al. [12] already point out that one can "compute it as a pseudo-random function of the node (or of its key) each time it is needed." In fact, the idea already appeared earlier in the work by Seidel and Aragon [10] on treaps. They suggest evaluating a degree 7 polynomial with randomly chosen coefficients at the (numerical representation of) the node's key. However, the 8-wise independence of the random variables generated by this technique is not sufficient to uphold the theoretical guarantees given by Tarjan et al. [12].

However, without any theoretical guarantees, a simpler method for reproducible ranks can be achieved by employing simple hashing algorithms. Note that even if applying universal hashing, we do not get a guarantee regarding the probability distribution for the values of individual bits of the hash values. However, in practice, we expect it to yield results similar

to true randomness. As a fast hashing method, we suggest using the $2/m$-almost-universal multiply-shift method from Dietzfelbinger et al. [6]. Since we are interested in generating an entire machine word in which we then search for the first bit set to 1, we can skip the "shift" part, and the whole process collapses into a simple multiplication.

4 Experimental Evaluation of Dynamic Segment Trees Bases

In this section, we experimentally evaluate zipping segment trees as well as dynamic segment trees based on two of the most prominent rotation-based balanced binary search trees: red-black trees and weight-balanced trees. Weight-balanced trees require a parametrization of their rebalancing operation. In [2], we perform an in-depth engineering of weight-balanced trees. For this analysis of dynamic segment trees, we pick only the two most promising variants of weight-balanced trees: top-down weight-balanced trees with $\langle \Delta, \Gamma \rangle = \langle 3, 2 \rangle$ and top-down weight-balanced trees with $\langle \Delta, \Gamma \rangle = \langle 2, 3/2 \rangle$.

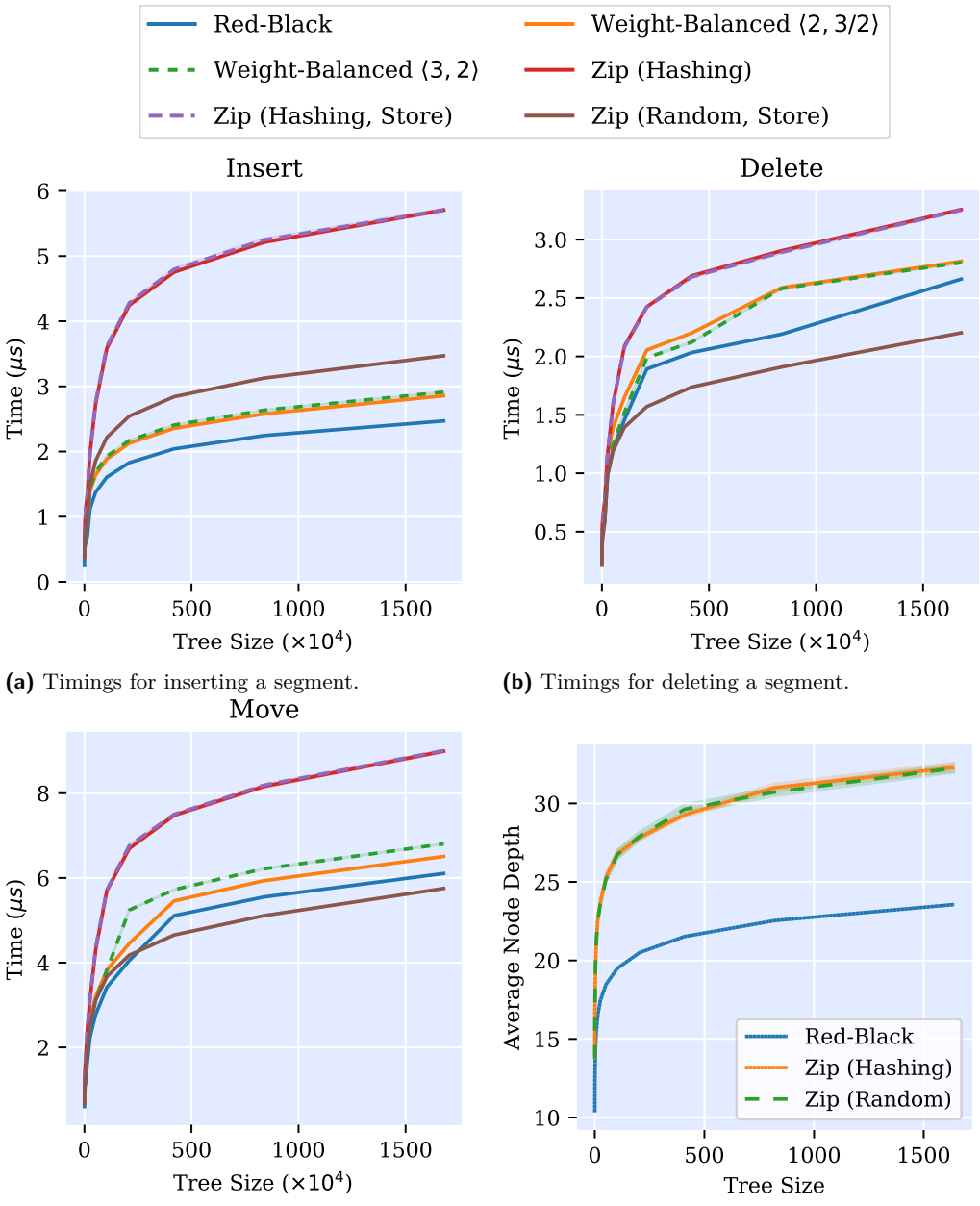
Note that since we are only interested in the performance effects of the trees underlying the DST, and not in the performance of an implementation of the complex union-copy data structure, we have implemented a simplified variant of DSTs which only reports the aggregate value of weighted segments at a stabbing query, instead of a list of the respective segments. See the full version of this paper for details. Evaluating the performance of the union-copy data structure is out of scope of this work.

For the zip trees, we choose a total of three variants, based on the choices explained in Section 3.4: The first variant, denoted *Hashing*, generates nodes' ranks by applying the fast hashing scheme by Dietzfelbinger et al. [6] to the nodes' memory addresses. In this variant, node ranks are not stored at the nodes but re-computed on the fly when they are needed. The second variant, denoted *Hashing, Store* also generates nodes' ranks from the same hashing scheme, but stores ranks at the nodes. The last variant, denoted *Random, Store* generates nodes' ranks independent of the nodes and stores the ranks at the nodes.

We first individually benchmark the two operations of inserting (resp. removing) a segment to (resp. from) the dynamic segment tree. Our benchmark works by first creating a base dynamic segment tree of a certain size, then inserting new segments (resp. removing segments) into that tree. The number of new (resp. removed) segments is chosen to be the minimum of 10^5 and 5% of the base tree size. Segment borders are chosen by drawing twice from a uniform distribution. All segments are associated with a real-valued value. We conduct our experiments on a machine equipped with 128 GB of RAM and an Intel® Xeon® E5-1630 CPU, which has 10 MB of level 3 cache. We compile using GCC 8.1, at optimization level "-O3 -ffast-math". We do not run experiments concurrently. Each experiment is repeated for ten different seed values, and repeated five times for each seed value to account for measurement noise. All our code is published, see the link at the beginning of this paper.

Figure 3a displays the results for the insert operation. We see that the red-black tree performs best for this operation, about a 30% faster ($\approx 2.5\mu s$ per operation at $1.5 \cdot 10^7$ nodes) than the fastest zip tree variant, which is the variant using random rank selection ($\approx 3.5\mu s$ per operation). The two weight-balanced trees lie between the red-black tree and the randomness-based zip tree. Both hashing-based zip trees are considerably slower.

For the deletion operation, shown in Figure 3b, the randomness-based zip tree is significantly faster than the best competitor, the red-black tree. Again, the weight-balanced trees are slightly slower than the red-black tree, and the hashing-based zip trees fare the worst.



(a) Timings for inserting a segment.

(b) Timings for deleting a segment.

(c) Timings for moving a segment.

(d) Average node depths.

Figure 3 (a) – (c): Benchmark times for dynamic segment trees based on different balancing binary search trees. The y axis indicates the measured time per operation, while the x axis indicates the size of the tree that the operation is performed on. The lines indicate mean values. The standard deviation is all cases too small to be visible.
(d): Average depths of the nodes in DSTs based on red-black trees and zip trees. The x axis specifies the number of inserted segments. Shaded areas indicate the standard deviation.

Since zip trees are the fastest choice for deletion and red-black trees are the fastest for insertion, benchmarking the combination of both is obvious. Also, using an *dynamic* segment tree makes no sense in the absence of deletion. Thus, we next benchmark a *move* operation, which consists of first removing a segment from the tree, changing its borders, and re-inserting it. The results are shown in Figure 3c. We see that the randomness-based zipping segment tree is the best-performing dynamic segment tree for trees with at least $2.5 \cdot 10^6$ segments.

The obvious measurement to explore why different trees perform differently is the trees' balance, i.e., the average depth of a node in the respective trees. We conduct this experiment as follows: For each of the trees under study, we create trees of various sizes with randomly generated segments. In a tree generated in this way, we only see the effects of the *insert* operation, and not the *delete* operation. Thus, we continue by moving each segment once by removing it, changing its interval borders and re-inserting it. This way, the effect of the *delete* operation on the tree balance is also accounted for. Since the weight-balanced trees were not competitive previously, we perform this experiment only for the red-black and zip trees. We repeat the experiment with 30 different seeds to account for randomness. The results can be found in Figure 3d. We can see that zipping segment trees, whether based on randomness or hashing, are surprisingly considerably less balanced than red-black-based DSTs. Also, whether ranks are generated from hashing or randomness does not impact balance.

Concluding the evaluation, we gain several insights. First, deletions in zipping segment trees are so much faster than for red-black-based DSTs that they more than make up for the slower insertion, and the fastest choice for moving segments are zipping segment trees with ranks generated randomly. Second, we see that this speed does not come from a better balance, but in spite of a worse balance. The speedup must therefore come from more efficient rebalancing operations. Third, and most surprising, the question of how ranks are generated does not influence tree balance, but has a significant impact on the performance of deletion and insertion. However, the hash function we have chosen is very fast. Also, during deletion, no ranks should be (re-) generated for the variant that stores the ranks at the nodes. Thus, the performance difference can not be explained by the slowness of the hash function. Generating ranks with our chosen hash function must therefore introduce some disadvantageous structure into the tree that does not impact the average node depth.

5 Conclusion

We have presented zipping segment trees – a variation of dynamic segment trees, based on zip trees. The technique to maintain the necessary annotations for dynamic segment trees is comparatively simple, requiring only very little adaption of zip trees' routines. In our evaluation, we were able to show that zipping segment trees perform well in practice, and outperform red-black-tree or weight-balanced-tree based DSTs with regards to modifications.

However, we were not yet able to discover exactly why generating ranks from a (very simple) hash function does negatively impact performance. Exploring the adverse effects of this hash function and possibly finding a different hash function that avoids these effects remains future work. Another compelling future experiment would be to evaluate the performance when combined with the actual data structure by van Kreveld and Overmars.

All things considered, their relatively simple implementation and the superior performance when modifying segments makes zipping segment trees a good alternative to classical dynamic segment trees built upon rotation-based balancing binary trees.

References

1 Lukas Barth and Dorothea Wagner. Shaving peaks by augmenting the dependency graph. In *Proceedings of the Tenth ACM International Conference on Future Energy Systems*, pages 181–191. ACM, 2019. doi:10.1145/3307772.3328298.

2 Lukas Barth and Dorothea Wagner. Engineering top-down weight-balanced trees. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 161–174. SIAM, 2020. doi:10.1137/1.9781611976007.13.

3 Jon Louis Bentley. Algorithms for Klee's rectangle problems. Technical report, Department of Computer Science, Carnegie-Mellon University, 1977. Unpublished notes.

4 Yeim-Kuan Chang and Yung-Chieh Lin. Dynamic segment trees for ranges and prefixes. *IEEE Transactions on Computers*, 56(6):769–784, 2007. doi:10.1109/TC.2007.1037.

5 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry*. Springer, 2008. doi:10.1007/978-3-540-77974-2.

6 Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997. doi:10.1006/jagm.1997.0873.

7 Stefan Edelkamp, Shahid Jabbar, and Thomas Willhalm. Geometric travel planning. *IEEE Transactions on Intelligent Transportation Systems*, 6(1):5–16, 2005. doi:10.1109/TITS.2004.838182.

8 Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. Evaluation of labeling strategies for rotating maps. *Journal of Experimental Algorithmics (JEA)*, 21:1–21, 2016. doi:10.1145/2851493.

9 Johannes Antonius La Poutré. New techniques for the union-find problem. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 54–63, 1990.

10 Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16(4-5):464–497, 1996. doi:10.1007/BF01940876.

11 Raimund Seidel and Micha Sharir. Top-down analysis of path compression. *SIAM Journal on Computing*, 34(3):515–525, 2005. doi:10.1137/S0097539703439088.

12 Robert E Tarjan, Caleb C Levy, and Stephen Timmel. Zip trees. *CoRR*, abs/1806.06726, 2018. arXiv:1806.06726.

13 Robert E Tarjan, Caleb C Levy, and Stephen Timmel. Zip trees. In *Workshop on Algorithms and Data Structures (WADS)*, pages 566–577. Springer, 2019. doi:10.1007/978-3-030-24766-9_41.

14 Marc J van Kreveld and Mark H Overmars. Union-copy structures and dynamic segment trees. *Journal of the ACM (JACM)*, 40(3):635–652, 1993. doi:10.1145/174130.174140.

Fast and Stable Repartitioning of Road Networks

Valentin Buchhold
Karlsruhe Institute of Technology, Germany

Daniel Delling
Apple Inc, Cupertino, CA, USA

Dennis Schieferdecker
Apple Inc, Cupertino, CA, USA

Michael Wegner
Apple Inc, Cupertino, CA, USA

Abstract

We study the problem of graph partitioning for evolving road networks. While the road network of the world is mostly stable, small updates happen on a relatively frequent basis, as can been observed with the OpenStreetMap project (`http://www.openstreetmap.org`). For various reasons, professional applications demand the graph partition to stay roughly the same over time, and that changes are limited to areas where graph updates occur. In this work, we define the problem, present algorithms to satisfy the stability needs, and evaluate our techniques on continental-sized road networks. Besides the stability gains, we show that, when the changes are low and local, running our novel techniques is an order of magnitude faster than running graph partitioning from scratch.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Dynamic graph algorithms

Keywords and phrases Graph repartitioning, stable partitions, road networks, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.26

Funding This work was done while the first author was interning at Apple Inc.

1 Introduction

Graph partitioning is a core subroutine of many applications such as distributed computing, VLSI design, or shortest path computation. Although being NP-hard in general, many very efficient algorithms perform really well for realistic input problems (see [5] for an overview). In particular, the road network of the world can be partitioned with high quality in a multi-threaded fashion on a single machine in a few hours [8, 26]. However, one problem when applying these techniques to production systems is that the partition is not *stable*, i.e., small changes to the input may result in very different solutions (see Figure 1). This is a problem for various reasons. Firstly, road networks update surprisingly frequent, as can be seen by OpenStreetMap data, but these changes are often only minor, and restricted to local areas. Secondly, computing via paths [1] with customizable route planning [7] or customizable contraction hierarchies [10] exploit boundary nodes of partitions, which then may show very different results between two inputs in areas where no update has been applied. Finally, rerunning the full partitioning for every update seems like a waste of CPU hours, unnecessarily increasing data build times.

Our Contribution. We study the problem of making graph partitioning stable. The key idea is to obtain a partition for a slightly changed graph but analyzing both the previous graph and its partition. We identify regions that have changes and then try to repair the partition by rerunning graph partitioning subroutines on much smaller subgraphs. As a

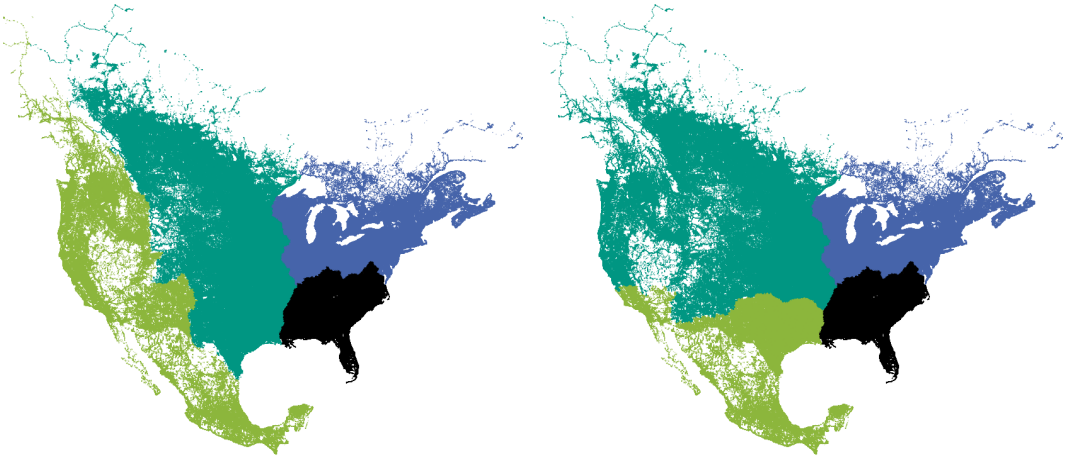


Figure 1 Two graph partitions with 4 cells each for the OpenStreetMap data of North America from August 2018 (left) and September 2018 (right). Both partitions have been obtained from the same algorithm with the same parameters and the same seeds. While the eastern part of the continent keeps roughly the same cut, the western part is cut very differently. However, within that month, only 0.5% of the vertices have churned (removed or added).

result, for an evolving road network like the OpenStreetMap data, we can keep the partition relatively stable, and are able to compute the updated partition an order of magnitude faster compared to running a graph partitioning algorithm from scratch.

Related Work. Motivated by several important applications, the graph partitioning problem has received considerable attention recently; see e.g. [5] for an overview. To partition road networks, early work [14, 2, 3] used general-purpose partitioners like Scotch [20], METIS [16], or Party [18]. However, one can compute partitions of significantly better quality when using special-purpose partitioners tailored to road networks.

The first such partitioning algorithm is PUNCH [8]. It introduces and exploits the concept of *natural cuts*, which are natural or man-made obstacles, such as rivers, mountains, and highways. At its heart is the *filtering phase*, which finds natural cuts by local maximum-flow computations and contracts all edges not contained in any natural cut. The *assembly phase* heuristically combines the resulting *fragments* to build a partition. Buffoon [24] incorporates the filtering phase of PUNCH into the general-purpose partitioning algorithm KaHIP [25].

An alternative approach to the filtering phase of PUNCH is Inertial Flow [26], which recursively bisects the network until the fragments are sufficiently small. To bisect the network, it exploits the geometric embedding of the network.

Like Inertial Flow, the FlowCutter algorithm [12] recursively bisects the network. For each bisection, it computes a Pareto set of nondominating cuts with respect to the cut size and balance, and picks a cut among those with a good tradeoff between cut size and balance. The recent InertialFlowCutter algorithm [11] is a variant of FlowCutter that uses geometric information, based on ideas from Inertial Flow. It is about 6 times faster than FlowCutter while preserving (or even slightly improving) the partition quality.

There has also been previous work on the graph *repartitioning* problem, although not in the context of road networks, but scientific computing applications. Various problems in solid and structural mechanics [37] and fluid dynamics [38] can be described by partial

differential equations (PDEs). Such PDEs can be solved by the finite-difference or finite-element method [36], which discretize the domain of the PDE by a *mesh*. The function value at each discretization point (i.e., vertex in the mesh) is approximately computed from the values at its neighboring vertices. Using an iterative scheme, new approximate values are determined by the values at the neighboring vertices from the previous iteration. Since finite-element meshes can become very large, they are partitioned into well-separated cells and distributed over multiple processing elements [5].

During the solution process, the mesh is refined in regions where large errors exist and coarsened in well-behaved regions. To maintain load balance, the mesh is periodically repartitioned. Besides the cut size (which correlates with the *communication volume*), the similarity between the old and new partition (which correlates with the *migration volume*) is an important optimization criterion for this application [5].

One approach are scratch-remap repartitioners [21, 32, 19, 29]. These first partition the new mesh using a state-of-the-art partitioner and then compute a migration-minimal mapping between the old and new partition. Since the new partition is produced from scratch, its cut size is small. However, the migration volume is often very high, since this criterion is considered only in the second phase.

Another approach are diffusion-based repartitioners [35, 27, 29]. These are inspired by the physical process of *diffusion*, i.e., vertices move from a cell of higher to one of lower vertex concentration. While this results in a good migration volume, the cut is often large.

The unified repartitioning algorithm [28] optimizes both criteria directly by combining the above-mentioned approaches. Following the multilevel graph partitioning approach, it iteratively contracts the new mesh using a variant of the heavy-edge matching algorithm [16], which matches two vertices only if they are in the same cell in the old partition. Next, the contracted mesh is partitioned twice using a scratch-remap and diffusion-based algorithm, respectively, and the partition with the better tradeoff between cut size and migration volume is picked. Finally, the mesh is iteratively uncontracted, using an improvement heuristic in each iteration to optimize the partition locally.

A rather simple approach [13, 33] is to introduce a zero-weight vertex for each cell in the old partition, which is not allowed to change its cell. This vertex is connected to each other vertex v in its cell by an edge whose weight represents the migration cost for v.

The problem we address is similar to the mesh repartitioning problem in that we optimize both the cut size and similarity. Note, however, that the old and new mesh are nested in the sense that new vertices result from splitting or merging vertices in the old mesh. Hence, there is a natural assignment of all vertices in the new mesh to cells in the old partition [34]. In fact, most work [21, 32, 27, 19, 29] considers a mesh with *fixed* topology, where adaptive mesh refinements are handled as vertex weight increases. In contrast, we allow vertices to be freely inserted into and removed from the network. For example, a newly constructed bridge that connects two previously disconnected cells has no natural assignment to any cell. Moreover, in the mesh repartitioning problem, the number of cells is fixed (since it is equal to the number of processing elements), while we allow the number of cells to change.

2 Preliminaries

We consider undirected graphs $G = (V, E)$ where each vertex $v \in V$ has a positive size $s(v)$ and each edge $\{u, v\} \in E$ has a positive weight $w(u, v)$. Our focus is on road networks, where vertices represent intersections and edges represent road segments. Partitioning algorithms often use *edge contractions*. To contract an edge $\{u, v\}$, we replace its endpoints by a single vertex w of size $s(w) = s(u) + s(v)$ and relink all edges incident on u or v to w. Multiple parallel edges are combined (adding up their weights) and self-loops are removed.

A *partition* of V is a set $P = \{C_1, \ldots, C_k\}$ of cells $C_i \subseteq V$ with the property that each vertex is contained in exactly one cell. A multilevel partition of V is a sequence $\mathcal{P} = \langle P^1, \ldots, P^L \rangle$ of partitions P^l, where l denotes the *level* of the partition. For ease of notation, we set $P^0 = \{\{v\} : v \in V\}$ and $P^{L+1} = \{V\}$. Since we use *nested* multilevel partitions, for each cell C_i^l, there is a cell $C_j^{l'}$ with $C_i^l \subseteq C_j^{l'}$ on all levels above; C_i^l is called a *subcell* of each $C_j^{l'}$. We denote by $c^l(v)$ the cell that contains v on level l. A *boundary* or *cut* edge on level l is an edge $\{u, v\}$ with $c^l(u) \neq c^l(v)$. Its endpoints are *boundary vertices* on level l. We denote by B^l the set of boundary vertices on level l.

2.1 Problem Statement

We are given two graphs $G = (V, E)$ and $\bar{G} = (\bar{V}, \bar{E})$, where $\bar{V}$ is obtained from V by inserting some vertices V^+ with $V^+ \cap V = \emptyset$ and removing some vertices $V^- \subseteq V$. Analogously, $\bar{E}$ is obtained from E by inserting some edges E^+ with $E^+ \cap E = \emptyset$ and removing some edges $E^- \subseteq E$. Hence, $\bar{V} = (V \setminus V^-) \cup V^+$ and $\bar{E} = (E \setminus E^-) \cup E^+$. We call G the *old graph* and $\bar{G}$ the *new graph*. In addition, we are given an L-level partition $\mathcal{P}$ of G with maximum cell sizes $U^1, \ldots, U^L$. The problem we consider is computing an L-level partition $\bar{\mathcal{P}}$ of $\bar{G}$ such that for each level l, $1 \leq l \leq L$, the size of each cell is bounded by U^l, the cut size is minimized, and the similarity between P^l and $\bar{P}^l$ is maximized.

It remains to formalize our notion of similarity. For real-world route planning systems following the partition-based overlay approach [30, 31, 15, 7], it is often desirable to keep the overlay topology fairly stable, as discussed in Section 1. Therefore, we define the similarity between two partitions P^l and $\bar{P}^l$ as the fraction of boundary vertices that are boundary vertices of both P^l and $\bar{P}^l$, i.e., $S^l = |B^l \cap \bar{B}^l| / |B^l \cup \bar{B}^l|$.

2.2 PUNCH

PUNCH [8] is a partitioning algorithm tailored to road networks. It has been applied [7, 6] to various partition-based shortest-path techniques, including CRP [7], Arc Flags [14, 17], and CHASE [3]. Given a graph G and a maximum cell size U, PUNCH splits the graph into cells of maximum size U while minimizing the cut size. It works in two phases. At its heart is the *filtering phase*, which finds *natural cuts* (natural or man-made obstacles, such as rivers, mountains, and railway tracks) and contracts all edges not contained in any natural cut. The *assembly phase* heuristically combines the resulting *fragments* to build a partition.

Filtering Phase. The *natural-cut heuristic* is executed in iterations. In each iteration, it picks a center c at random and builds a breath-first search (BFS) [23] tree T rooted at c until the total size of the vertices visited during the BFS reaches αU, where α is a parameter in $(0, 1]$. The neighbors $v \notin T$ of the vertices in T form the *ring* of c. Moreover, the vertices visited by the BFS before the total size reached $\alpha U / f$ form the *core* of c, where $f > 1$ is a second parameter. Then, the natural-cut heuristic temporarily contracts the core and the ring into a single source and sink vertex, respectively, determines a maximum flow between the source and the sink, finds a corresponding minimum cut, and marks all edges in this cut. The procedure stops when each vertex has been contained in at least one core.

To increase the number of marked edges, the iterative procedure is repeated $\mathcal{C}$ times. Afterwards, the natural-cut heuristic contracts all unmarked edges. The vertices in the resulting graph are called *fragments*. Note that each fragment represents a subgraph of G that was never cut and that each two adjacent fragments are separated by a natural cut. Typical parameter values for the filtering phase are $\alpha = 1$, $f = 10$, and $\mathcal{C} = 2$.

Assembly Phase. PUNCH runs a *greedy algorithm* to compute an initial partition of G from the fragment graph. It repeatedly contracts two adjacent vertices in the fragment graph until no contraction is possible without violating the upper bound U. The two vertices to be contracted next are picked based on a randomized *score function* [8]. Intuitively, the algorithm prefers small vertices that are tightly connected. The output of the greedy algorithm is a *contracted graph* H, where each vertex represents a cell in the partition of G.

The initial partition is then improved by an iterative *local search*. In each iteration, it picks two adjacent cells R, S at random from H. Let H_{RS} be the subgraph of H induced by R, S, and their neighbors in H and let G'_{RS} be obtained from H_{RS} by unpacking R and S into their constituent fragments (the neighbors remain contracted). Then, the greedy algorithm is run on G'_{RS}, outputting a contracted graph H'_{RS}. If H'_{RS} represents a better partition (i.e, one with a smaller cut size) than H_{RS}, we replace H_{RS} by H'_{RS} in the current solution H. The local search stops when each pair of adjacent cells in H has been considered φ times in succession without improving the current solution.

Since both the greedy algorithm and the local search are randomized, PUNCH uses a *multistart heuristic*, which runs the greedy algorithm followed by the local search multiple times on the fragment graph. After M candidate solutions have been generated, the best solution seen so far is returned. Alternatively, the candidate solutions generated by the multistart heuristic can be combined using an *evolutionary algorithm* [8]. Typical parameter values for the assembly phase are $\varphi = 16$ and $M = 9$.

2.3 Inertial Flow

Inertial Flow [26] is a partitioner tailored to road networks that exploits their geometric embedding. It has been applied [9, 4] to the partition-based shortest-path algorithms CRP [7] and CCH [10]. Its core algorithm bisects a graph $G = (V, E)$ with an embedding $\sigma : V \to \mathbb{R}^2$ into two balanced parts as follows:

(1) Pick a line ℓ with direction $d \in \mathbb{R}^2$.
(2) Project each point $\sigma(v)$, $v \in V$, orthogonally onto ℓ.
(3) Sort the vertices by their occurrence on ℓ.
(4) Determine a maximum flow between the first $\lfloor b|V| \rfloor$ vertices and the last $\lfloor b|V| \rfloor$ vertices.
(5) Find a corresponding minimum cut.

A typical parameter value is $b = 0.25$.

To find a partition of G with maximum cell size U, Inertial Flow recursively bisects G until the resulting parts have a size of at most U. For each bisection, Inertial Flow runs the core algorithm multiple times with parameter d set to $(0, 1)$, $(1, 0)$, $(1, 1)$, and $(-1, 1)$, respectively, and picks the smallest cut among those.

Inertial Flow computes partitions of reasonable quality. To improve the quality, Inertial Flow can be used to produce a partition with at most U/f vertices per cell, where $f > 1$ is a parameter. Contracting the edges within each cell yields a fragment graph. The fragments can then be combined as in the assembly phase of PUNCH. A typical parameter is $f = 32$.

3 Our Approach

This section discusses our approach to repartition road networks. Instead of partitioning the new graph from scratch, we start from the given partition, incorporate the vertices V^+, and repair and reoptimize the partition. We assume there are stable identifiers associated with the vertices in both graphs that allow us to map vertices in the old and new graph to

each other. Both OpenStreetMap and the proprietary data we are aware of provide such identifiers in the form of 64-bit integers. In case there are no stable identifiers available, we can heuristically map the vertices using, for example, their coordinates.

Our approach starts by mapping the partition $\mathcal{P}$ of the old graph G to the new graph $\bar{G}$. More precisely, each vertex $v \in V \cap \bar{V}$ inherits its cell identifiers from $\mathcal{P}$, i.e., we set $\bar{c}^l(v) = c^l(v)$ for all levels l. The vertices V^+ are not assigned to a cell on any level. After a quick preprocessing step (Section 3.1), we consider each cell $\bar{C}_i^l$ in the partition in descending level order, starting with the single cell on level $L+1$, which contains the entire new graph. Each cell $\bar{C}_i^l$, which induces the subgraph $\bar{G}\left[\bar{C}_i^l\right]$ of $\bar{G}$, is processed in two phases. The first phase assigns each vertex $v \in V^+$ to an existing cell on level $l-1$ (Section 3.2). The second phase repairs and reoptimizes the partition (Section 3.3).

We handle cells on the same level in parallel if multiple CPU cores are available. Moreover, if there is no change within a cell, we skip its subcells on all levels below.

3.1 Detecting Tiny Components

For several reasons (including data errors), there can be distinct components consisting of only a few vertices in the old graph that are connected to their neighborhood in the new graph. For example, consider a newly constructed road. In the old graph, it could still be under construction and not connected to the main network, and therefore a distinct component of its own. A partition could assign this component and its neighborhood to different cells, since there are no cut edges between them. However, connecting the new road to the main network leads to cut edges that are often unsuitably chosen. Hence, before the first cell assignment phase, we reset all cell identifiers for each vertex that belongs to a *tiny component* in the old but not in the new graph. We want such vertices to inherit the cell identifiers of their neighborhood rather than keeping their old identifiers. Formally, a tiny component is a connected component with a size that is below a given threshold.

3.2 Cell Assignment

Executing the first phase on cell $\bar{C}_i^l$ assigns each vertex $v \in V^+ \cap \bar{C}_i^l$ to a level-$(l-1)$ cell based on the cells of its neighbors. This phase resembles the label propagation algorithm [22] for clustering (evolving) networks. Each vertex is assigned to the cell to which the majority of its neighbors belong, with ties broken uniformly at random. We perform this process iteratively, where at every step, one vertex updates its cell. Note that the first assignment of a vertex is not necessarily final. Therefore, the process continues until no vertex $v \in V^+ \cap \bar{C}_i^l$ changes its cell anymore. Convergence is guaranteed since we move a vertex v from C_i to C_j only if $N_{C_j}(v)$ is *strictly* greater than $N_{C_i}(v)$, where $N_C(v)$ is the number of neighbors in cell C. Hence, with every such move, the sum $\sum_v N_{c(v)}(v)$ increases by $2(N_{C_j}(v) - N_{C_i}(v)) \geq 2$ (note that $N_{c(u)}(u)$ changes not only for $u = v$ but also for each neighbor u of v in $C_i \cup C_j$, causing the factor of 2). As the sum cannot exceed $\sum_v \deg(v)$, the process eventually stops.

To implement this approach efficiently, we keep track of the next vertex to assign with a min-heap, initialized with all new vertices adjacent to at least one old vertex. Every time we assign a vertex to a different cell, we insert its neighbors in V^+ into the min-heap. The priority of a vertex v is given by $key(v) = N_\perp(v) + N_2(v) - N_1(v)$, where $N_i(v)$ is the number of neighbors in the i-th most common neighboring cell (ties broken uniformly at random), and $N_\perp(v)$ is the number of as-yet-unassigned neighbors. The intuition here is that all assignments are both final and unambiguous as long as we only extract vertices v with $key(v) < 0$. This is easy to verify by induction on the number of delete-min operations.

When $key(v) = 0$, the assignment is not unambiguous but still final. Therefore, the choice of priorities ensures that we start with as many final assignments as possible, and thus reduces the number of cell corrections and the time to converge. Note that vertices unreachable from any vertex $v \in (V \setminus V^-) \cap \bar{C}_i^l$ remain unassigned after this phase. These vertices will be assigned to cells during the second phase that repairs and reoptimizes the partition.

To process the single cell on level $L + 1$, we must run cell assignment on the full input graph. For each cell C on all levels below, cell assignment must be run on $\bar{G}[C]$. For efficiency, we create a temporary copy of $\bar{G}[C]$ and run cell assignment on it. This simplifies cell assignment, allows us to use sequential local IDs, and improves locality.

3.3 Repair and Reoptimization

After the cell assignment phase, the partition of $\bar{G}\left[\bar{C}_i^l\right]$ is not necessarily feasible. First, there may be *oversized cells*, i.e., cells containing more than U^{l-1} vertices. Second, vertices unreachable from any vertex $v \in (V \setminus V^-) \cap \bar{C}_i^l$ have not been assigned to a cell yet. In the second phase, we repair both issues and locally reoptimize the partition.

Let K be a graph whose vertices are the cells in the partition. Each as-yet-unassigned vertex in $\bar{G}\left[\bar{C}_i^l\right]$ forms a cell of its own. The size of each vertex in K is the number of vertices in the corresponding cell. There is an edge $\{R, S\}$ in K if there is an edge $\{u, v\}$ in $\bar{G}\left[\bar{C}_i^l\right]$ with $u \in R$ and $v \in S$. Its weight is the total weight of the corresponding edges in $\bar{G}\left[\bar{C}_i^l\right]$.

Let K' be a graph obtained from K by *unpacking* some of the cells (we will discuss cell unpacking in detail in Section 3.4). In the following, we compute a partition of K', which can easily be transformed into a partition of $\bar{G}\left[\bar{C}_i^l\right]$. Note that to obtain a feasible partition, we must unpack at least each oversized cell. To increase the similarity between $\mathcal{P}$ and $\bar{\mathcal{P}}$, we can relax our definition of oversized cells, allowing cells to contain at most $g^{l-1}U^{l-1}$ vertices, where $g^{l-1} \geq 1$ is the *growth factor* on level $l - 1$.

To find an initial feasible partition, we run the greedy algorithm from PUNCH on K', yielding a contracted graph H. This partition is then reoptimized by running a variant of the local search from PUNCH on H. That is, we repeatedly pick two adjacent cells R, S at random from H, run the greedy algorithm on the subgraph of H induced by R, S, and their neighbors in H, and update the current solution H accordingly. However, while PUNCH unpacks R and S into their constituent fragments, we unpack them into the corresponding vertices in K'. Since both the greedy algorithm and the local search are randomized, we run them M times on K' and return the partition with the smallest cut size.

As already mentioned, we handle cells on a level $l \leq L$ in parallel. On such levels, we run a sequential version of the local search. On level $L + 1$ (where there is only a single cell), we parallelize the local search by trying multiple pairs of adjacent vertices concurrently.

3.4 Cell Unpacking

We considered three variants of cell unpacking, inspired by PUNCH, which differ in how much they unpack. See Figure 2 for an illustration. The simplest variant replaces the single vertex in K representing an oversized level-l cell $\bar{C}_i^l$ with one vertex for each level-$(l - d)$ subcell C_j^{l-d} with $C_j^{l-d} \cap \bar{C}_i^l \neq \emptyset$ (where $d \geq 1$ denotes the *descent step*) and one vertex for each vertex $v \in V^+$ that has been assigned to $\bar{C}_i^l$ during the first phase (cell assignment). The size of these vertices is the number of vertices in the new graph that they represent (possibly one). We call this variant *simple unpacking*.

The second variant, *neighbor unpacking*, gives the second phase more degrees of freedom to reoptimize the partition. Besides unpacking oversized cells, this variant also unpacks all cells that have a common boundary with an oversized cell.

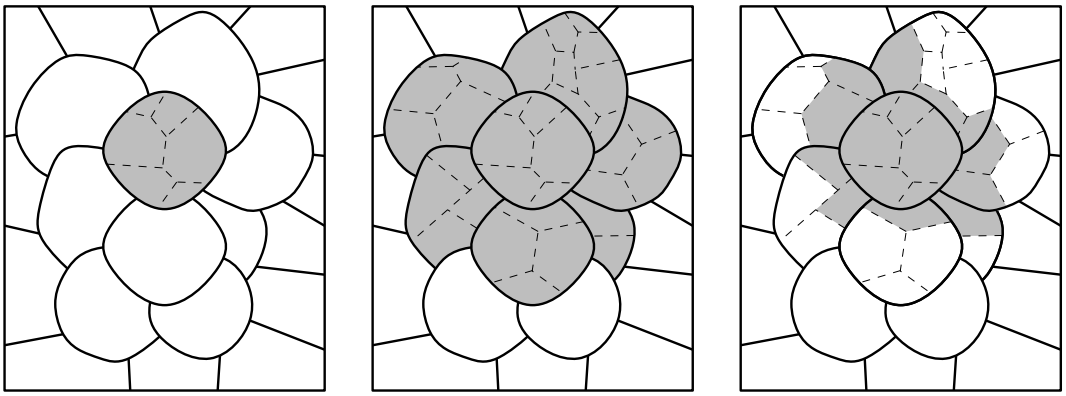


Figure 2 Unpacking an oversized cell during the reoptimization phase. Left: The simple variant unpacks only the oversized cell. Middle: Neighbor unpacking also unpacks all neighboring cells. Right: Partial unpacking unpacks the neighboring cells partially.

The third variant, which we call *partial unpacking*, unpacks each oversized level-l cell $\bar{C}_i^l$ fully and each level-l cell $\bar{C}_j^l$ that has a common boundary with $\bar{C}_i^l$ partially. More precisely, we replace the single vertex in K representing $\bar{C}_j^l$ with one vertex for each level-$(l-d)$ subcell C_k^{l-d} with $C_k^{l-d} \cap \bar{C}_j^l \neq \emptyset$ that directly borders on $\bar{C}_i^l$, one vertex for each vertex $v \in V^+$ that has been assigned to $\bar{C}_j^l$ during the first phase, and one vertex representing the remaining level-$(l-d)$ subcells of $\bar{C}_j^l$. Again, the size of each vertex is the number of vertices in the new graph that are represented by the vertex.

4 Experiments

4.1 Implementation and System

Both the partitioning and repartitioning algorithms are implemented in C++11 and were compiled with GCC 4.8.5 on a system running CentOS 7.7. For parsing the instances we use the RoutingKIT library [10]. The machine has 2 NUMA nodes, each equipped with a 10 core/20 threads Intel®Xeon®CPU E5-2640 v4 clocked at 2.40GHz with 2.5 MiB L2 and 25 MiB L3 cache. It has 192 GiB of DDR4-2400 RAM.

4.2 Instances

We evaluate and compare our algorithm with the full partitioning algorithm (FP) on road networks extracted from OpenStreetMap (`https://www.openstreetmap.org`). Our FP algorithm uses Inertial Flow to find a starting solution and an assembly phase similar to the one from PUNCH to optimize it. OpenStreetMap's contributors edit the map regularly which enables us to test our repartitioning algorithm on snapshots taken at different dates of the same cutout. We test our algorithms on the Australia, South America, North America and Europe instances available from GeoFabrik (`https://download.geofabrik.de/index.html`). Evaluation is performed between snapshots that are one year (1/1/2018 - 1/1/2019) and one month apart (10/1/2019 - 11/1/2019). Shorter time periods (e.g. a week) differ less and thus repartitioning performs at least as good as it does on monthly and yearly instances. More detailed information on the instances can be found in Table 1. For the remainder of this section, we reference the graph pairs of the time frames by an M and a Y suffix for the monthly and yearly instances respectively (e.g. $NorthAmerica_Y$ for the North America graph pair on 1/1/2018 and 1/1/2019).

Table 1 Instances and their properties. Absolute numbers in millions, relative numbers and vertex churn (VC) in percent. VC is defined as the ratio of $\frac{|V^+ \cup V^-|}{|V|}$.

Instance [1]	1/1/2018		1/1/2019			10/1/2019		11/1/2019		
	$\lvert V_{18}\rvert$	$\lvert E_{18}\rvert$	$\frac{\lvert V_{19}\rvert}{\lvert V_{18}\rvert}$	$\frac{\lvert E_{19}\rvert}{\lvert E_{18}\rvert}$	VC_Y	$\lvert V_{10}\rvert$	$\lvert E_{10}\rvert$	$\frac{\lvert V_{11}\rvert}{\lvert V_{10}\rvert}$	$\frac{\lvert E_{11}\rvert}{\lvert E_{10}\rvert}$	VC_M
Australia	1.23	2.85	7.83	6.63	11.33	1.41	3.18	0.45	0.41	0.62
N. America	24.90	61.38	1.27	0.98	6.84	26.05	63.94	0.28	0.26	0.99
Europe	30.55	71.46	3.44	3.05	7.12	32.69	76.00	0.54	0.52	1.00

1) Downloaded from `https://download.geofabrik.de` on 12/16/2019.

Table 2 Algorithm quality and performance on $Australia_Y$. Best values in bold.

Algorithm	CutSize [%]	S^1 [%]	S^2 [%]	S^3 [%]	S^4 [%]	S^5 [%]	Runtime [s]
SU	2.00	**79.53**	**71.18**	**65.15**	**57.19**	**49.78**	58.57
NU	**0.40**	68.05	63.57	58.25	50.64	22.11	98.06
PU	2.00	**79.53**	**71.18**	**65.15**	**57.19**	**49.78**	**58.35**

4.3 Parameters

Algorithms. We start with a comparison of our repartitioning algorithm variants SU (simple unpacking), NU (neighbor unpacking) and PU (partial unpacking) on the $Australia_Y$ instance with a cell growth of 0% and a descent step $d = 1$ (cf. Section 3.3) in Table 2. Using a larger d does not result in better quality or similarity based on our experiments. Simple unpacking and partial unpacking perform almost identically, both increasing the overall cut size by 2% compared to the result of the full partitioning algorithm. In our experiments we found that PU often has slightly worse similarity on the two lowest levels and identical similarity on the higher levels compared to NU while runtimes are comparable. The neighbor unpacking approach considers even more cells than PU for distributing unassigned vertices which results in smaller cuts at the cost of a reduced similarity and higher runtimes. Other instances produce similar results. Based on this evaluation, we focus on the simple unpacking approach a descent step $d = 1$ as it produces partitions of good quality and similarity with reasonable runtimes. In the remainder of this section, we use RP to denote our repartitioning algorithm using simple unpacking and FP to denote the full partitioning (from scratch) algorithm. The level-dependent parameters for both algorithms can be found in Table 3. We use the default parameters for our FP algorithm, whereas we reduce ϕ_{RP} and M_{RP} on the higher levels. Running more local searches and producing more candidates on these levels reduces similarity since the searches optimize cut size, not similarity and it increases the runtime.

Cell Growth. When new vertices are added to the graph, some cells have to be split in order to hold the size constraint on the cell size. However, most often these additional vertices do not affect the boundary of a cell but are contained in it. So instead of splitting cells, increasing the cut size and decreasing similarity, it is better to allow some cell growth in order to improve similarity. We evaluate the effect of cell growth on $Australia_Y$ and $Australia_M$ in Table 4. As expected, the similarity increases on all levels with higher cell growth at the cost of a more imbalanced partition - some cells utilizing all the allowed growth. The similarity change is most pronounced on the highest level, on the lowest level the change

Table 3 Common partitioning parameters per level of FP and RP.

Level	U	f	φ_{FP}	φ_{RP}	M_{FP}	M_{RP}
1	25	16	9	9	3	3
2	200	16	9	9	3	3
3	1600	32	16	9	4	3
4	12800	32	16	9	4	3
5	102400	32	32	9	6	3
6	819200	32	32	9	6	3
7	6553600	32	32	9	16	3

Table 4 Impact of allowing cell growth when running RP on $Australia_Y$ and $Australia_M$.

	Oversized Cells [%]		S^L [%]		Runtime [s]	
Cell Growth [%]	Year	Month	Year	Month	Year	Month
0	0.00	0.00	49.78	77.00	58.57	2.26
1	0.86	0.66	37.24	97.60	63.17	2.05
5	13.25	1.24	61.62	97.60	39.16	1.89
10	19.72	1.70	75.86	94.14	20.39	1.82

is less than 2%. There are now oversized cells that exceed the maximum cell size on each level. In the table we report the ratio of the total number of oversized cells over the total number of cells over all levels. The runtime of RP decreases with higher cell growth because the local optimizer has less work to do. While the imbalance introduced for the $Australia_M$ instance is always below 2%, it is significantly higher for $Australia_Y$ which is due to the fact that the latter instance has much more churn in both vertices and edges which makes it harder to repair the partition.

4.4 Comparison with Full Partitioning

Quality and Performance over Time. The more a graph churns over time, the harder it gets to keep the partition stable which is reflected in increased runtimes of our algorithm. Figures 3 and 4 show the effect of increased churn on the quality and runtime on monthly North America snapshots from 02/01/2018 to 12/01/2018 with 01/01/2018 used as the baseline partition that we want to keep stable. Vertex churn starts at 0.7% for 02/01/2018 and increases strictly monotonously to 6.1% for 12/01/2018. We report the similarity S^1, the total relative amount of oversized cells (over all levels) and the total runtime of RP with cell growth parameters 0%, 5% and 10%. For comparison, we include the FP algorithm with a cell growth of 0%. For FP, cell growth means increasing the maximum cell size U on each level. Higher cell growths for FP do not change the runtime significantly and lead to worse similarity as FP optimizes cut size and not similarity, so we exclude them in the figure. Similarity on higher levels follows the same trend as the similarity on level one, just slightly lower as is the case in our other experiments. A cell growth of 10% results in the best similarity values but there is never more than 1% difference between the different cell growth configurations. In contrast, the similarity of FP is much worse as FP does not optimize this measure. In terms of the partition quality, we compare cut size increases over the partition obtained by FP with the same amount of cell growth and notice that the

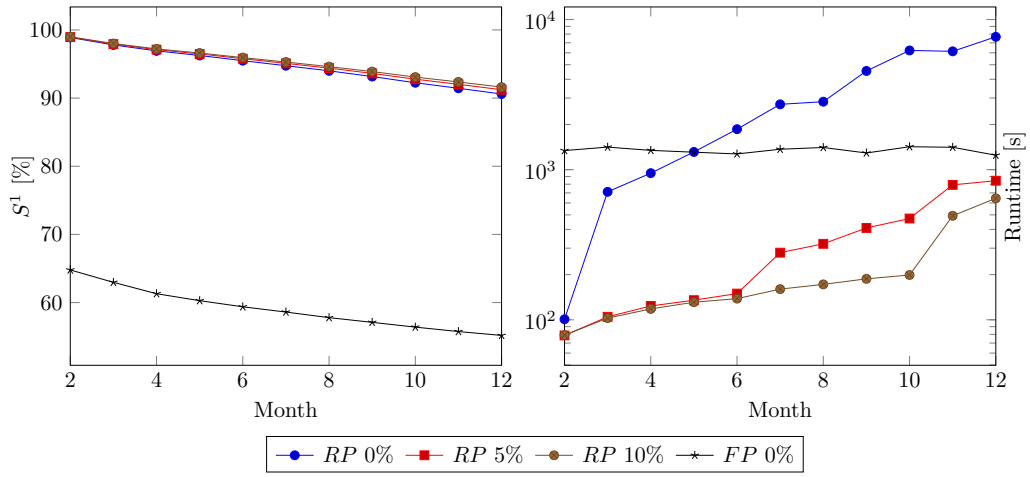


Figure 3 Similarity and runtime comparison of RP and FP for different cell growths on the monthly North America graphs between 02/01/2018 and 12/01/2018 with 01/01/2018 as the baseline partition to be kept stable.

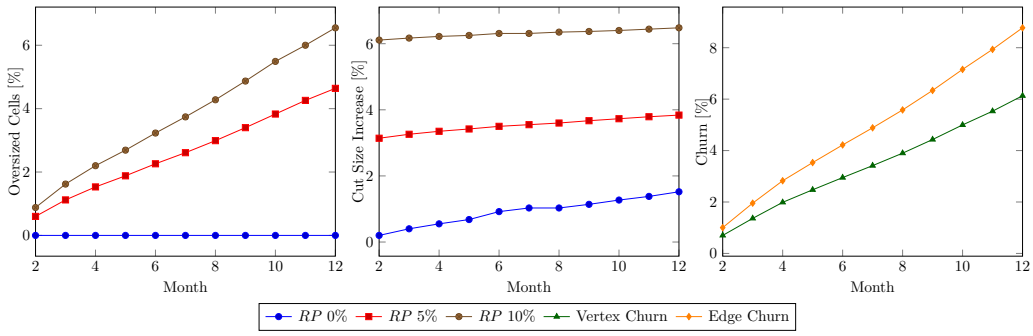


Figure 4 Oversized cells, cut size increase of RP for difference cell growths and graph churn on the monthly North America graphs between 02/01/2018 and 12/01/2018 with 01/01/2018 as the baseline partition to be kept stable. Cut size is compared to FP with the same cell growth.

cut size increases by roughly 1%, 3% and 6% for the respective cell growths and does not significantly increase with more churn. This can be explained by the fact that our algorithm mainly optimizes similarity to the input partition whereas allowing FP a higher imbalance can lead to different (smaller) cuts. As expected, the ratio of all oversized cells compared to the number of cells increases for a cell growth greater than 0% but stays within reasonable limits for the purpose of road network partitioning. The runtime of RP with 0% cell growth is comparable to the higher cell growths for the first month but increases sharply starting with the third month. Allowing cell growths of 5% and 10% yield similar running times up to month 6. Starting with month 7, however, RP with cell growth 5% has a significantly higher runtime, while the vertex/edge churn does not have any significant increase during these months. A possible explanation might be the merging of small connected components with bigger ones. In this case, a larger cell growth often allows to assign all vertices of the previously connected component to the now neighboring cell, improving the runtime and retaining the similarity.

Table 5 Comparison of RP and FP, both with 5% cell growth, on the monthly instances.

Instance	CutSize [%]	S^1 [%]		S^L [%]		Runtime [s]	
	RP	RP	FP	RP	FP	RP	FP
$Australia_M$	3.09	98.87	64.57	97.60	39.27	2.01	24.27
$NAmerica_M$	3.11	98.64	64.01	87.16	31.07	144.26	1390.57
$Europe_M$	3.24	98.35	62.70	94.70	50.73	197.90	1851.13

Table 6 Comparison of RP and FP, both with 20% cell growth, on the yearly instances.

Instance	CutSize [%]	S^1 [%]		S^L [%]		Runtime [s]	
	RP	RP	FP	RP	FP	RP	FP
$Australia_Y$	10.91	85.16	49.60	75.49	23.33	3.50	22.08
$NAmerica_Y$	13.14	91.04	54.55	62.46	32.53	356.81	1459.72
$Europe_Y$	12.83	90.41	52.98	73.64	46.03	251.47	1880.93

Quality and Performance on Monthly Instances. Based on the results of our monthly RP evaluation over the course of a year in the previous paragraph, we select a cell growth of 5% for the comparison with FP on the monthly instances. For FP, we use a cell growth of 0% for comparing similarity for best FP results and a cell growth of 5% for a fair comparison of cut sizes. The result of that evaluation can be found in Table 5. The similarity to the previous partition is always higher than 98% on the lowest level compared to about 67% for FP. RP is able to maintain high similarity values on higher levels as well. The cut size is no more than about 3% higher compared to FP and the runtime is up to a factor of 12 lower.

Quality and Performance on Yearly Instances. We also include quality and performance figures for the yearly instances where we select a cell growth of 20% to maximize similarity. For FP, we chose the same testing methodology as in the monthly comparison in terms of cell growth. While our algorithm is able to maintain good similarity values of 85% or higher on the lowest level, similarity decreases more drastically compared to the monthly instances. The FP algorithm only achieves up to 54% similarity on the lowest level and higher levels even worse. In terms of cut size, it shows the limitations of trying to keep a partition stable for a full year with good similarity as our algorithm produces cuts that are overall up to 14% higher compared to FP.

5 Conclusion

We studied the stable graph partitioning problem in road networks. We showed how to keep a graph partition stable on OpenStreetMap road networks over time. A nice benefit of our approach is a reduction of an order of magnitude in runtime compared to partitioning the networks from scratch. Regarding future work, we are interested in dealing with larger churn in the graph, like adding large new regions to the input. One possible approach here might be to find natural cuts in these new regions and then add the fragments to our scheme. Finally, we are interested in studying other types of evolving networks.

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):1.3:1–1.3:17, 2013. doi:10.1145/2444016.2444019.

2 Reinhard Bauer and Daniel Delling. SHARC: Fast and robust unidirectional routing. *ACM Journal of Experimental Algorithmics*, 14:2.4:1–2.4:29, 2009. doi:10.1145/1498698.1537599.

3 Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. *ACM Journal of Experimental Algorithmics*, 15:2.3:1–2.3:31, 2010. doi:10.1145/1671970.1671976.

4 Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Real-time traffic assignment using engineered customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 24(2):2.4:1–2.4:28, 2019. doi:10.1145/3362693.

5 Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering: Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 117–158. Springer, 2016. doi:10.1007/978-3-319-49487-6_4.

6 Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013. doi:10.1016/j.jpdc.2012.02.007.

7 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.

8 Daniel Delling, Andrew V. Goldberg, Ilya P. Razenshteyn, and Renato F. Werneck. Graph partitioning with natural cuts. In *25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'11)*, pages 1135–1146. IEEE Computer Society, 2011. doi:10.1109/IPDPS.2011.108.

9 Daniel Delling, Dennis Schieferdecker, and Christian Sommer. Traffic-aware routing in road networks. In *34th IEEE International Conference on Data Engineering (ICDE'18)*, pages 1543–1548. IEEE Computer Society, 2018. doi:10.1109/ICDE.2018.00172.

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

11 Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. Faster and better nested dissection orders for customizable contraction hierarchies. *Algorithms*, 12(9):1–20, 2019. doi:10.3390/a12090196.

12 Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *ACM Journal of Experimental Algorithmics*, 23(1):1.2:1–1.2:34, 2018. doi:10.1145/3173045.

13 Bruce Hendrickson, Robert W. Leland, and Rafael Van Driessche. Enhancing data locality by using terminal propagation. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences (HICSS'96)*, pages 565–574. IEEE Computer Society, 1996. doi:10.1109/HICSS.1996.495507.

14 Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 41–72. American Mathematical Society, 2009.

15 Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics*, 13:2.5:1–2.5:26, 2008. doi:10.1145/1412228.1412239.

16 George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998. doi:10.1137/S1064827595287997.

17 Ulrich Lauther. An experimental evaluation of point-to-point shortest path calculation on road networks with precalculated edge-flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 19–39. American Mathematical Society, 2009.

18 Burkhard Monien and Stefan Schamberger. Graph partitioning with the party library: Helpful-sets in practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 198–205. IEEE Computer Society, 2004. doi:10.1109/SBAC-PAD.2004.18.

19 Leonid Oliker and Rupak Biswas. PLUM: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998. doi:10.1006/jpdc.1998.1469.

20 François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In Heather M. Liddell, Adrian Colbrook, Louis O. Hertzberger, and Peter M. A. Sloot, editors, *Proceedings of the 4th International Conference and Exhibition on High-Performance Computing and Networking (HPCN'96)*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer, 1996. doi:10.1007/3-540-61142-8_588.

21 Eddy Pramono, Horst D. Simon, and Andrew Sohn. Dynamic load balancing for finite element calculations on parallel computers. In David H. Bailey, Petter E. Bjørstad, John R. Gilbert, Michael Mascagni, Robert S. Schreiber, Horst D. Simon, Virginia Torczon, and Layne T. Watson, editors, *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, pages 599–604. SIAM, 1995.

22 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), 2007. doi:10.1103/PhysRevE.76.036106.

23 Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures – The Basic Toolbox*. Springer, 2019. doi:10.1007/978-3-030-25209-0.

24 Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. In David A. Bader and Petra Mutzel, editors, *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 16–29. SIAM, 2012. doi:10.1137/1.9781611972924.2.

25 Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013. doi:10.1007/978-3-642-38527-8_16.

26 Aaron Schild and Christian Sommer. On balanced separators in road networks. In Evripidis Bampis, editor, *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, volume 9125 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 2015. doi:10.1007/978-3-319-20086-6_22.

27 Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997. doi:10.1006/jpdc.1997.1410.

28 Kirk Schloegel, George Karypis, and Vipin Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In Louis H. Turcotte, editor, *Proceedings of the 13th ACM/IEEE Supercomputing Conference (SC'00)*, pages 1–11. IEEE Computer Society, 2000. doi:10.1109/SC.2000.10035.

29 Kirk Schloegel, George Karypis, and Vipin Kumar. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):451–466, 2001. doi:10.1109/71.926167.

30 Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics*, 5:1–23, 2000. doi:10.1145/351827.384254.

31 Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Using multi-level graphs for timetable information in railway systems. In David M. Mount and Clifford Stein, editors, *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002. doi:10.1007/3-540-45643-0_4.

32 Andrew Sohn and Horst D. Simon. JOVE: A dynamic load balancing framework for adaptive computations on an SP-2 distributed-memory multiprocessor. Technical Report 94-60, New Jersey Institute of Technology, Department of Computer and Information Science, 1994.

33 Chris Walshaw. Variable partition inertia: Graph repartitioning and load balancing for adaptive meshes. In Manish Parashar and Xiaolin Li, editors, *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*, pages 357–380. John Wiley & Sons, 2009. doi:10.1002/9780470558027.ch17.

34 Chris Walshaw and Martin Berzins. Dynamic load-balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice and Experience*, 7(1):17–28, 1995. doi:10.1002/cpe.4330070103.

35 Chris Walshaw, Mark Cross, and Martin G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, 1997. doi:10.1006/jpdc.1997.1407.

36 Pei-bai Zhou. *Numerical Analysis of Electromagnetic Fields*. Electric Energy Systems and Engineering Series. Springer, 1993. doi:10.1007/978-3-642-50319-1.

37 Olgierd C. Zienkiewicz, Robert L. Taylor, and David D. Fox. *The Finite Element Method for Solid and Structural Mechanics*. Butterworth-Heinemann, 2014. doi:10.1016/C2009-0-26332-X.

38 Olgierd C. Zienkiewicz, Robert L. Taylor, and Perumal Nithiarasu. *The Finite Element Method for Fluid Dynamics*. Butterworth-Heinemann, 2014. doi:10.1016/C2009-0-26328-8.

Path Query Data Structures in Practice

Meng He
Faculty of Computer Science, Dalhousie University, Halifax, Canada
mhe@cs.dal.ca

Serikzhan Kazi
Faculty of Computer Science, Dalhousie University, Halifax, Canada
skazi@dal.ca

Abstract

We perform experimental studies on data structures that answer path median, path counting, and path reporting queries in weighted trees. These query problems generalize the well-known range median query problem in arrays, as well as the $2d$ orthogonal range counting and reporting problems in planar point sets, to tree structured data. We propose practical realizations of the latest theoretical results on path queries. Our data structures, which use tree extraction, heavy-path decomposition and wavelet trees, are implemented in both succinct and pointer-based form. Our succinct data structures are further specialized to be plain or entropy-compressed. Through experiments on large sets, we show that succinct data structures for path queries may present a viable alternative to standard pointer-based realizations, in practical scenarios. Compared to naïve approaches that compute the answer by explicit traversal of the query path, our succinct data structures are several times faster in path median queries and perform comparably in path counting and path reporting queries, while being several times more space-efficient. Plain pointer-based realizations of our data structures, requiring a few times more space than the naïve ones, yield up to 100-times speed-up over them.

2012 ACM Subject Classification Information systems → Data structures

Keywords and phrases path query, path median, path counting, path reporting, weighted tree

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.27

Related Version `https://arxiv.org/abs/2001.10567v3`

Supplementary Material The source code is accessible at `https://github.com/serkazi/tree_path_queries`.

Funding This work was supported by NSERC of Canada.

1 Introduction

Let T be an ordinal tree on n nodes, with each node x associated with a *weight* $\mathbf{w}(x)$ over an alphabet $[\sigma]$.[1] A *path query* in such a tree asks to evaluate a certain given function on the path $P_{x,y}$, which is the path between two given query nodes, x and y. A *path median* query asks for the median weight on $P_{x,y}$. A *path counting* (*path reporting*) query counts (reports) the nodes on $P_{x,y}$ with weights falling inside the given query weight range. These queries generalize the range median problem on arrays, as well as the $2d$ orthogonal counting and reporting queries in point sets, by replacing one of the dimensions with tree topology. Formally, query arguments consist of a pair of vertices $x, y \in T$ along with an interval Q. The goal is to preprocess the tree T for the following types of queries:

- *Path Counting*: return $|\{z \in P_{x,y} \mid \mathbf{w}(z) \in Q\}|$.
- *Path Reporting*: enumerate $\{z \in P_{x,y} \mid \mathbf{w}(z) \in Q\}$.
- *Path Selection*: return the k^{th} $(0 \leq k < |P_{x,y}|)$ weight in the sorted list of weights on $P_{x,y}$; k is given at query time. In the special case of $k = \lfloor |P_{x,y}|/2 \rfloor$, a path selection is a *path median query*.

[1] we set $[n] \triangleq \{1, 2, \ldots, n\}$.

Path queries is a widely-researched topic in computer science community [7, 15, 23, 32, 28, 13, 24]. Apart from theoretical appeal, queries on tree topologies reflect the needs of efficient information retrieval from hierarchical data, and are gaining ground in established domains such as RDBMS [2]. The expected height of T being $\sqrt{2\pi n}$ [42], this calls for the development of methods beyond naïve.

Previous work includes that of Krizanc et al. [32], who were the first to introduce path median query problem (henceforth `PM`) in trees, and gave an $\mathcal{O}(\lg n)$ query-time data structure with the space cost of $\mathcal{O}(n \lg^2 n)$ words. They also gave an $\mathcal{O}(n \log_b n)$ words data structure to answer `PM` queries in time $\mathcal{O}(b \lg^3 n / \lg b)$, for any fixed $1 \leq b \leq n$. Chazelle [15] gave an emulation dag-based linear-space data structure for solving path counting (henceforth `PC`) queries in trees in time $\mathcal{O}(\lg n)$.

While [32, 15] design different data structures for `PM` and `PC`, He et al. [26, 28] use *tree extraction* to solve both `PC` and the path selection problem (henceforth `PS`), as well as the path reporting problem (henceforth `PR`), which they were the first to introduce. The running times for `PS`/`PC` were $\mathcal{O}(\lg \sigma)$, while a `PR` query is answered in $\mathcal{O}((1+\kappa)\lg \sigma)$ time, with κ henceforth denoting output size. Also given is an $\mathcal{O}(n \lg\lg \sigma)$-words and $\mathcal{O}(\lg \sigma + \kappa \lg\lg \sigma)$ query time solution, for `PR`, in the RAM model.

Further, solutions based on *succinct* data structures started to appear. (In the interests of brevity, the convention throughout this paper is that a data structure is *succinct* if its size in bits is close to the information-theoretic lower bound.) Patil et al. [40] presented an $\mathcal{O}(\lg n \cdot \lg \sigma)$ query time data structure for `PS`/`PC`, occupying $6n + n \lg \sigma + o(n \lg \sigma)$ bits of space. Therein, the tree structure and the weights distribution are decoupled and delegated to respectively heavy-path decomposition [43] and wavelet trees [37]. Their data structure also solves `PR` in $\mathcal{O}(\lg n \lg \sigma + (1+\kappa)\lg \sigma)$ query time.

Parallel to [40], He et al. [26, 28] devised a succinct data structure occupying $nH(W_T) + o(n \lg \sigma)$ bits of space to answer `PS`/`PC` in $\mathcal{O}(\frac{\lg \sigma}{\lg\lg n} + 1)$, and `PR` in $\mathcal{O}((1+\kappa)(\frac{\lg \sigma}{\lg\log n} + 1))$ time. Here, W_T is the multiset of weights of the tree T, and $H(W_T)$ is there entropy thereof. Combining tree extraction and the ball-inheritance problem [14], Chan et al. [13] proposed further trade-offs, one of them being an $\mathcal{O}(n \lg^{\epsilon} n)$-word structure with $\mathcal{O}(\lg\lg n + \kappa)$ query time, for `PR`.

Despite the vast body of work, little is known on the practical performance of the data structures for path queries, with empirical studies on weighted trees definitely lacking, and existing related experiments being limited to navigation in unlabeled trees only [8], or to very specific domains [5, 38]. By contrast, the empirical study of traditional orthogonal range queries have attracted much attention [9, 12, 29]. We therefore contribute to remedying this imbalance.

1.1 Our work

In this article, we provide an experimental study of data structures for path queries. The types of queries we consider are `PM`, `PC`, and `PR`. The theoretical foundation of our work are the data structures and algorithms developed in [26, 40, 27, 28]. The succinct data structure by He et al. [28] is optimal both in space and time in the RAM model. However, it builds on components that are likely to be cumbersome in practice. We therefore present a practical compact implementation of this data structure that uses $3n \lg \sigma + o(n \lg \sigma)$ bits of space as opposed to the original $nH(W_T) + o(n \lg \sigma)$ bits of space in [28]. For brevity, we henceforth refer to the data structures based on tree extraction as `ext`. Our implementation of `ext` achieves the query time of $\mathcal{O}(\lg \sigma)$ for `PM` and `PC` queries, and $\mathcal{O}((1+\kappa)\lg \sigma)$ time for `PR`. Further, we present an exact implementation of the data structure (henceforth `whp`) by

Patil et al. [40]. The theoretical guarantees of `whp` are $6n + n\lg\sigma + o(n\lg\sigma)$ bits of space, with $\mathcal{O}(\lg n \lg\sigma)$ and $\mathcal{O}(\lg n\lg\sigma + (1+\kappa)\lg\sigma)$ query times for respectively `PM/PC` and `PR`. Although `whp` is optimal neither in space nor in time, it proves competitive with `ext` on the practical datasets we use. Further, we evaluate time- and space-impact of succinctness by realizing plain pointer-based versions of both `ext` and `whp`. We show that succinct data structures based on `ext` and `whp` offer an attractive alternative for their fast but space-consuming counterparts, with query-time slow-down of 30-40 times yet commensurate savings in space. We also implement, in pointer-based and succinct variations, a naïve approach of not preprocessing the tree at all but rather answering the query by explicit scanning. The succinct solutions compare favourably to the naïve ones, the slowest former being 7-8 times faster than naïve `PM`, while occupying up to 20 times less space. We also compare the performance of different succinct solutions relative to each other.

2 Preliminaries

This section introduces notation and main algorithmic techniques at the core of our data structures.

Notation. The i^{th} node visited during a preorder traversal of the given tree T is said to have *preorder rank* i. We identify a node by its preorder rank. For a node $x \in T$, its set of ancestors $\mathcal{A}(x)$ includes x itself. Given nodes $x, y \in T$, where $y \in \mathcal{A}(x)$, we set $A_{x,y} \triangleq P_{x,y} \setminus \{y\}$; one then has $P_{x,y} = A_{x,z} \sqcup \{z\} \sqcup A_{y,z}$, where $z = LCA(x,y)$. The primitives `rank/select/access` are defined in a standard way, i.e. $\texttt{rank}_1(B,i)$ is the number of 1-bits in positions less than i, $\texttt{select}_1(B,j)$ returns the position of the j^{th} 1-bit, and $\texttt{access}(\texttt{B},\texttt{i})$ returns the bit at the i^{th} position, all with respect to a given bitmap B, which is omitted when the context is clear.

Compact representations of ordinal trees. Compact representations of ordinal trees is a well-researched area, mainstream methodologies including *balanced parentheses* (BP) [30, 35, 20, 33, 34], *depth-first unary degree sequence* (DFUDS) [11, 21, 31], *level-order unary degree sequence* (LOUDS) [30, 17], and *tree covering* (TC) [21, 25, 18]. Of these, BP-based representations "combine good time- and space-performance with rich functionality" in practice [8], and we use BP in our solutions. BP is a way of linearising the tree by emitting

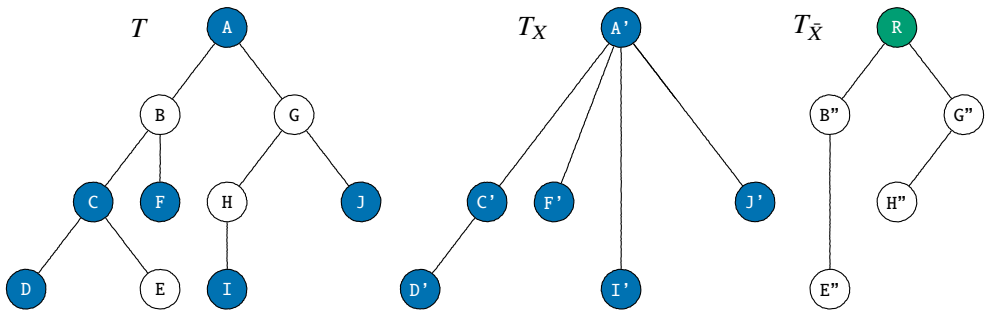


Figure 1 Tree extraction. Original tree (left), extracted tree T_X (middle), and extraction of the complement of X, tree $T_{\bar{X}}$ (right). The blue shaded nodes in T form the set X. In the tree T_X, node `C'` corresponds to node `C` in the original tree T, and node `C'` in the extracted tree T_X is the T_X-view of nodes `C` and `E` in the original tree T. Finally, node `C` in T is the T-source of the node `C'` in T_X. Extraction of the complement, $T_{\bar{X}}$, demonstrates the case of adding a dummy root `R`.

"(" upon first entering a node and ")" upon exiting, having explored all its descendants during the preorder traversal of the tree. For example, ((((())())())(((())()))) would be a BP-sequence for the tree T in Figure 1.

As shown in [35, 33, 34], an ordinal tree T on n nodes can be represented in $2n+o(n)$ bits of space to support the following operations in $\mathcal{O}(1)$ time, for any node $x \in T$: $\texttt{child}(T, x, i)$, the i-th child of x; $\texttt{depth}(T, x)$, the number of ancestors of x; $\texttt{LCA}(T, x, y)$, the lowest common ancestor of nodes $x, y \in T$; and $\texttt{level\_anc}(T, x, i)$, the i^{th} lowest ancestor of x.

Tree extraction. Tree extraction [28] selects a subset X of nodes while maintaining the underlying hierarchical relationship among the nodes in X. Given a subset X of tree nodes called *extracted nodes*, an *extracted tree* T_X can be obtained from the original tree T through the following procedure. Let $v \notin X$ be an arbitrary node. The node v and all its incident edges in T are removed from T, thereby exposing the parent p of v and v's children, $v_1, v_2, \ldots, v_k$. Then the nodes $v_1, v_2, \ldots, v_k$ (in this order) become new children of p, occupying the contiguous segment of positions starting from the (old) position of v. After thus removing all the nodes $v \notin X$, we have $T_X \equiv F_X$, if the forest F_X obtained is a tree; otherwise, a dummy root r holds the roots of the trees in F_X (in the original left-to-right order) as its children. (The symmetry between X and $\bar{X} = V \setminus X$ brings about the *complement* $T_{\bar{X}}$ of the extracted tree T_X.) An original node $x \in X$ of T and its copy, x', in T_X are said to *correspond* to each other; also, x' is the T_X*-view* of x, and x is the T*-source* of x'. The T_X-view of a node $y \in T$ (y is not necessarily in X) is generally defined to be the node $y' \in T_X$ corresponding to the lowest node in $\mathcal{A}(y) \cap X$. In this paper, tree extraction is predominantly used to classify nodes into categories, and the labels assigned indicate the weight ranges the original weights belong to.

Figure 1 gives an example of an extracted tree, views and sources.

3 Data Structures for Path Queries

This section gives the design details of the `whp` and `ext` data structures.

3.1 Data structures based on heavy-path decomposition

We now describe the approach of [40], which is based on heavy-path decomposition [43].

Heavy-path decomposition (HPD) imposes a structure on a tree. In HPD, for each non-leaf node, a *heavy child* is defined as the child whose subtree has the maximum cardinality. HPD of a tree T with root r is a collection of disjoint chains, first of which is obtained by always following the heavy child, starting from r, until reaching a leaf. The subsequent chains are obtained by the same procedure, starting from the non-visited nodes closest to the root (ties broken arbitrarily). The crucial property is that any root-to-leaf path in the tree encounters $\mathcal{O}(\lg n)$ distinct chains. A chain's *head* is the node of the chain that is closest to the root; a chain's tail is therefore a leaf.

Patil et al. [40] used HPD to decompose a path query into $\mathcal{O}(\lg n)$ queries in sequences. To save space, they designed the following data structure to represent the tree and its HPD. If x is the head of a chain ϕ, all the nodes in ϕ have a (conceptual) *reference* pointing to x, while x points to itself. A *reference count* of a node x (denoted as rc_x) stands for the number of times a node serves as a reference. Obviously, only heads feature non-zero reference counts – precisely the lengths of their respective chain. The reference counts of all the nodes are stored in unary in preorder in a bitmap $B = 10^{rc_1}10^{rc_2}\ldots10^{rc_n}$ using $2n + o(n)$ bits. Then, one has that $rc_x = \texttt{rank}_0(\texttt{B}, \texttt{select}_1(\texttt{B}, \texttt{x}+1)) - \texttt{rank}_0(\texttt{B}, \texttt{select}_1(\texttt{B}, \texttt{x}))$. The topology of the original tree T is represented succinctly in another $2n + o(n)$ bits. In addition, they

encode the HPD structure of T using a new tree T' that is obtained from T via the following transformation. All the non-head nodes become leaves and are directly connected to their respective heads; the heads themselves (except the root) become children of the references of their original parents. All these connections are established respecting the preorder ranks of the nodes in the original tree T. Namely, a node farther from the head attaches to it only after the higher-residing nodes of the chain have done so. This transformation preserves the original preorder ranks. On T', operation `ref(x)` is supported, which returns the head of chain to which the node x in the original tree belongs.

To encode weights they call C_x the weight-list of x if it collects, in preorder, all the nodes for which x is a reference. Thus, a non-head node's list is empty; a head's list spells the weights in the relevant chain. Define $C = C_1C_2 \dots C_n$. Then, in C, the weight of $x \in T$ resides at position

$$1 + \texttt{select}_1(\texttt{B}, \texttt{ref(x)}) - \texttt{ref(x)} + \texttt{depth(x)} - \texttt{depth(ref(x))} \tag{1}$$

(where `depth(x)` and `ref(x)` are provided by T and T', respectively). C is then encoded in a wavelet tree (WT). To answer a query, T, T', B, and Equation (1) are used to partition the query path into $\mathcal{O}(\lg n)$ sub-chains that it overlaps in HPD; and for each sub-chain, one computes the interval in C storing the weights of the nodes in the chain. I_m denotes the set of intervals computed. Precisely, for a node x, one uses B to find out whether x is the head of its chain; if not, the parent of x in T' returns one (say, y). Then Equation (1) maps the path $A_{x,y}$ to its corresponding interval in C. One proceeds to the next chain by fetching the (original) parent of y, using T. Then, the WT is queried with $\mathcal{O}(\lg n)$ simultaneous (i) range quantile (for `PM`); or (ii) orthogonal range $2d$ queries (for `PC` and `PR`).

Range quantile query over a collection of ranges is accomplished via a straightforward extension of the algorithm of Gagie et al. [19]. One descends the wavelet tree W_C maintaining a set of current weights $[a, b]$ (initially $[\sigma]$), the current node v (initially the root of W_C), and I_m. When querying the current node v of W_C with an interval $[l_j, r_j] \in I_m$, one finds out, in $\mathcal{O}(1)$ time, how many weights in the interval are lighter than the mid-point c of $[a, b]$, and how many of them are heavier. The sum of these values then determines which subtree of W_C to descend to. There being $\mathcal{O}(\lg \sigma)$ levels in W_C, and spending $\mathcal{O}(1)$ time for each segment in I_m, the overall running time is $\mathcal{O}(\lg n \lg \sigma)$. `PC`/`PR` proceed by querying each interval, independently of the others, with the standard $2d$ search over W_C.

3.2 Data structures based on tree extraction

The solution by He et al. [28] is based on performing a hierarchy of tree extractions, as follows. One starts with the original tree T weighted over $[\sigma]$, and extracts two trees $T_0 = T_{1,m}$ and $T_1 = T_{m+1,\sigma}$, respectively associated with the intervals $I_0 = [1, m]$ and $I_1 = [m+1, \sigma]$, where $m = \lfloor \frac{1+\sigma}{2} \rfloor$. Then both T_0 and T_1 are subject to the same procedure, stopping only when the current tree is weight-homogeneous. We refer to the tree we have started with as the *outermost* tree.

The key insight of tree extraction is that the number of nodes n' with weights from I_0 on the path from u to v equals $\texttt{n}' = \texttt{depth}_0(\texttt{u}_0) + \texttt{depth}_0(\texttt{v}_0) - 2 \cdot \texttt{depth}_0(\texttt{z}_0) + 1_{\texttt{w(z)} \in \texttt{I}_0}$, where $\texttt{depth}_0(\cdot)$ is the depth function in T_0, $z = LCA(u, v)$, u_0, v_0, z_0 are the T_0-views of u, v, and z, and 1_{pred} is 1 if predicate *pred* is TRUE, and 0 otherwise. The key step is then, for a given node x, how to efficiently find its 0/1-*parent*, whose purpose is analogous to a `rank`-query when descending down the WT. Consider a node $x \in T$ and its T_0-view x_0. The corresponding node $x' \in T$ of $x_0 \in T_0$ is then called 0-*parent* of x. The 1-parent is defined analogously. Supporting 0/1-parents in compact space is one of the main implementation

challenges of the technique, as storing the views explicitly is space-expensive. In [28], the hierarchy of extractions is done by dividing the range not to 2 but $f = \mathcal{O}(\lg^\epsilon n)$ parts, with $0 < \epsilon < 1$ being a constant. They classify the nodes according to weights using these $f = \lceil \lg^\epsilon n \rceil$ labels and use tree covering to represent the tree with small labels in order to find T_α-views for arbitrary $\alpha \in [\sigma]$, in constant time. They also use this representation to identify, in constant time, which extractions to explore. Therefore, at each of the $\mathcal{O}(\lg \sigma / \lg\lg n)$ levels of the hierarchy of extractions, constant time work is done, yielding an $\mathcal{O}(\lg \sigma / \lg\lg n)$-time algorithm for `PC`. Space-wise, it is shown that each of the $\mathcal{O}(\lg \sigma / \lg\lg n)$ levels can be stored in $2n + nH_0(W) + o(n \lg \sigma)$ bits of space in total (where W is the multiset of weights on the level) which, summed over all the levels, yields $nH_0(W_T) + \mathcal{O}(n \lg \sigma / \lg\lg n)$ bits of space. The components of this optimal result, however, use word-parallel techniques that are unlikely to be practical. In addition, one of the components, tree covering (TC) for trees labeled over $[\sigma]$, $\sigma = \mathcal{O}(\lg^\epsilon n)$ has not been implemented and experimentally evaluated even for unlabeled versions thereof. Finally, lookup tables for the word-RAM data structures may either be rendered too heavy by word alignment, or too slow by the concomitant arithmetic for accessing its entries. In practice, small blocks of data are usually explicitly scanned [8]. However, we can see no fast way to scan small labeled trees. At the same time, a generic multi-parentheses approach [37] would spare the effort altogether, immediately yielding a $4n \lg \sigma + 2n + o(n \lg \sigma)$-bit encoding of the tree, with $\mathcal{O}(1)$-time support for 0/1-parents. We achieve instead $3n \lg \sigma + o(n \lg \sigma)$ bits of space, as we proceed to describe next.

We store $2n + o(n)$ bits as a regular BP-structure S of the original tree, in which a 1-bit represents an opening parenthesis, and a 0-bit represents a closing one, and mark in a separate length-n bitmap B the *types* (i.e. whether it is a 0- or 1-node) of the n opening parentheses in S. The type of an opening parenthesis at position i in S is thus given by $\texttt{access}(\texttt{B}, \texttt{rank}_1(\texttt{S}, \texttt{i}))$. Given S and B, we find the $t \in \{0, 1\}$-parent of v with an approach described in [27]. For completeness, we outline in Algorithm 1 how to locate the T_t-view of a node v.

Algorithm 1 Locate the view of $v \in T$ in T_t, where T_t is the extraction from T of the t-nodes.

Require: $t \in \{0, 1\}$

1: **function** VIEW_OF(v, t)
2: **if** $\texttt{B}[\texttt{v}] == \texttt{t}$ **then** ▷ v is a t-node itself
3: **return** $\texttt{B.rank}_\texttt{t}(\texttt{v})$
4: $\lambda \leftarrow \texttt{rank}_\texttt{t}(\texttt{B}, \texttt{v})$ ▷ how many t-nodes precede v?
5: **if** $\lambda == 0$ **then**
6: **return** `null`
7: $\texttt{u} \leftarrow \texttt{select}_\texttt{t}(\texttt{B}, \lambda)$ ▷ find the λ^{th} t-node
8: **if** $\texttt{LCA}(\texttt{u}, \texttt{v}) == \texttt{u}$ **then**
9: **return** $\texttt{B.rank}_\texttt{t}(\texttt{u})$
10: $\texttt{z} \leftarrow \texttt{LCA}(\texttt{u}, \texttt{v})$ ▷ z is LCA of a t-node u and a non-t-node v
11: **if** $\texttt{z} == \texttt{null}$ **or** $\texttt{B}[\texttt{z}] == \texttt{t}$ **then** ▷ z is a t-node $\Rightarrow \nexists$ t-parent closer to v
12: **return** $\texttt{B.rank}_\texttt{t}(\texttt{z})$ ▷ or `null`
13: $\lambda \leftarrow \texttt{rank}_\texttt{t}(\texttt{B}, \texttt{z})$ ▷ how many t-nodes precede z?
14: $\texttt{r} \leftarrow \texttt{select}_\texttt{t}(\texttt{B}, \lambda + 1)$ ▷ the first t-descendant of z
15: $\texttt{z}_\texttt{t} \leftarrow \texttt{rank}_\texttt{t}(\texttt{B}, \texttt{r})$ ▷ z_t is the T_t-view of r
16: $\texttt{p} \leftarrow \texttt{T}_\texttt{t}\texttt{.parent}(\texttt{z}_\texttt{t})$ ▷ p can be `null` if z_t is 0
17: **return** p

First, find the number of t-nodes preceding v (line 4). If none exists (line 5), we are done; otherwise, let u be the t-node immediately preceding v (line 7). If u is an ancestor of v, it is the answer (line 9); else, set $z = LCA(u, v)$. If z is a t-node, or non-existent (because the tree is actually a forest), then return z or `null`, respectively. Otherwise (z exists and not a t-node), in line 14 we find the first t-descendant r of z (it exists because of u). This descendant cannot be a parent of v, since otherwise we would have found it before. It must share though the same t-parent with v. We map this descendant to a node z_t in T_t (line 15). Finally, we find the parent of z_t in T_t (line 16).

The combined cost of S and B is $2n + n + o(n) = 3n + o(n)$ bits. At each of the $\lg \sigma$ levels of extraction, we encode 0/1-labeled trees in the same way, so the total space is $3n \lg \sigma + o(n \lg \sigma)$ bits.

Query algorithms in the `ext` data structure proceed within the generic framework of extracting T_0 and T_1. Let $n' = |P_{u_0,v_0}|$. In `PM`, we recurse on T_0 if $k < n'$, for a query that asks for a node with the k^{th} smallest weight on the path P_{u_0,v_0}; otherwise, we recurse on T_1 with $k \leftarrow k - n'$ and u_1, v_1. We stop upon encountering a tree with homogeneous weights. This gives an $\mathcal{O}(\lg \sigma)$-time algorithm. We defer the details to the full version of the paper.

A procedure for the `PC` and `PR` is essentially similar to that for the `PM` problem. We maintain two nodes, u and v, as the query nodes with respect to the current extraction T, and a node z as the lowest common ancestor of u and v in the current tree T. Initially, $u, v \in T$ are the original query nodes, and T is the outermost tree. Correspondingly, z is the LCA of the nodes u and v in the original tree; we determine the weight of z and store it in w, which is passed down the recursion. Let $[a, b]$ be the query interval, and $[p, q]$ be the current range of weights of the tree. Initially, $[p, q] = [\sigma]$. First, we check whether the current interval $[p, q]$ is contained within $[a, b]$. If so, the entire path $A_{u,z} \cup A_{v,z}$ belongs to the answer. Here, we also check whether $w \in [a, b]$. Then we recurse on T_t ($t \in \{0, 1\}$) having computed the corresponding T_t-views of the nodes u, v, and z, and with the corresponding current range. This algorithm's running time is $\mathcal{O}(\lg \sigma)$. The details are deferred to the full version.

To summarize, the variant of `ext` that we design here uses $3n \lg \sigma + o(n \lg \sigma)$ bits to support `PM` and `PC` in $\mathcal{O}(\lg \sigma)$ time, and `PR` in $\mathcal{O}((1 + \kappa) \lg \sigma)$ time. Compared to the original succinct solution [28] based on tree extraction, our variant uses about 3 times the space with a minor slow-down in query time, but is easily implementable using bitmaps and BP, both of which have been studied experimentally (see e.g. [8] and [37] for an extensive review).

4 Experimental Results

We now conduct experimental studies on data structures for path queries.

4.1 Implementation

For ease of reference, we outline the data structures implemented in Table 1.

Naïve approaches (both plain pointer-based `nv`/`nv`$^{\mathtt{L}}$ and succinct `nv`$^{\mathtt{c}}$) resolve a query on the path $P_{x,y}$ by explicitly traversing it from x to y. At each encountered node, we either (i) collect its weight into an array (for `PM`); (ii) check if its weight is in the query range (for `PC`); (iii) if the check in (ii) succeeds, we collect the node into a container (for `PR`). In `PM`, we subsequently call a standard *introspective selection* algorithm [36] over the array of collected weights. Depths and parent pointers, explicitly stored at each node, guide in upwards traversal from x and y to their common ancestor. Plain pointer-based tree topologies are stored using *forward-star* [6] representation. In `nv`$^{\mathtt{L}}$, we equip `nv` with the linear-space and $\mathcal{O}(1)$-time LCA-support structure of [10].

Table 1 The implemented data structures and the abbreviations used to refer to them.

	Symbol	Description
pointer-based	`nv`	Naïve data structure in Section 4.1
	$\texttt{nv}^{\texttt{L}}$	Naïve data structure in Section 4.1, augmented with $\mathcal{O}(1)$ query-time LCA of [10]
	$\texttt{ext}^{\dagger}$	A solution based on tree extraction [28] in Section 2
	$\texttt{whp}^{\dagger}$	A non-succinct version of the wavelet tree- and heavy-path decomposition-based solution of [40] in Section 3.
succinct	$\texttt{nv}^{\texttt{c}}$	Naïve data structure of Section 4.1, using succinct data structures to represent the tree structure and weights
	$\texttt{ext}^{\texttt{c}}$	$3n \lg \sigma + o(n \lg \sigma)$-bits-of-space scheme for tree extraction of Section 3.2, with compressed bitmaps
	$\texttt{ext}^{\texttt{p}}$	$3n \lg \sigma + o(n \lg \sigma)$-bits-of-space scheme for tree extraction of Section 3.2, with uncompressed bitmaps
	$\texttt{whp}^{\texttt{c}}$	Succinct version of `whp`, with compressed bitmaps
	$\texttt{whp}^{\texttt{p}}$	Succinct version of `whp`, with uncompressed bitmaps

Succinct structures $\texttt{ext}^{\texttt{c}}/\texttt{ext}^{\texttt{p}}/\texttt{whp}^{\texttt{c}}/\texttt{whp}^{\texttt{p}}$ are implemented with the help of the succinct data structures library `sdsl-lite` of Gog et al. [22]. To implement `whp` and the practical variant of `ext` we designed in Section 3.2, two types of bitmaps are used: a compressed bitmap [41] (implemented in `sdsl::rrr_vector` of `sdsl-lite`) and plain bitmap (implemented in `sdsl::bit_vector` of `sdsl-lite`). For $\texttt{nv}^{\texttt{c}}$, the weights are stored using $\lceil \lg \sigma \rceil$ bits each in a sequence and the structure theoretically occupies $2n + n \lg \sigma + o(n \lg \sigma)$ bits. For uniformity, across our data structures, tree navigation is provided solely by a BP representation based on [21] (implemented in `sdsl::bp_support_gg`), chosen on the basis of our benchmarks.

Plain pointer-based implementation $\texttt{ext}^{\dagger}$ is an implementation of the solution by He et al. [28] for the pointer-machine model, which uses tree extraction. In it, the views $x_0 \in T_0$, $x_1 \in T_1$ for each node that arises in the hierarchy of extractions, as well as the depths in T, are explicitly stored. Similarly, $\texttt{whp}^{\dagger}$ is a plain pointer-based implementation of the data structure by Patil et al. [40]. The relevant source code is accessible at `https://github.com/serkazi/tree_path_queries`.

4.2 Experimental setup

The platform used is a 128GiB RAM, Intel(R) Xeon(R) Gold 6234 CPU 3.30GHz server running 4.15.0-54-generic 58-Ubuntu SMP x86_64 kernel. The build is due to `clang-8` with `-g,-O2, -std=c++17,mcmodel=large,-NDEBUG` flags. Our datasets originate from geographical information systems (GIS). In Table 2, the relevant meta-data on our datasets is given.

We generated query paths by choosing a pair uniformly at random (u.a.r.). To generate a range of weights, $[a, b]$, we follow the methodology of [16] and consider `large`, `medium`, and `small` configurations: given K, we generate the left bound $a \in [W]$ u.a.r., whereas b is generated u.a.r. from $[a, a + \lceil \frac{W-a}{K} \rceil]$. We set $K = 1, 10$, and 100 for respectively `large`, `medium`, and `small`. To counteract skew in weight distribution in some of the datasets, when generating the weight-range $[a, b]$, we in fact generate a pair from $[n]$ rather than $[\sigma]$ and map the positions to the sorted list of input-weights, ensuring the number of nodes covered by the generated weight-range to be proportional to K^{-1}.

Table 2 Datasets metadata. DEM stands for Digital Elevation Model, and MST for minimum spanning tree. Weights are over $\{0, 1, \ldots, \sigma - 1\}$, and H_0 is the entropy of the multiset of weights. In DEM, elevation (in meters) is used as weights. For `eu.mst.osm`, distance in meters between locations, and for `eu.mst.dmcs`, travel time between locations, for a proprietary "car" profile in tenths of a second, are used as weights.

	num nodes	diameter	σ	$\log \sigma$	H_0	Description
eu.mst.osm	27,024,535	109,251	121,270	16.89	9.52	An MST we constructed over map of Europe [39]
eu.mst.dmcs	18,010,173	115,920	843,781	19.69	8.93	An MST we constructured over European road network [1]
eu.emst.dem	50,000,000	175,518	5020	12.29	9.95	An Euclidean MST we constructed over DEM of Europe [4]
mrs.emst.dem	30,000,000	164,482	29,367	14.84	13.23	An Euclidean MST we constructed over DEM of Mars [3]

Table 3 (upper) Space occupancy of our data structures, in bits per node, when loaded into memory; (lower) peak memory usage (**m** in bits per node) during construction and construction time (t in seconds) shown as $\mathbf{m}/t$.

	Dataset	nv	nv$^{\mathsf{L}}$	whp†	ext†	nv$^{\mathsf{c}}$	ext$^{\mathsf{c}}$	ext$^{\mathsf{p}}$	whp$^{\mathsf{c}}$	whp$^{\mathsf{p}}$
space	eu.mst.osm	406.3	972.1	3801	5943	21.71	59.85	75.74	21.71	34.42
	eu.mst.dmcs	406.4	974.0	4274	6768	34.46	82.16	106.0	29.69	48.77
	eu.emst.dem	394.1	988.5	3342	4613	19.64	45.41	59.15	19.64	31.66
	mrs.emst.dem	386.7	1005	3579	5383	17.35	51.71	66.02	17.35	28.80
peak/time	eu.mst.osm	**491.0**/1	**987.9**/5	**3785**/28	**9586**/47	**21.71**/1	**295.0**/23	**295.0**/23	**1347**/62	**1347**/61
	eu.mst.dmcs	**439.8**/1	**1002**/4	**4403**/19	**12382**/37	**29.69**/1	**399.7**/18	**399.7**/18	**1360**/42	**1360**/42
	eu.emst.dem	**401.0**/2	**1021**/10	**3460**/47	**5286**/67	**19.64**/1	**287.6**/32	**287.6**/32	**1333**/115	**1333**/115
	mrs.emst.dem	**392.4**/1	**1016**/5	**3719**/30	**6027**/46	**17.35**/1	**269.3**/22	**269.3**/22	**1337**/69	**1337**/69

4.3 Space performance and construction costs

A single data structure we implement (be it ever `nv`-, `ext`-, or `whp`-family), taken individually, answers all three types of queries (`PM`, `PC`, and `PR`). Hence, we consider space consumption first.

The upper part of the Table 3 shows the space usage of our data structures. The structures `nv`/nv$^{\mathsf{L}}$ are lighter than ext†/whp†, as expected. Adding fast LCA support doubles the space requirement for `nv`, whereas succinctness (nv$^{\mathsf{c}}$) uses up to 20 times less space than `nv`. The difference between ext† and whp†, in turn, is in explicit storage of the 0-views for each of the $\Theta(n \lg \sigma)$ nodes occurring during tree extraction. In whp†, by contrast, $\texttt{rank}_0$ is induced from $\texttt{rank}_1$ (via subtraction) – hence the difference in the empirical sizes of the otherwise $\Theta(n \lg \sigma)$-word data structures.

The succinct nv$^{\mathsf{c}}$'s empirical space occupancy is close to the information-theoretic minimum given by $\lg \sigma + 2$ (Table 2). The structures ext$^{\mathsf{c}}$/ext$^{\mathsf{p}}$ occupy about three times as much, which is consistent with the design of our practical solution (Section 3.2). It is interesting to note that the data structure whp$^{\mathsf{c}}$ occupies space close to bare succinct storage of the input alone (nv$^{\mathsf{c}}$). Entropy-compression significantly impacts both families of succinct structures, `whp` and `ext`, saving up to 20 bits per node when switching from plain bitmap to a compressed one. Compared to pointer-based solutions (`nv`/nv$^{\mathsf{L}}$/whp†/ext†), we note that ext$^{\mathsf{c}}$/ext$^{\mathsf{p}}$/whp$^{\mathsf{c}}$/whp$^{\mathsf{p}}$ still allow usual navigational operations on T, whereas the former shed this redundancy, to save space, after preprocessing.

Table 4 Average time to answer a query, from a fixed set of 10^6 randomly generated path median and path counting queries, in microseconds. Path counting queries are given in `large`, `medium`, and `small` configurations.

	Dataset	nv	$\texttt{nv}^{\texttt{L}}$	$\texttt{ext}^{\dagger}$	$\texttt{whp}^{\dagger}$	$\texttt{nv}^{\texttt{c}}$	$\texttt{ext}^{\texttt{c}}$	$\texttt{ext}^{\texttt{p}}$	$\texttt{whp}^{\texttt{c}}$	$\texttt{whp}^{\texttt{p}}$	
median	eu.mst.osm	658	475	4.22	6.10	7078	85.3	51.1	111	51.2	
	eu.mst.dmcs	566	412	5.16	6.28	6556	84.6	54.8	120	54.7	
	eu.emst.dem	710	436	4.44	5.10	9404	106	81.9	96.7	54.9	
	mrs.emst.dem	472	298	4.93	4.53	7018	124	97.0	88.3	49.5	
counting	eu.mst.osm	238	140	6.88	18.4	3553	247	167	139	56.9	large
	eu.mst.dmcs	204	121	7.31	19.7	3300	253	178	142	57.3	
	eu.emst.dem	338	195	5.97	11.5	4835	215	168	105	55.9	
	mrs.emst.dem	232	174	5.25	8.40	3614	206	164	91	49.3	
	eu.mst.osm	244	143	5.47	17.8	3555	213	146	129	54.2	medium
	eu.mst.dmcs	209	124	6.94	18.4	3297	224	160	133	56.5	
	eu.emst.dem	339	195	4.55	10.0	4840	178	140	100	54.9	
	mrs.emst.dem	237	143	5.91	8.74	3613	199	154	89.7	48.9	
	eu.mst.osm	239	139	5.25	15.4	3551	190	132	119	53.9	small
	eu.mst.dmcs	209	123	5.25	18.9	3300	206	148	126	55.2	
	eu.emst.dem	347	200	3.92	9.34	4832	154	124	94.9	53.2	
	mrs.emst.dem	238	144	4.82	7.41	3615	178	133	84.2	47.6	

Overall, the succinct $\texttt{whp}^{\texttt{p}}/\texttt{whp}^{\texttt{c}}/\texttt{ext}^{\texttt{p}}/\texttt{ext}^{\texttt{c}}$ perform very well, being all well-under 1 gigabyte for the large datasets we use. This suggests scalability: when trees are so large as not to fit into main memory, it is clear that the succinct solutions are the method of choice.

The lower part in Table 3 shows peak memory usage ($\mathbf{m}$, in bits per node) and construction time (t, in seconds), as $\mathbf{m}/t$. The structures $\texttt{ext}^{\texttt{p}}/\texttt{ext}^{\texttt{c}}$ are about three times faster than $\texttt{whp}^{\texttt{p}}/\texttt{whp}^{\texttt{c}}$ to build, and use four times less space at peak. This is expected, as `whp` builds two different structures (HPD and then WT). This is reversed for $\texttt{ext}^{\dagger}/\texttt{whp}^{\dagger}$; time-wise, as $\texttt{ext}^{\dagger}$ performs more memory allocations during construction (although our succinct structures are flattened into a heap layout, $\texttt{ext}^{\dagger}$ stores pointers to T_0/T_1; this is less of a concern for $\texttt{whp}^{\dagger}$, whose very purpose is tree linearisation).

4.4 Path median queries

The upper section of Table 4 records the mean time for a single median query (in µs) averaged over a fixed set of 10^6 randomly generated queries.

Succinct structures $\texttt{whp}^{\texttt{c}}/\texttt{whp}^{\texttt{p}}/\texttt{ext}^{\texttt{c}}/\texttt{ext}^{\texttt{p}}$ perform well on these queries, with a slow-down of at most 20-30 times from their respective pointer-based counterparts. Using entropy-compression degrades the speed of `whp` almost twice. Overall, the families `whp` and `ext` seem to perform at the same order of magnitude. This is surprising, as in theory `whp` should be a factor of $\lg n$ slower. The discrepancy is explained partly by small average number of segments in HPD, averaging 9 ± 2 for our queries. (The number of unary-degree nodes in our datasets is 35%-56%, which makes smaller number of heavy-path segments prevalent. We did not use trees with few unary-degree nodes in our experiments, as the height of such trees are not large enough to make constructing data structures for path queries worthwhile.) When the queries are partitioned by the number of chains in the HPD, the curves for $\texttt{ext}^{\texttt{c}}/\texttt{ext}^{\texttt{p}}$ stay flat whereas those for $\texttt{whp}^{\texttt{c}}/\texttt{whp}^{\texttt{p}}$ grow linearly. In `eu.mst.dmcs`, if the query path is

partitioned into 9 chains, $\texttt{ext}^{\texttt{P}}$ is only slightly faster than $\texttt{whp}^{\texttt{P}}$, whereas with the query path containing 19 chains, $\texttt{ext}^{\texttt{P}}$ is about 2.3 times so. (The details are given in the full version of the paper.) This suggests to favour the `ext` family over `whp` whenever performance in the worst case is important. Furthermore, navigational operations in $\texttt{ext}^{\texttt{P}}/\texttt{ext}^{\texttt{c}}$ and $\texttt{whp}^{\texttt{P}}/\texttt{whp}^{\texttt{c}}$, despite of similar theoretical worst-case guarantees, involve different patterns of using the `rank/select` primitives. For one, $\texttt{whp}^{\texttt{P}}/\texttt{whp}^{\texttt{c}}$ does not call LCA during the search – mapping of the search ranges when descending down the recursion is accomplished by a single `rank` call, whereas $\texttt{ext}^{\texttt{P}}/\texttt{ext}^{\texttt{c}}$ computes LCA at each level of descent (for its its own analog of `rank` – the view computation in Algorithm 1). Now, LCA is a non-trivial combination of `rank/select` calls. The difference between $\texttt{ext}^{\texttt{P}}/\texttt{ext}^{\texttt{c}}$ and $\texttt{whp}^{\texttt{P}}/\texttt{whp}^{\texttt{c}}$ will therefore become pronounced in a large enough tree; with tangible HPD sizes, the constants involved in (albeit theoretically $\mathcal{O}(1)$) LCA calls are overcome by $\lg n$.

Naïve structures $\texttt{nv}/\texttt{nv}^{\texttt{L}}/\texttt{nv}^{\texttt{c}}$ are visibly slower in `PM` than in `PC` (considered in Section 4.5), as expected – for `PM`, having collected the nodes encountered, we also call a selection algorithm. In `PC`, by contrast, neither insertions into a container nor a subsequent search for median are involved. Navigation and weights-uncompression in $\texttt{nv}^{\texttt{c}}$ render it about 10 times slower than its plain counterpart. The $\texttt{nv}^{\texttt{L}}$ being little less than twice faster than its LCA-devoid counterpart, `nv`, is explained by the latter effectively traversing the query path twice – once to locate the LCA, and again to answer the query proper. Any succinct solution is about 4-8 times faster than the fastest naïve, $\texttt{nv}^{\texttt{L}}$.

4.5 Path counting queries

The lower section in Table 4 records the mean time for a single counting query (in µs) averaged over a fixed set of 10^6 randomly generated queries, for `large`, `medium`, and `small` setups.

Structures $\texttt{nv}/\texttt{nv}^{\texttt{L}}/\texttt{nv}^{\texttt{c}}$ are insensitive to κ, as the bottleneck is in physically traversing the path.

Succinct structures $\texttt{whp}^{\texttt{P}}/\texttt{whp}^{\texttt{c}}$ and $\texttt{ext}^{\texttt{P}}/\texttt{ext}^{\texttt{c}}$ exhibit decreasing running times as one moves from `large` to `small` – as the query weight-range shrinks, so does the chance of branching during the traversal of the implicit range tree. The fastest (uncompressed) $\texttt{whp}^{\texttt{P}}$ and the slowest (compressed) $\texttt{ext}^{\texttt{c}}$ succinct solutions differ by a factor of 4, which is intrinsically larger constants in $\texttt{ext}^{\texttt{c}}$'s implementation compounded with slower `rank/select` primitives in compressed bitmaps, at play. The uncompressed $\texttt{whp}^{\texttt{P}}$ is about 2-3 times faster than $\texttt{ext}^{\texttt{P}}$, the gap narrowing towards the `small` setup. The slowest succinct structure, $\texttt{ext}^{\texttt{c}}$, is nonetheless competitive with the $\texttt{nv}/\texttt{nv}^{\texttt{L}}$ already in `large` configuration, with the advantage of being insensitive to tree topology.

In $\texttt{ext}^{\dagger}$-$\texttt{whp}^{\dagger}$ pair, $\texttt{whp}^{\dagger}$ is 2-3 times slower. This is predictable, as the inherent $\lg n$-factor slow-down in $\texttt{whp}^{\dagger}$ is no longer offset by differing memory access patterns – following a pointer "downwards" (i.e. 0/1-view in $\texttt{ext}^{\dagger}$ and $\texttt{rank}_{0/1}()$ in $\texttt{whp}^{\dagger}$) each require a single memory access.

4.6 Path reporting queries

Table 5 records the mean time for a single reporting query (in µs) averaged over a fixed set of 10^6 randomly generated queries, for `large`, `medium`, and `small` setups.

Structures $\texttt{whp}^{\texttt{c}}/\texttt{whp}^{\texttt{P}}/\texttt{ext}^{\texttt{c}}/\texttt{ext}^{\texttt{P}}$ recover each reported node's weight in $\mathcal{O}(\lg \sigma)$ time. Thus, when $\lg n \ll \kappa$, the query time for both `ext` and `whp` families become $\mathcal{O}(\kappa \cdot \log \sigma)$. (At this juncture, a caveat is in order: design of `whp`'s in Section 3.1 allows a `PR`-query to

Table 5 Average time to answer a path reporting query, from a fixed set of 10^6 randomly generated path reporting queries, in microseconds. The queries are given in `large`, `medium`, and `small` configurations. Average output size for each group is given in column κ.

Dataset	κ	nv	nv$^{\mathtt{L}}$	ext†	whp†	nv$^{\mathtt{c}}$	ext$^{\mathtt{c}}$	ext$^{\mathtt{p}}$	whp$^{\mathtt{c}}$	whp$^{\mathtt{p}}$	
eu.mst.osm	9,840	356	256	184	70.7	3766					large
eu.mst.dmcs	9,163	309	224	147	66.8	3485					large
eu.emst.dem	14,211	389	241	140	77.5	4926					large
mrs.emst.dem	10,576	267	178	89.2	55.1	3668					large
eu.mst.osm	1,093	322	222	43.7	28.8	3706					medium
eu.mst.dmcs	1,090	277	196	34.0	29.7	3434					medium
eu.emst.dem	1,464	354	206	32.1	20.1	4880					medium
mrs.emst.dem	1,392	250	151	22.1	15.6	3639					medium
eu.mst.osm	182	311	212	13.8	19.0	3685	1965	485	795	226	small
eu.mst.dmcs	236	271	193	13.2	21.0	3529	2518	632	1043	292	small
eu.emst.dem	215	353	203	10.2	12.7	4873	1276	378	590	205	small
mrs.emst.dem	117	242	145	8.88	9.57	3632	881	278	475	162	small

only return the *index in the array* C – not the original preorder identifier of the node, as does the `ext`.) When κ is large, therefore, these structures are not suitable for use in `PR`, as nv/nv$^{\mathtt{L}}$/nv$^{\mathtt{c}}$are clearly superior ($\mathcal{O}((1+\kappa)\lg n)$ *vs* $\mathcal{O}(\kappa)$), and we confine the experiments for ext$^{\mathtt{c}}$/ext$^{\mathtt{p}}$/whp$^{\mathtt{c}}$/whp$^{\mathtt{p}}$ to the `small` setup only (bottom-right corner in Table 5).

We observe that the succinct structures ext$^{\mathtt{p}}$ and whp$^{\mathtt{p}}$ are competitive with nv/nv$^{\mathtt{L}}$, in `small` setting: informally, time saved in locating the nodes to report is used to uncompress the nodes' weights (whereas in nv/nv$^{\mathtt{L}}$ the weights are explicit). Between the succinct `ext` and `whp`, clearly `whp` is faster, as `select()` on a sequence as we go up the wavelet tree tend to have lower constant factors than the counterpart operation on BP.

Structures whp† and ext† exhibit same order of magnitude in query time, with the former being sometimes about 2 times faster on non-`small` setups. Among two somewhat intertwined reasons, one is that whp† returns an index to the permuted array, as noted above. (Converting to the original id would necessitate an additional memory access.) Secondly, in the implicit range tree during the $2d$ search in whp†, when the current range is contained within the query interval, we start reporting the node weights by merely incrementing a counter – position in the WT sequence. By contrast, in such situations ext† iterates through the nodes being reported calling `parent()` for the current node, which is one additional memory access compared to whp† (at the scale of µs, this matters). Indeed, operations on trees tend to be little more expensive than similar operations on sequences.

Structures nv/nv$^{\mathtt{L}}$/nv$^{\mathtt{c}}$ are less sensitive to the query weight range's magnitude, since they simply scan the path along with pushing into a container. The differences in running time in Table 5 between the configurations are thus accounted for by container operations' cost. Naïve structures' query times for `PR` being dependent solely on the query path's length, they are unfeasible for large-diameters trees (whereas they may be suitable for shallow ones, e.g. originating from "small-world" networks).

Overall evaluation. We visualize in Figure 2 some typical entries in Table 4 to illustrate the structures clustering along the space/time trade-offs: nv/nv$^{\mathtt{L}}$ (upper-left corner) are lighter in terms of space, but slow; pointer-based ext†/whp† are very fast, but space-heavy. Between the two extremes of the spectrum, the succinct structures ext$^{\mathtt{c}}$/ext$^{\mathtt{p}}$/whp$^{\mathtt{c}}$/whp$^{\mathtt{p}}$, whose mutual configuration is shown magnified in inner rectangle, are space-economical and yet offer fast query times.

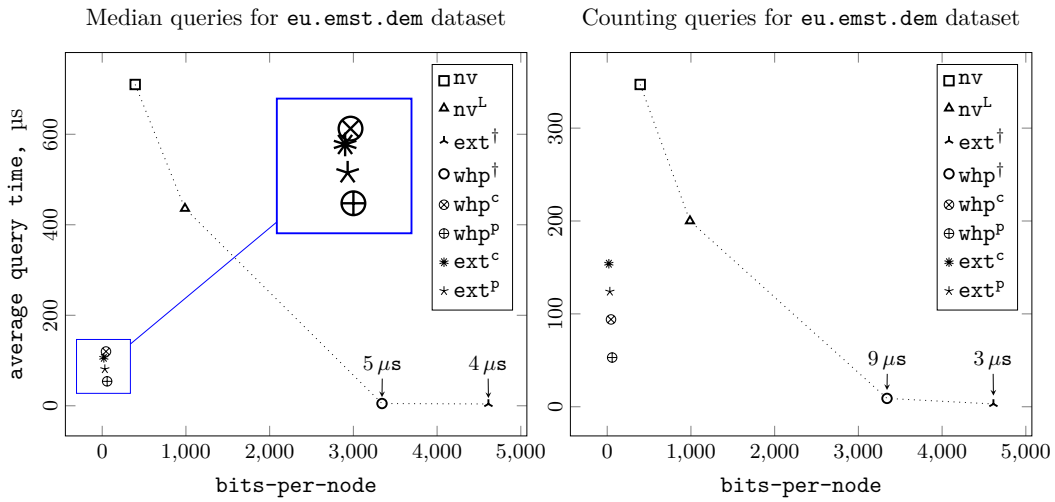


Figure 2 Visualization of some of the entries in Table 4. Inner rectangle magnifies the mutual configuration of the succinct data structures `whp`$^{\mathtt{p}}$,`whp`$^{\mathtt{c}}$,`ext`$^{\mathtt{p}}$, and `ext`$^{\mathtt{c}}$. The succinct naïve structure `nv`$^{\mathtt{c}}$ is not shown.

5 Conclusion

We have designed and experimentally evaluated recent algorithmic proposals in path queries in weighted trees, by either faithfully replicating them or offering practical alternatives. Our data structures include both plain pointer-based and succinct implementations. Our succinct realizations are themselves further specialized to be either plain or entropy-compressed.

We measure both query time and space performance of our data structures on large practical sets. We find that the succinct structures we implement offer an attractive alternative to plain pointer-based solutions, in scenarios with critical space- and query time-performance and reasonable tolerance to slow-down. Some of the structures we implement (`whp`$^{\mathtt{c}}$) occupy space equal to bare compressed storage (`nv`$^{\mathtt{c}}$) of the object and yet offer fast queries on top of it, while another structure (`ext`$^{\mathtt{c}}$/`ext`$^{\mathtt{p}}$) occupies space comparable to `nv`$^{\mathtt{c}}$, offers fast queries and low peak memory in construction. While `whp` succinct family performs well in average case, thus offering attractive trade-offs between query time and space occupancy, `ext` is robust to the structure of the underlying tree, and is therefore recommended when strong worst-case guarantees are vital.

Our design of the practical succinct structure based on tree extraction (`ext`) results in a theoretical space occupancy of $3n \lg \sigma + \mathcal{O}(n \lg \sigma)$ bits, which helps explain its somewhat higher empirical space cost when compared to the succinct `whp` family. At the same time, verbatim implementation of the space-optimal solution by He et al. [28] draws on components that are likely to be cumbersome in practice. For the path query types considered in this study, therefore, realization of the theoretically time- and space-optimal data structure – or indeed some feasible alternative thereof – remains an interesting open problem in algorithm engineering.

References

1 KIT roadgraphs. https://i11www.iti.kit.edu/information/roadgraphs. Accessed: 07/12/2018.

2 ltree module for PostreSQL RDBMS. https://www.postgresql.org/docs/current/ltree.html. Accessed: 10/01/2020.

3 MOLA Mars Orbiter Laser Altimeter data from NASA Mars Global Surveyor. https://planetarymaps.usgs.gov/mosaic/Mars_MGS_MOLA_DEM_mosaic_global_463m.tif. Accessed: 10/01/2019.

4 SRTM Shuttle Radar Topography Mission. http://srtm.csi.cgiar.org/srtmdata/. Accessed: 10/01/2019.

5 Andrés Abeliuk, Rodrigo Cánovas, and Gonzalo Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013. doi:10.3390/a6020319.

6 Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows - theory, algorithms and applications*. Prentice Hall, 1993.

7 Noga Alon and Baruch Schieber. Optimal preprocessing for answering on-line product queries. Technical report, Tel-Aviv University, 1987.

8 Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010*, pages 84–97, 2010. doi:10.1137/1.9781611972900.9.

9 Diego Arroyuelo, Francisco Claude, Reza Dorrigiv, Stephane Durocher, Meng He, Alejandro López-Ortiz, J. Ian Munro, Patrick K. Nicholson, Alejandro Salinger, and Matthew Skala. Untangled monotonic chains and adaptive range search. *Theor. Comput. Sci.*, 412(32):4200–4211, 2011. doi:10.1016/j.tcs.2011.01.037.

10 Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005. doi:10.1016/j.jalgor.2005.08.001.

11 David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. doi:10.1007/s00453-004-1146-6.

12 Nieves R. Brisaboa, Guillermo de Bernardo, Roberto Konow, Gonzalo Navarro, and Diego Seco. Aggregated 2d range queries on clustered points. *Inf. Syst.*, 60:34–49, 2016. doi:10.1016/j.is.2016.03.004.

13 Timothy M. Chan, Meng He, J. Ian Munro, and Gelin Zhou. Succinct indices for path minimum, with applications. *Algorithmica*, 78(2):453–491, 2017. doi:10.1007/s00453-016-0170-7.

14 Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. Orthogonal range searching on the ram, revisited. In *Computational Geometry, 27th ACM Symposium, SoCG 2011, Paris, France, June 13-15, 2011. Proceedings*, pages 1–10, 2011. doi:10.1145/1998196.1998198.

15 Bernard Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2(1):337–361, November 1987. doi:10.1007/BF01840366.

16 Francisco Claude, J. Ian Munro, and Patrick K. Nicholson. Range queries over untangled chains. In *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*, pages 82–93, 2010. doi:10.1007/978-3-642-16321-0_8.

17 O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Engineering the LOUDS succinct tree representation. In *Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings*, pages 134–145, 2006. doi:10.1007/11764298_12.

18 Arash Farzan and J. Ian Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, 2014. doi:10.1007/s00453-012-9664-0.

19 Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In *String Processing and Information Retrieval, 16th International Symposium, SPIRE 2009, Saariselkä, Finland, August 25-27, 2009, Proceedings*, pages 1–6, 2009. doi:10.1007/978-3-642-03784-9_1.

20 Richard F. Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3):231–246, 2006. doi:10.1016/j.tcs.2006.09.014.

21 Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Trans. Algorithms*, 2(4):510–534, 2006. doi:10.1145/1198513.1198516.

22 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.

23 Torben Hagerup. Parallel preprocessing for path queries without concurrent reading. *Inf. Comput.*, 158(1):18–28, 2000. doi:10.1006/inco.1999.2814.

24 Meng He and Serikzhan Kazi. Path and ancestor queries over trees with multidimensional weight vectors. In *30th International Symposium on Algorithms and Computation, ISAAC 2019, December 8-11, 2019, Shanghai University of Finance and Economics, Shanghai, China*, pages 45:1–45:17, 2019. doi:10.4230/LIPIcs.ISAAC.2019.45.

25 Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct ordinal trees based on tree covering. *ACM Trans. Algorithms*, 8(4):42:1–42:32, 2012. doi:10.1145/2344422.2344432.

26 Meng He, J. Ian Munro, and Gelin Zhou. Path queries in weighted trees. In *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, pages 140–149, 2011. doi:10.1007/978-3-642-25591-5_16.

27 Meng He, J. Ian Munro, and Gelin Zhou. A framework for succinct labeled ordinal trees over large alphabets. *Algorithmica*, 70(4):696–717, 2014. doi:10.1007/s00453-014-9894-4.

28 Meng He, J. Ian Munro, and Gelin Zhou. Data structures for path queries. *ACM Trans. Algorithms*, 12(4):53:1–53:32, 2016. doi:10.1145/2905368.

29 Kazuki Ishiyama and Kunihiko Sadakane. A succinct data structure for multidimensional orthogonal range searching. In *2017 Data Compression Conference, DCC 2017, Snowbird, UT, USA, April 4-7, 2017*, pages 270–279, 2017. doi:10.1109/DCC.2017.47.

30 Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.

31 Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees with applications. *J. Comput. Syst. Sci.*, 78(2):619–631, 2012. doi:10.1016/j.jcss.2011.09.002.

32 Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

33 Hsueh-I Lu and Chia-Chi Yeh. Balanced parentheses strike back. *ACM Trans. Algorithms*, 4(3):28:1–28:13, 2008. doi:10.1145/1367064.1367068.

34 J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theor. Comput. Sci.*, 438:74–88, 2012. doi:10.1016/j.tcs.2012.03.005.

35 J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

36 David R. Musser. Introspective sorting and selection algorithms. *Softw., Pract. Exper.*, 27(8):983–993, 1997. doi:10.1002/(SICI)1097-024X(199708)27:8\%3C983::AID-SPE117\%3E3.0.CO;2-\%23.

37 Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016. URL: http://www.cambridge.org/de/academic/subjects/computer-science/

algorithmics-complexity-computer-algebra-and-computational-g/compact-data-structures-practical-approach?format=HB.

38 Gonzalo Navarro and Alberto Ordóñez Pereira. Faster compressed suffix trees for repetitive collections. *ACM Journal of Experimental Algorithmics*, 21(1):1.8:1–1.8:38, 2016. doi:10.1145/2851495.

39 OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org, 2017.

40 Manish Patil, Rahul Shah, and Sharma V. Thankachan. Succinct representations of weighted trees supporting path queries. *J. Discrete Algorithms*, 17:103–108, 2012. doi:10.1016/j.jda.2012.08.003.

41 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

42 A. Rényi and G. Szekeres. On the height of trees. *Journal of the Australian Mathematical Society*, 7(4):497–507, 1967.

43 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, June 1983. doi:10.1016/0022-0000(83)90006-5.