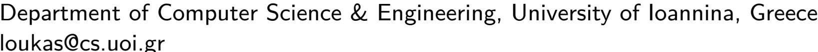
21st International Symposium on Experimental Algorithms

SEA 2023, July 24–26, 2023, Barcelona, Spain

Edited by

Loukas Georgiadis

Editors

Loukas Georgiadis

Department of Computer Science & Engineering, University of Ioannina, Greece
loukas@cs.uoi.gr

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

ISBN 978-3-95977-279-2

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/978-3-95977-279-2.

Publication date
July, 2023

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.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.SEA.2023.0

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.

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University – Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB and Nanyang Technological University, SG)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface
Loukas Georgiadis 0:vii

Steering Committee
.... 0:ix

Organization
.... 0:xi

Papers

Engineering a Preprocessor for Symmetry Detection
Markus Anders, Pascal Schweitzer, and Julian Stieß 1:1–1:21

Fast Reachability Using DAG Decomposition
Giorgos Kritikakis and Ioannis G. Tollis 2:1–2:17

Partitioning the Bags of a Tree Decomposition into Cliques
Thomas Bläsius, Maximilian Katzmann, and Marcus Wilhelm 3:1–3:19

Subset Wavelet Trees
Jarno N. Alanko, Elena Biagi, Simon J. Puglisi, and Jaakko Vuohtoniemi 4:1–4:14

Engineering Shared-Memory Parallel Shuffling to Generate Random Permutations In-Place
Manuel Penschuck 5:1–5:20

Proxying Betweenness Centrality Rankings in Temporal Networks
Ruben Becker, Pierluigi Crescenzi, Antonio Cruciani, and Bojana Kodric 6:1–6:22

Simple Runs-Bounded FM-Index Designs Are Fast
Diego Díaz-Domínguez, Saska Dönges, Simon J. Puglisi, and Leena Salmela 7:1–7:16

Noisy Sorting Without Searching: Data Oblivious Sorting with Comparison Errors
Ramtin Afshar, Michael Dillencourt, Michael T. Goodrich, and Evrim Ozel 8:1–8:18

Optimizing over the Efficient Set of a Multi-Objective Discrete Optimization Problem
Satya Tamby and Daniel Vanderpooten 9:1–9:13

Solving Directed Feedback Vertex Set by Iterative Reduction to Vertex Cover
Sebastian Angrick, Ben Bals, Katrin Casel, Sarel Cohen, Tobias Friedrich, Niko Hastrich, Theresa Hradilak, Davis Issac, Otto Kißig, Jonas Schmidt, and Leo Wendt 10:1–10:14

CompDP: A Framework for Simultaneous Subgraph Counting Under Connectivity Constraints
Kengo Nakamura, Masaaki Nishino, Norihito Yasuda, and Shin-ichi Minato 11:1–11:20

Multilinear Formulations for Computing a Nash Equilibrium of Multi-Player Games
Miriam Fischer and Akshay Gupte 12:1–12:14

21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Integer Programming Formulations and Cutting Plane Algorithms for the Maximum Selective Tree Problem
Ömer Burak Onar, Tınaz Ekim, and Z. Caner Taşkın 13:1–13:18

A Graph-Theoretic Formulation of Exploratory Blockmodeling
Alexander Bille, Niels Grüttemeier, Christian Komusiewicz, and Nils Morawietz 14:1–14:20

FREIGHT: Fast Streaming Hypergraph Partitioning
Kamal Eyubov, Marcelo Fonseca Faraj, and Christian Schulz 15:1–15:16

Arc-Flags Meet Trip-Based Public Transit Routing
Ernestine Großmann, Jonas Sauer, Christian Schulz, and Patrick Steil 16:1–16:18

Greedy Heuristics for Judicious Hypergraph Partitioning
Noah Wahl and Lars Gottesbüren 17:1–17:16

Hierarchical Relative Lempel-Ziv Compression
Philip Bille, Inge Li Gørtz, Simon J. Puglisi, and Simon R. Tarnow 18:1–18:16

Exact and Approximate Range Mode Query Data Structures in Practice
Meng He and Zhen Liu 19:1–19:22

Efficient Yao Graph Construction
Daniel Funke and Peter Sanders 20:1–20:20

Maximum Coverage in Sublinear Space, Faster
Stephen Jaud, Anthony Wirth, and Farhana Choudhury 21:1–21:20

Preface

We are pleased to present the collection of papers accepted for presentation at the 21th edition of the International Symposium on Experimental Algorithms (SEA 2023) which was held in Barcelona from 24th July 2023 to 26th July 2023. SEA, previously known as Workshop on Experimental Algorithms (WEA), is an international forum for researchers in the area of the design, analysis, and experimental evaluation and engineering of algorithms, as well as in various aspects of computational optimization and its applications (telecommunications, transport, bioinformatics, cryptography, learning methods, etc.). The symposium aims at attracting papers from both the Computer Science and the Operations Research/Mathematical Programming communities. Submissions to SEA are requested to present significant contributions supported by experimental evaluation, methodological issues in the design and interpretation of experiments, the use of heuristics and meta-heuritics, or application-driven case studies that deepen the understanding of the complexity of a problem. A main goal of SEA is also the creation of a friendly environment that can lead to and ease the establishment or strengthening of scientific collaborations and exchanges among attendees. For this reason, the symposium solicits high-quality original research papers (including significant work-in-progress) on any aspect of experimental algorithms. Each submission that was made to SEA 2023 was reviewed by at least three Program Committee members or external reviewers. After a careful peer review and evaluation process, 21 papers were accepted for presentation and for inclusion in the LIPIcs proceedings, according to the reviewers' recommendations. The acceptance rate was 50%. The scientific program of the symposium also includes presentations by two keynote speakers: Monika Henzinger (Research Group Theory and Applications of Algorithms, Universität Wien) and Pankaj K. Agarwal (Levine Science Research Center, Duke University). Since last year, the conference reintroduced a best paper award. The SEA 2023 best paper award was selected by the Program Committee. Based on the committee's careful assessment, the best paper was selected to be "FREIGHT: Fast Streaming Hypergraph Partitioning" by Kamal Eyubov, Marcelo Fonseca Faraj and Christian Schulz. We congratulate the authors for receiving this award. The 21th edition of SEA was organized by the Universitat Politècnica de Catalunya (UPC). We thank Maria J. Blesa, Amalia Duch, Guillem Rodríguez, and Maria Serna for the organization of the symposium. We also thank the UPC staff, Sito Ibáñez Escudero (ICT and Innovation Support Service), Ana Ibáñez Julià (Administration and Services staff at the UTG ICT Area), and Gabriel Verdejo Álvarez (RDLAB, Computer Science Department) for their support, and the faculty of Universitat Politècnica de Catalunya - BarcelonaTech for providing us with the facilities for the conference. Moreover, we would like to thank the SEA steering committee for giving us the opportunity to host SEA 2023. Special thanks to Ulrich Meyer, chair of the steering committee, for his valuable help. Finally, we express our gratitude to the members of the Program Committee as well as the external reviewers for their support, collaboration, and excellent work.

Barcelona, July 2023
Loukas Georgiadis

21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Steering Committee

- Gianlorenzo D'Angelo (Gran Sasso Science Institute, Italy)
- Domenico Cantone (Università degli Studi di Catania, Italy)
- David Coudert (INRIA, France)
- Simone Faro (Università di Catania, Italy)
- Ulrich Meyer (Goethe University Frankfurt, Germany) [**chair**]
- Emanuele Natale (CNRS, Universite Cote d'Azur, I3S, INRIA, France)
- Gonzalo Navarro (University of Chile, Chile)
- Cynthia Phillips (Sandia National Laboratories, USA)
- Simon Puglisi (University of Helsinki, Finland)
- Christian Schulz (Heidelberg Univeristy, Germany)
- Sabine Storandt (University of Konstanz, Germany)
- Laurent Viennot (INRIA, France)
- Bora Uçar (CNRS, Laboratoire LIP, Lyon, France)
- Dorothea Wagner (Karlsruhe Institute of Technology, Germany)
- Christos Zaroliagis (University of Patras, Greece)

21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Organization

Program Chair

- Loukas Georgiadis (University of Ioannina, Greece)

Program Committee

- Hideo Bannai (Tokyo Medical and Dental University, Japan)
- Gerth Brodal (Aarhus University, Denmark)
- Kevin Buchin (Technische Universität Dortmund, Germany)
- Mateus De Oliveira Oliveira (Stockholm University, Sweden and University of Bergen, Norway)
- Donatella Firmani (Sapienza University of Rome, Italy)
- Andrew V. Goldberg (USA)
- Yan Gu (University of California, Riverside, USA)
- Meng He (Dalhousie University, Canada)
- Giuseppe Italiano (LUISS, Italy)
- Spyros Kontogiannis (University of Patras, Greece)
- Luigi Laura (Uninenettuno, Italy)
- Leo Liberti (LIX CNRS, École Polytechnique, Institut Polytechnique de Paris, France)
- Matthias Mnich (TUHH - Hamburg University of Technology, Germany)
- André Nusser (University of Copenhagen, Denmark)
- Charis Papadopoulos (University of Ioannina, Greece)
- Vicky Papadopoulou-Lesta (European University Cyprus)
- Nikos Parotsidis (Google Research, Switzerland)
- Ignaz Rutter (University of Passau, Germany)
- Stavros Sintos (University of Illinois at Chicago, USA)
- Przemyslaw Uznanski (Pathway, Poland)
- Renato Werneck (Amazon, USA)
- Anthony Wirth (University of Melbourne, Australia)
- Helen Xu (Lawrence Berkeley National Laboratory, USA)

External Reviewers

Wagner Alan Aparecido da Rocha, Daniel Porumbel, Jonas Silva, Sven Mallach, Tobias Stamm, Václav Rozhoň, Morteza Monemizadeh, Monique Teillaud, Niels Grüttemeier, Serikzhan Kazi, Jens Kristian, Refsgaard Schou, Arghya Bhattacharya, Simone Zanella, Brian Wheatman, Frédéric Simard, Jessica Shi, Kaiyu Wu, Jerin George Mathew, Daniil Tsokaktsis, Stefano Leucci, Esther Galby, Yixiang Fang, Matthias Kaul, Rahul Raychaudhury, Shweta Jain, Laura Codazzi, Pingan Cheng, Kunal Dutta, Max Deppert, David Fischer, Athanasios Konstantinidis, Steffan Sølvsten, Sabine Storandt, Matthias Pfretzschner, Evangelos Kosinas, Lorenzo Balzotti, Simon D. Fink, Christian Komusiewicz, Casper Rysgaard, Jose Fuentes, Keisuke Goto, Yihan Sun, Rolf Svenning, André van Renssen, Dionysios Kefallinos, Christian Konrad, Martin Costa

Engineering a Preprocessor for Symmetry Detection

Markus Anders
TU Darmstadt, Germany

Pascal Schweitzer
TU Darmstadt, Germany

Julian Stieß
University of Koblenz-Landau, Germany

Abstract

State-of-the-art solvers for symmetry detection in combinatorial objects are becoming increasingly sophisticated software libraries. Most of the solvers were initially designed with inputs from combinatorics in mind (NAUTY, BLISS, TRACES, DEJAVU). They excel at dealing with a complicated core of the input. Others focus on practical instances that exhibit sparsity. They excel at dealing with comparatively easy but extremely large substructures of the input (SAUCY). In practice, these differences manifest in significantly diverging performances on different types of graph classes.

We engineer a preprocessor for symmetry detection. The result is a tool designed to shrink sparse, large substructures of the input graph. On most of the practical instances, the preprocessor improves the overall running time significantly for many of the state-of-the-art solvers. At the same time, our benchmarks show that the additional overhead is negligible.

Overall we obtain single algorithms with competitive performance across all benchmark graphs. As such, the preprocessor bridges the disparity between solvers that focus on combinatorial graphs and large practical graphs. In fact, on most of the practical instances the combined setup significantly outperforms previous state-of-the-art.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases graph isomorphism, automorphism groups, symmetry detection, preprocessors

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.1

Supplementary Material *Software (Source Code)*: `https://github.com/markusa4/sassy` archived at `swh:1:dir:ba57bf62762f6c5d0bd51ce07862a70df70c8468`

Funding Supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EngageS: grant No. 820148).

Acknowledgements We thank Marc E. Pfetsch and Christopher Hojny for giving us further insights into the user-side of symmetry detection software, as well as providing us with the MIP2017 graphs.

1 Introduction

Exploitation of symmetries is an indispensable instrument in a vast number of algorithmic application areas such as SAT [20, 5, 13], SMT [12], QBF [21], CSP [15], ILP [25, 27, 17] and many more. However, in order to exploit symmetries, we have to compute them first.

Many types of objects can be modelled efficiently as graphs, so that the objects' symmetries correspond to the symmetries of the graph. This includes formulas, equation systems, finite relational structures, and many more (see [29]). Hence, computing the symmetries of these objects reduces to computing symmetries of graphs. We refer to the act of computing the symmetries of a graph as *symmetry detection*.

© Markus Anders, Pascal Schweitzer, and Julian Stieß;
licensed under Creative Commons License CC-BY 4.0

State-of-the-art symmetry detection tools are NAUTY [26], SAUCY [11], BLISS [19], TRACES [26], and DEJAVU [7]. Given as input a vertex-colored graph, they output all the symmetries of the graph. All state-of-the-art tools are based around the so-called individualization-refinement (IR) paradigm. Yet, they substantially differ in the applied search strategies, pruning invariants, symmetry handling, and various other heuristics (see [26, 7]). This is also reflected in diverging performances on different graph classes.

We want to highlight two examples where the diverging performance between the solvers is notable, namely "practical graphs" and "combinatorial graphs". For large practical graphs, such as graphs arising in SAT, QBF, MIP, or road networks, the solver SAUCY outperforms all other solvers significantly (see, e.g., the results in [5] or the benchmarks of this paper in Section 9). Indeed, designed with satisfiability-checking in mind, SAUCY has been delicately engineered specifically for these types of graphs. Intuitively, graphs arising from practical applications tend to be large in size but comparatively simple in their structure. On the other hand, on almost all graph classes that are difficult relative to their size (e.g., projective planes, CFI graphs, and other regular combinatorial objects) TRACES and DEJAVU readily outperform other solvers due to their more sophisticated search strategies (see [7] and [26] for a more nuanced discussion). In Figure 1, we demonstrate the large disparity between SAUCY and DEJAVU on a difficult graph class from combinatorics and a class of practical graphs.

Only having solvers available that are geared towards specific types of graphs is of course an undesirable situation. This for example means we have to choose a solver and thus understand the type of input we are faced with. Also, we will struggle with inputs that are combinations of the different kinds of graphs. Quite naturally, it is desirable instead to have a single solver performing well on all graphs.

A commonly used paradigm to make solvers for computational problems more widely applicable is to add a preprocessor. The use of preprocessors has indeed already led to countless success stories, in particular in SAT, QBF or MaxSAT [14, 10, 24]. In these applications, it is nowadays standard to apply a preprocessor to all inputs. In contrast to this, to date, no preprocessor has been available for symmetry detection. In fact McKay and Piperno [26] explicitly highlight that in their opinion "graphs of [particular types] ought to be handled by preprocessing" before using their tools.

Beyond increasing performance, there are various other benefits to having a preprocessor. Firstly, the problem of initially simplifying the graph can be tackled independently from the design of the main solver. This is especially desirable since implementations of state-of-the-art symmetry detection solvers are complex and detailed descriptions of the inner workings largely unavailable. Secondly, in turn, a preprocessor could even reduce the complexity of solver implementations if certain cases are reliably handled before running the solver. Lastly, implementing strategies in a common preprocessor makes them available to all the solvers simultaneously.

Given the lack of an existing preprocessor for symmetry detection, TRACES, for example, has complicated subroutines that simplify some low-degree vertices before (and sometimes during) the computation (see the implementation [2]). Overall, the question is whether it is possible to design a common preprocessor that can simplify inputs and is beneficial to all state-of-the-art solvers.

Contribution. We implement the first preprocessor SASSY for symmetry detection. It is compatible by design with all state-of-the-art symmetry detection tools. Our benchmarks (Section 9) corroborate that solver configurations using the preprocessor significantly outperform state-of-the-art on many practical graph classes. At the same time, the preprocessor introduces only a negligible overhead.

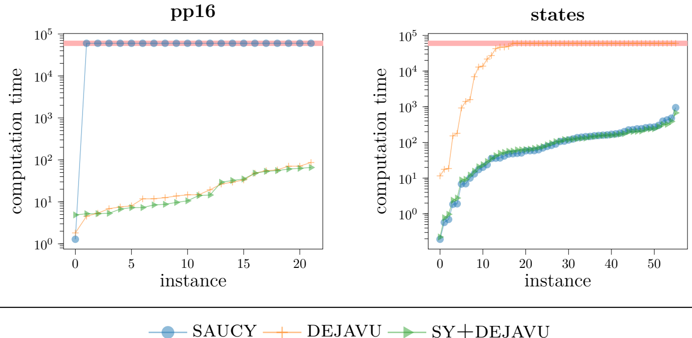


Figure 1 Comparing solvers on difficult combinatorial graphs (**pp16**) and large practical graphs (**states**). Timeout is $60s$ (red bar). SY+DEJAVU refers to DEJAVU with the preprocessor of this paper.

The preprocessor bridges the disparity that exists between solvers that focus on difficult combinatorial graphs (TRACES, DEJAVU, BLISS, NAUTY) and those that focus on large practical graphs (SAUCY). Through the use of the preprocessor, the former kind of solvers now outperform SAUCY on most practical graphs.

Techniques. The preprocessor implements mainly techniques to handle graphs that are sparse, both in the input and output (e.g., practical graphs). In particular, it is made up of the following building blocks which we discuss throughout the paper:

1. A framework to capture reduction techniques for input graphs. In particular, it captures the reconstruction of symmetries from the reduced graph back to the input graph, both theoretically and practically (Section 4).
2. A technique to efficiently remove vertices of degree 0 and 1 (Section 5.1 and Section 5.2).
3. Partial removal of degree 2 vertices avoiding the introduction of colored or directed edges (Section 5.3).
4. An individualization-refinement-based probing technique for “sparse automorphisms” (Section 6).
5. Exploiting connected components and homogeneous connections using the concept of quotient graphs (Section 7).

While SASSY is the first universal preprocessor for symmetry detection, we want to remark that a flavor of (2) is already implemented in TRACES. All the techniques other than (2) are novel contributions, however, we do want to mention that (4) and (5) draw some inspiration from existing techniques of solvers. We explain this in detail in the respective sections.

2 Philosophy of the Preprocessor

When designing a preprocessor, one of the main challenges is to map out which techniques and methods fall within the responsibility of the preprocessor and which task should be resolved by the main algorithm. Another delicate matter are the preprocessor/main solver and the user/preprocessor interfaces. In the design of our preprocessor we were guided by conceptual principles as well as technical requirements.

Conceptual principles. On a conceptual level, our goal is to design efficient preprocessing subroutines that simplify the task of computing symmetries. Naturally, a preprocessor should only apply procedures that are comparatively fast in relation to the running time of the main algorithm.

Figure 2 Our proposed preprocessor/main solver and user/preprocessor interfaces. The preprocessor may already determine some (or all) symmetries of G during graph reduction. The reduced instance is then passed on to the main solver.

The design of our preprocessor is centered around the so-called *color refinement* algorithm. Color refinement is a powerful heuristic for symmetry detection. It is continuously and repeatedly applied in all state-of-the-art solvers. Thus, procedures that run within or close to color-refinement-time are safe to apply.

The general idea of the preprocessor is to remove substructures of the graph that are already "basically resolved" by an application of color refinement. The main difficulty lies in detecting and exploiting these substructures as efficiently as possible. Essentially, any part that can be handled efficiently ought to be carefully handled using precisely the right technique.

Overall, we need to balance efficiency, effectiveness, and generality for our subroutines.

Technical requirements. On a technical level, we want our preprocessor to be compatible with all state-of-the-art solvers. Hence, we need to use an interface that is universal for all the existing tools. All tools read vertex-colored graphs and output symmetries. Hence, this is the interface that the preprocessor uses as well.

The preprocessor reads a vertex-colored graph and outputs a reduced vertex-colored graph passed to a main solver. Moreover, the preprocessor may already determine some or all of the symmetries, and immediately outputs these to the user. There is one more technicality: symmetries of the reduced graph which are computed by the main solver are, by definition, not symmetries of the original graph. To rectify this, the preprocessor employs a backward-translation (i.e., a form of postprocessing) to lift symmetries that were discovered by the main solver back to being symmetries of the original input graph.

Our design is illustrated in Figure 2.

3 Preliminaries

A graph G is a finite, simple, undirected graph, unless stated otherwise. The neighborhood of a vertex v is denoted $N(v)$, its *degree* is $\deg(v) := |N(v)|$. For a set of vertices $V' \subseteq V(G)$ the *neighborhood* is the set $N[V'] \coloneqq (\bigcup_{v \in V'} N(v)) \setminus V'$.

A *coloring* of a graph G is a map $\pi \colon V(G) \to \mathcal{C}$ from the vertices to some set of colors. A (color) *class* C is a set $\pi^{-1}(c)$ of vertices of the same color. A coloring π is referred to as *discrete* whenever π is injective. In other words, in a discrete coloring each vertex has its own unique color. Unless stated otherwise, we work with *colored graphs* $G = (V, E, \pi)$ which consist of vertex set V, edge set E, and a coloring π. Slightly abusing notation the

pair (G, π) for an uncolored graph $G = (V, E)$ is identified with (V, E, π). For a subset of the vertices $V' \subseteq V$ the *induced subgraph* of $G = (V, E, \pi)$ is $G[V'] = (V', E', \pi|_{V'})$ where $E' = \{e \in E \mid e \subseteq V' \times V'\}$.

A bijection $\varphi : V \mapsto V$ is called an *automorphism* (symmetry) whenever $(\varphi(V), \varphi(E)) = (V, E)$ (applying φ element-wise to the vertices in the edges of E). If G is colored, φ also has to respect colors (i.e., satisfy $\pi(\varphi(v)) = \pi(v)$). The number of automorphisms can be exponential in the size of the graph. The symmetries form a permutation group under the composition operation. The *automorphism group* containing all automorphisms of a (colored) graph G is $\mathrm{Aut}(G)$. The *support* of an automorphism $\varphi \in \mathrm{Aut}(G)$ is $\mathrm{supp}(\varphi) := \{\varphi(x) \neq x \mid x \in V(G)\}$, i.e., vertices not fixed by the automorphism. A subset of automorphisms $S \subseteq \mathrm{Aut}(G)$ is a *generating set* of $\mathrm{Aut}(G)$, whenever exhaustively composing permutations of S leads to all elements of $\mathrm{Aut}(G)$. We write $\langle S \rangle = \mathrm{Aut}(G)$. This enables a concise encoding of $\mathrm{Aut}(G)$. Solvers generally only output a generating set of $\mathrm{Aut}(G)$.

3.1 Color Refinement

The color refinement algorithm is a well-studied procedure [8, 9, 26]. For a colored graph it splits apart colors in a specific way to produce a "finer" coloring. Crucially, this process does *not* change the symmetries of the graph.

Formally, a coloring π of a graph is *equitable* if for all pairs of (not necessarily distinct) color classes C_1, C_2, all vertices in C_1 have the same number of neighbors in C_2 (i.e., $|N(v) \cap C_2| = |N(v') \cap C_2|$ for all $v, v' \in C_1$.) Given a coloring π, color refinement computes an equitable refinement π' (i.e., an equitable coloring π' for which $\pi'(v) = \pi'(v')$ implies $\pi(v) = \pi(v')$). In fact, it computes the coarsest equitable refinement. Crucially, automorphisms of $G = (V, E, \pi)$ are also automorphisms of $G = (V, E, \pi')$ (and vice versa). It is thus beneficial and routine to work with π' instead of π. Color refinement can be implemented in such a way that it admits a worst case running time of $\Theta((n + m)(\log n))$ (see [9]). From an implementation perspective it is the most crucial subroutine and therefore highly engineered.

3.2 Quotient Graph

For an equitable coloring π of an (otherwise uncolored) graph G, the *quotient graph* $Q(G, \pi)$ captures information regarding the number of neighbors that vertices in one color class have in another color class. A quotient graph is a complete directed graph in which every vertex has a self-loop. The vertex set of $Q(G, \pi)$ is $V(Q(G, \pi)) := \pi(V(G))$, i.e., the set of colors of vertices under π. The vertices of $Q(G, \pi)$ are colored with the color they represent in G. We color the edge (c_1, c_2) with the number of neighbors a vertex color c_1 has of color c_2 (possibly $c_1 = c_2$). Recall that, since π is equitable, all vertices of c_1 have the same number of neighbors in c_2. Two graphs are *indistinguishable by color refinement* if and only if their quotient graphs with respect to the coarsest equitable coloring are equal (see e.g. [8]).

4 A Toolbox for Reducing Graphs

We now embark on our journey of describing techniques that simplify a graph for symmetry detection. The goal is always to efficiently reduce the number of vertices and edges of the graph. However, whenever we alter the graph, we need to make sure that either no symmetries are lost, or that we output the symmetries that would be lost immediately. Furthermore, we have to ensure that after preprocessing is done symmetries of the reduced graph can be

mapped back to symmetries of the original graph. After all, we are interested in symmetries of the original graph. In order to ease this process, we first lay out some general techniques that we use throughout the paper.

The first type of technique we describe modifies an input graph G on vertex set V to another graph G' with vertex set $V' \subseteq V$ so that

1. $\mathrm{Aut}(G)|_{V'} \subseteq \mathrm{Aut}(G')$ (symmetry preservation) and
2. $\mathrm{Aut}(G)|_{V'} \supseteq \mathrm{Aut}(G')$ (symmetry lifting) hold.

Here by $\mathrm{Aut}(G)|_{V'}$ we mean the set of maps obtained by restricting the domain of each $\varphi \in \mathrm{Aut}(G)$ to V' (and the range to $\varphi(V')$). If conditions (1) and (2) hold, V' must also be invariant under $\mathrm{Aut}(G)$.

Under these conditions the restriction to V' is a natural homomorphism $p\colon \mathrm{Aut}(G) \to \mathrm{Aut}(G')$. The orbit-stabilizer theorem (see [18, Theorem 2.16]) implies then that if $S' \subseteq \mathrm{Aut}(G)$ is a set of lifts of a generating set S of $\mathrm{Aut}(G')$, i.e. $p(S') = S$, then $\mathrm{Aut}(G) = \langle S', \ker(p)\rangle$ (where $\langle\Gamma\rangle$ denotes the group *generated* by Γ, see [30]). Here $\ker(p) = \{\varphi \in \mathrm{Aut}(G) \mid p(\varphi) \neq 1\}$ is the *kernel* of p and 1 denotes the identity.

Overall this enables us to separate the computation of $\mathrm{Aut}(G)$ into computing automorphisms of the removed parts of the graph and the automorphisms of the reduced graph. Crucial for the techniques is now that G' and a generating set of $\ker(p)$ can be efficiently computed from G, and that the set of lifts S' can be efficiently computed from a generating set of $\mathrm{Aut}(G')$. In particular, we require an efficient postprocessing technique for lifting of automorphisms to parts that were reduced, which is described in the following.

Canonical Representation Strings. During preprocessing, the parts we remove from the original graph might be symmetrical to (i.e., in the same orbit as) other parts of the graph. So, after symmetries of the reduced graph have been computed, we need to lift symmetries of the reduced graph to symmetries of the original graph. In particular, the lifted symmetries must map all the removed parts correctly. To simplify the lifting of symmetries we introduce *representation strings* associated with the remaining vertices. These encode the nature (i.e., the "isomorphism type") of the vertices that were removed. The encoding is stored in the color of a suitable vertex that remains. If a remaining vertex is then mapped to another vertex, the corresponding subgraphs represented by the strings are then mapped to each other in a canonical way.

We define this process formally through a *representation mapping* $\mathcal{R}(v) : V \mapsto V^*$ from the vertices to sequences of vertices as follows. Assume we have a graph $G := (V, E, \pi)$ which is reduced to $G' := (V', E', \pi')$ with $V' \subseteq V$ and $E' \subseteq E$. We require the following:

1. It holds that $\mathcal{R}(v) := vS$ with $S \in V^*$ for all $v \in V'$, i.e., each remaining vertex must represent itself first.
2. It holds that $\mathcal{R}(v) := \epsilon$ for all $v \in V \setminus V'$, i.e., a removed vertex does not represent any vertex.
3. For each deleted vertex $v \in V \setminus V'$ there is at most one $v' \in V'$ and at most one $i \in \mathbb{N}$ such that $v := \mathcal{R}(v')_i$, i.e., each deleted vertex is represented by at most one remaining vertex, once.

For each automorphism of the remaining graph $\varphi \in \mathrm{Aut}(G')$ we now define its *lifted bijection* $\varphi_{\mathcal{R}}(v) \in \mathrm{Sym}(V)$ (the symmetric group on V). First, we require that $\varphi(v) = v' \implies |\mathcal{R}(v)| = |\mathcal{R}(v')|$ holds, otherwise we can not construct a lifted bijection. We define $\varphi_{\mathcal{R}}(v) :=$

$$\begin{cases} \varphi(v) & \text{if } v \in V' \\ \mathcal{R}(\varphi(v'))_i & \text{if } v = \mathcal{R}(v')_i \text{ for } v' \in V', i \in \mathbb{N} \\ v & \text{if } v \neq \mathcal{R}(v')_i \text{ for all } v' \in V', i \in \mathbb{N}. \end{cases}$$

We call $\mathcal{R}$ a *canonical representation mapping* if $\varphi_{\mathcal{R}} \in \mathrm{Aut}(G)$ for all $\varphi \in \mathrm{Aut}(G')$.

We note by definition, canonical representation mappings can be chained, i.e., if we reduce a graph G multiple times, we can simply apply the respective canonical representation mappings in reverse until we reach an automorphism of G. We can even rewrite chained canonical representation mappings into a single map by essentially composing the functions. (More accurately we have to interpret strings of strings as simple strings using concatenation.)

Sparse Automorphisms and Restoration. A concept that we implicitly use throughout the following sections is sparse encodings of automorphisms. A conventional way to do this is the cycle notation of permutations, i.e., store only for each non-fixed element its image [18]. The precise encoding used is of no importance, however. Crucially, automorphisms ought to be encoded using space that is proportional to the size of their support, i.e., in $\mathcal{O}(|\operatorname{supp}(\varphi)|)$.

Using a canonical representation mapping $\mathcal{R}$ and sparse automorphism encodings, automorphisms of a reduced graph G' can be efficiently lifted to automorphisms of the original graph G. Indeed, lifts can be computed in time (and in space) linear in the size of the support of the lift, by replacing vertices by their represented strings.

▶ **Fact 1.** *Given $\varphi \in \operatorname{Aut}(G')$, the lift $\varphi_{\mathcal{R}} \in \operatorname{Aut}(G)$ can be computed in time $\mathcal{O}(|\operatorname{supp}(\varphi_{\mathcal{R}})|)$.*

Let us remark that often canonical representations in fact ensure that lifted supports are as small as possible. We say that a representation mapping $\mathcal{R}$ *respects kernel orbits* if it has the property that $v_1 \in \mathcal{R}(v) \Leftrightarrow v_2 \in \mathcal{R}(v)$ whenever v_1 and v_2 are in the same orbit of $\ker(p)$. All representations we describe subsequently respect kernel orbits.

▶ **Fact 2.** *If $\mathcal{R}$ respects kernel orbits then $p(\psi) = \varphi$ implies that $|\operatorname{supp}(\varphi_{\mathcal{R}})| \leq |\operatorname{supp}(\psi)|$.*

We should remark that none of the state-of-the-art solvers except for SAUCY feature an interface for sparse automorphisms, i.e., an interface that enables access to an automorphism in time $\mathcal{O}(|\operatorname{supp}(\varphi)|)$. Instead, access is only possible in $\Omega(|V|)$. If a user-application uses the interface for sparse automorphisms correctly, this can yield substantial running time benefits on graphs that contain a large number of sparse automorphisms (which is the case for many practical graphs). Most solvers internally incur a cost of $\Omega(|V|)$ to handle automorphisms anyway, in turn making the sparse interface unnecessary. Since this is not true for our preprocessor and to ensure potential running time benefits to user-applications, automorphisms found by the preprocessor are of course accessible in a sparse manner.

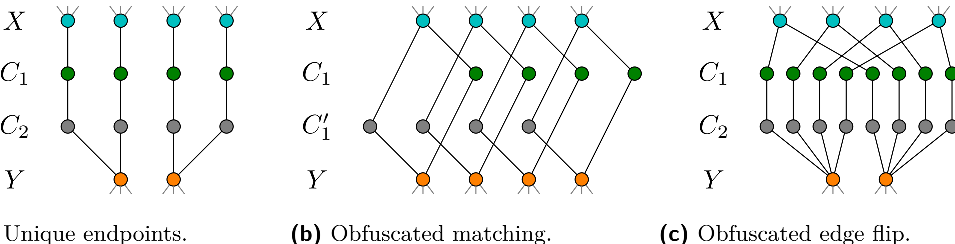
5 Removing low degree vertices

The first class of efficient reduction techniques we describe removes vertices of low degree. We propose strategies for vertices of degree $0, 1$ and 2. Techniques for preprocessing vertices of degree 0 and 1 can also be found in the implementation of TRACES [2]. The TRACES implementation for degree $0, 1$ differs from our proposed strategy in that it does *not* compute color refinement before removing degree 0 and 1 vertices.

5.1 Degree 0 Vertices

Preprocessing vertices of degree 0 (and analogously $n - 1$) is simple. The algorithm detects color classes consisting of vertices of degree 0. We let V' be the set of vertices of degree larger than 0. By simply removing vertices of degree 0 and not representing them in $\mathcal{R}$ at all, $\mathcal{R}$ indeed defines a canonical representation mapping.

The kernel $\ker(p)$ of the restriction p onto V' is computed as follows. For each color class of degree 0 vertices in G we output generators for the symmetric group on the class.



(a) Unique endpoints. **(b)** Obfuscated matching. **(c)** Obfuscated edge flip.

Figure 3 Reducible degree 2 patterns.

5.2 Degree 1 Vertices

Exhaustively removing all vertices of degree 1 (and analogously $n-2$) essentially removes all tree-like appendages from graphs. It is well-known that applying color refinement produces the orbit partitioning on these tree-like appendages – with the notable exception of not determining whether the roots of these appendages are in the same orbit or not.

We can remove degree 1 vertices recursively. Let G be a graph that contains degree 1 vertices. We describe G' and $\mathcal{R}$ where we remove a color class of degree 1 vertices.

Let C denote such a color class of degree 1 vertices. Since the coloring is equitable, all neighbors of vertices of C are in the same color class P. In case $P = C$ we have connected components of size 2. This case can be handled similar to the reduction of degree 0 vertices, so we assume $P \neq C$. We partition C into classes $C_1, \dots, C_m$ where $c \in C_i$ is adjacent to $p_i \in P$. For the representation mapping, we set $\mathcal{R}(p_i) := p_i C_i$ (where C_i may appear in arbitrary order). We set $G' := G \setminus \{C\}$. The coloring π remains unchanged. Note that π is still an equitable coloring for G'. The kernel $\ker(p)$ is the direct product of the symmetric group $\mathrm{Sym}(C_i)$ for each $i \in \{1, \dots, m\}$ (and points outside C are fixed). The process can then be repeated until all vertices of degree 1 are removed.

By construction, the reduction is symmetry preserving and symmetry lifting, thus it holds that $\mathrm{Aut}(G) = \langle S', \ker(p) \rangle$. As before, S is a generating set for $\mathrm{Aut}(G')$ and S' a corresponding set of lifts.

5.3 Degree 2 Vertices

If we were to allow graphs produced by our preprocessor to contain directed, colored edges, there is a simple reduction that removes all vertices of degree 2: we may encode the multiset of paths between two vertices v_1 and v_2 with $\deg(v_i) \geq 3$ whose internal vertices all have degree 2 as one directed, colored edge between v_1 and v_2 (see also [22, Proof of Lemma 15]).

There are, however, drawbacks to this approach: most solvers do not implement directed and colored edges. Since we want our preprocessor to be compatible with all modern solvers, this immediately disallows the use of directed, colored edges. Even when they do, using directed and colored edges comes at the price of additional overhead [28]. Intuitively, while removing all degree 2 vertices can cause a significant size-reduction, some of the complexity of the removed path is only shifted into the color encoding of the edges. In turn, we require refinements to take into account edge colors. This complicates color refinement, the central subroutine.

For these reasons, if possible, we prefer to remove degree 2 vertices in a way that does not require the introduction of directed or colored edges.

Non-branching paths with unique endpoint. We describe a heuristic which we found to be often applicable in practical data sets. It encodes paths with internal vertices of degree 2 that run between two color classes by a set of edges connecting the endpoints directly. However, it only does so if the set of paths can be reconstructed unambiguously from the set of edges. In particular, the inserted edges may not interfere with existing edges.

We detect paths of length t between distinct color classes X and Y whose internal vertices have degree 2. In each vertex of X exactly one such path should start (see Figure 3a). More formally, suppose $X = C_0, C_1, \dots, C_t, C_{t+1} = Y$ are colors so that (1) vertices in X do not have neighbors in Y, (2) for $i \in \{1, \dots, t\}$ vertices in C_i have degree 2, (3) for $i \in \{1, \dots, t\}$ vertices in C_i have a neighbor in C_{i-1} and C_{i+1}, and (4) every node in X has exactly one neighbor in C_1. Then we define $G' = (V', E')$ via $V' \coloneqq V - (C_1 \cup \dots \cup C_t)$ and $E' \coloneqq E(G[V']) \cup E''$, where E'' consists of pairs (x, y) for which there is a path $(x, c_1, \dots, c_t, y)$ with $c_i \in C_i$. The corresponding representation map is $\mathcal{R}(x) = xc_1c_2 \cdots c_t$, where $(x, c_1, \dots, c_t, y)$ is the unique path from x to some vertex $y \in Y$ with $c_i \in C_i$.

Note that the newly introduced edges E'' form a biregular bipartite graph between X and Y in which vertices of X have degree 1. It is not difficult to check that this yields a canonical representation map that respects kernel orbits.

Obfuscated Matchings. The preprocessor has special fast code for the particular case in which $|X| = |Y|$. In this case E'' encodes a perfect matching between X and Y.

A slight extension of the technique checks for other choices of C_i whether they also satisfy the required properties and yield exactly the same matching E''. In fact, if there is another matching via color classes $C'_1, \dots, C'_{t'}$ between X and Y which encodes E'' , we also delete vertices in the C'_i (see Figure 3b). The special purpose code uses arrays and can efficiently check whether matchings coincide.

We should mention that in the implementation, we only perform the check for paths of length $t = 1$ for obfuscated matchings. It turns out that the special case of $t = 1$ and in fact multiple such paths encoding the same matching is very common in particular on the MIP and SAT benchmarks.

Obfuscated Edge Flip. A case that also can be handled efficiently and is not covered by previous techniques is where X and Y are connected by $|X||Y|$ equally-colored, unique paths. In this case, each vertex $x \in X$ is connected to all $y \in Y$ by a path (see Figure 3c). It is easy to see that deleting all such paths is both symmetry preserving and symmetry lifting (this is related to the edge flip described in Section 7.1).

Formally, suppose $X = C_0, C_1, \dots, C_t, C_{t+1} = Y$ are colors so that (1) for $i \in \{1, \dots, t\}$ vertices in C_i have degree 2, (2) for $i \in \{1, \dots, t\}$ vertices in C_i have a neighbor in C_{i-1} and C_{i+1}, and (3) every node in X has exactly $|Y|$ neighbors in C_1, where the corresponding paths end in all $y \in Y$. The technique in turn removes all $C_0, C_1, \dots, C_t$ from the graph.

Let us now consider computing the lift of this reduction. Unfortunately, canonical representation strings are not sufficient to express the lift: we need to determine how $C_0, C_1, \dots, C_t$ are mapped, and this depends on *both* the vertices of X and Y. We can not simply attach $C_0, C_1, \dots, C_t$ to the canonical representation strings of one of the color classes. However, if we know how both X and Y are mapped, it is trivial to reconstruct the original symmetry: assume a symmetry maps $x \in X$ to x' and $y \in Y$ to y'. This just means that in the lift, we need to map the path connecting x to y to the path connecting x' to y'. Hence, the lift can still be computed very easily and efficiently.

In the implementation, we do write vertices of $C_0, C_1, \ldots, C_t$ into both the representation strings of X and Y, breaking the formal requirement of not having double entries. We use an encoding trick to denote the double entries, which triggers special code during the reconstruction of the symmetries.

Again, these types of degree 2 vertices can often be found in graphs stemming from SAT.

6 Probing for Sparse Automorphisms

We propose a strategy for probing for sparse automorphisms. If successful and automorphisms are discovered, we "divide them out", breaking the symmetry by *individualization*, i.e., giving a vertex a unique color. We give a brief, high-level description. The full description can be found in Appendix A.

Our strategy is inspired by a heuristic of SAUCY: for two colorings π_1, π_2 we may check whether interchanging vertices in color classes of size 1 (i.e., *singleton vertices*) of corresponding singleton colors and fixing all other vertices yields an automorphism of the graph. More formally, we define the permutation $\varphi_{\pi_1,\pi_2}(v) :=$

$$\begin{cases} v & \text{if } |\pi_1^{-1}(\pi_1(v))| \neq 1 \vee |\pi_2^{-1}(\pi_2(v))| \neq 1 \\ \pi_2^{-1}(\pi_1(v)) & \text{otherwise.} \end{cases}$$

Then, we may simply check whether φ_{π_1,π_2} is indeed an automorphism of G. Indeed, this check can be computed in time $\mathcal{O}(\Sigma_{v \in \text{supp}(\varphi_{\pi_1,\pi_2})} 1 + \deg(v))$.

SAUCY performs the check for local automorphisms during its depth-first search of its backtracking tree. It can then store the information about the automorphism and internally exploit its existence. For preprocessing purposes, however, we want to make the graph simpler or smaller.

Our probing strategy chooses a color class C of the graph and then concurrently performs two arbitrary root-to-leaf walks on the backtracking tree (*individualization-refinement* tree) that is also used by main solvers. Through the design of the backtracking procedure, each walk has a natural corresponding coloring (e.g., π_1 and π_2). We then continuously check whether the two walks already imply an automorphism (using φ_{π_1,π_2}). If, using this strategy, we find enough automorphisms to determine that C is equivalent to an orbit, we can individualize a vertex of C, thus simplifying the graph.

7 Exploiting the Quotient Graph

We now introduce another set of techniques which make use of the quotient graph $Q(G, \pi)$.

7.1 Edge Flip and Removal of Trivial Components

First, we describe how to efficiently *flip edges* between color classes. Let C_1, C_2 be two distinct color classes of π. Assume they are connected by m edges. The maximum number of edges between C_1 and C_2 is $|C_1||C_2|$. If $m > {}^{|C_1||C_2|}/_2$, we can flip every edge to a non-edge, and every non-edge to an edge, reducing the total number of edges in the graph. Since this operation is isomorphism-invariant and reversible, the automorphism group of the graph does not change.

When applying edge flips repeatedly and exhaustively, singleton vertices become vertices of degree 0. In fact, instead of performing edge flips in which singletons are involved, we can remove singletons directly without changing the automorphism group.

We want to remark that in the implementation, we use one canonical representation mapping to keep track of all removed vertices. This also includes removed singletons. Hence, we use string representations throughout all the techniques described in the paper. In addition to acting as a global canonical representation mapping, we also allow a renaming of vertices, which enables us to map all remaining vertices into the interval $\{1, 2, \ldots, n\}$, whenever n vertices remain.

7.2 Connected Components

A strategy more general than removing singletons is to exploit connected components of the quotient graph.

Consider the quotient graph $Q = Q(G, \pi)$ of a graph G with respect to a vertex coloring π. The (weakly) connected components of Q partition the vertex set of G into parts that are homogeneously connected. This allows us to treat components independently:

▶ **Lemma 1.** *If $D_1, \ldots, D_t$ are the connected components of the quotient graph $Q(G, \pi)$ then* $\mathrm{Aut}(G, \pi) = \Pi_{i=1}^{t} \mathrm{Aut}((G, \pi)[D_i])$.

By flipping edges between two color classes we can only ever shrink the components of $Q(G, \pi)$. It is therefore beneficial to first exhaustively flip edges and then consider connected components (see also [23]).

These types of components have previously been employed for isomorphism and automorphism testing [16, 19]. (In these contexts flips are not employed but rather edges in the quotient graph are characterized by non-homogeneous connections, which is equivalent.)

Regarding the implementation, we compute the connected components of the quotient graph without explicitly computing the quotient graph. We first perform edge flips for all fully connected color classes, i.e., whenever the number of edges between C_1, C_2 equals $|C_1||C_2|$. Then, we modify a basic algorithm for computing connected components as follows: usually, the algorithm determines for a vertex v its neighborhood $N(v)$ and adds this neighborhood to the connected component of v. Our modification simply also adds $\pi^{-1}\pi(v)$ in addition to $N(v)$ (i.e., it adds entire color classes). In turn, the algorithm gives us a partition of the vertices into the components of the quotient graph.

We use this to perform the probing strategy of Section 6 for each component of the quotient graph separately. We want to mention that after preprocessing is done, we could, theoretically, also use the components of the quotient graph to make independent calls to the main solver on the subgraphs induced by the components. These would, in turn, be smaller, and their handling could be parallelized. However, in our testing, after preprocessing is done, usually only one component is left, or there is one very large component and several smaller ones. We thus, at least so far, did not find it beneficial to use independent solver calls.

8 Scheduling of Techniques

We now describe when and how the preprocessor combines the techniques described in the previous sections.

The first step of the preprocessor is to apply color refinement to produce an equitable coloring. The coloring remains equitable throughout the entire algorithm, by reapplying color refinement whenever necessary (i.e., for the probing techniques). We also continuously remove singletons. Beyond this, our implementation allows the user to freely specify a schedule for the various techniques.

The schedule used to produce the benchmarks is as follows. We remove vertices of degree 0 and 1, and apply the heuristics described for vertices of degree 2. Next, we flip edges and apply probing for sparse automorphisms while making use of quotient graph components. Lastly, we repeat the schedule as long as the graph still contains vertices of degree 0 or 1 and the number of vertices of the graph shrunk by at least 25%. Note that this ensures that the schedule is only repeated at most a logarithmic number of times in the original graph size.

The implementation is called SASSY. It is implemented in C++ and uses the color refinement of DEJAVU (which is itself an amalgam of color refinement implementations in TRACES and SAUCY). The implementation is open source and freely available at [3].

9 Benchmarks

We split the benchmark section into three parts: first, we check whether applying the preprocessor speeds up state-of-the-art solvers on graph classes where the preprocessing techniques are supposedly effective. At the same time we check whether we introduced excessive overhead on graphs where the techniques are not effective. Secondly, we compare the performance of solver configurations using the preprocessor to state-of-the-art SAUCY and TRACES on a wide range of practical data sets. Thirdly, we analyze the separate impact of each of the different techniques used in the preprocessor (see Appendix D).

Whenever we apply the preprocessor followed by an execution of a main SOLVER, we write SY+SOLVER. The reported running time for a configuration SY+SOLVER is always the time used for preprocessing *and* solving the graph. All benchmarks were run on a machine featuring an Intel Core i7 9700K, 64GB of RAM on Ubuntu 20.04. We used NAUTY/TRACES 2.6, SAUCY 3.0, BLISS 0.73 and DEJAVU 1.2 (1% error bound and 4 threads for DEJAVU, all other tools are only able to run single-threaded). We ran all benchmarks 3 consecutive times in order to check whether running times are stable. We report the average and standard deviation.

Conventionally, the way to test symmetry detection solvers is to first randomly permute all given benchmark graphs [26, 7]. However, we feel that for many of the practical graphs, it is not clear whether this is the right way to test the tools: the initial order is often not arbitrary and may indeed encode information. For example in SAT, usually "literal" vertices and "clause" vertices are never mixed but appear as contiguous blocks of vertices. While this does not immediately help the symmetry detection process, aspects such as cache-efficiency might be affected. Therefore, we ran all benchmarks both ways: in the conventional manner of randomly permuting the instances (denoted with **(p)**), as well as using unaltered instances. Benchmarks for permuted graphs are in this section, while results for non-permuted graphs are in Appendix B. Overall, the results for both agree.

9.1 Preprocessed versus Unprocessed

We prepared two collections of graphs to test the impact of applying the preprocessor for each solver. **pract** contains practical graphs with a lot of exploitable structure for the preprocessor. On the other hand, the set **comb** contains combinatorial graphs where there is no or very little exploitable structure. In the following, we describe how we composed both sets.

Set "pract". The goal of this set is to measure whether preprocessing is worthwhile for a given solver on graphs where there is a lot of exploitable structure. Thus, this set contains practical graphs. Note that we test practical graphs much more thoroughly in the next

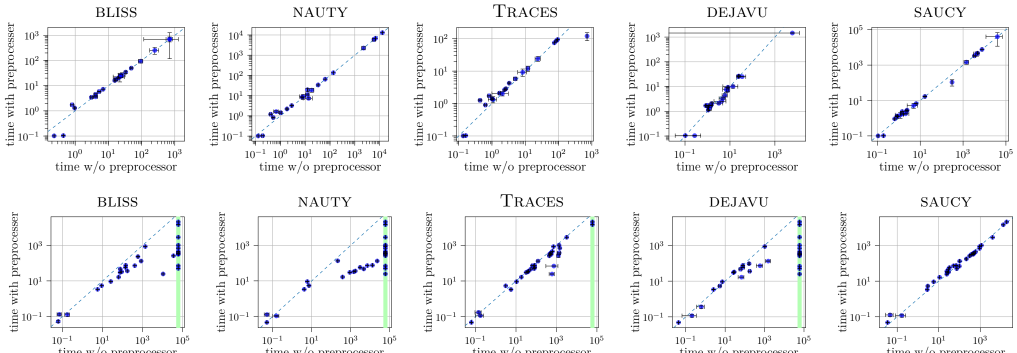


Figure 4 Solvers with SASSY vs. solvers without SASSY on **comb (p)** (top) and **pract (p)** (bottom). Timeout is 60s. The green bar shows instances that timed out without the preprocessor.

	state-of-the-art		this paper				
set	SAUCY	TRACES	SY+DEJAVU	SY+TRACES	SY+BLISS	SY+SAUCY	SY+NAUTY
dac (p)	0.51 ± 0.057	2.49 ± 0.072	$\mathbf{0.38} \pm 0.006$	0.91 ± 0.016	0.5 ± 0.002	0.41 ± 0.058	0.46 ± 0.005
states (p)	7.32 ± 0.031	12.55 ± 0.281	$\mathbf{6.79} \pm 0.075$	6.8 ± 0.069	6.8 ± 0.062	6.83 ± 0.085	6.8 ± 0.074
internet (p)	0.19 ± 0.005	2.47 ± 0.379	$\mathbf{0.14} \pm 0.003$	0.15 ± 0.004	$\mathbf{0.14} \pm 0.002$	$\mathbf{0.14} \pm 0.004$	$\mathbf{0.14} \pm 0.003$
ispd (p)	7.12 ± 0.069	7.04 ± 0.085	5.48 ± 0.046	$\mathbf{5.43} \pm 0.023$	5.46 ± 0.005	5.45 ± 0.025	5.45 ± 0.008
MIP2017 (p)	22.3 ± 0.22	803.63 ± 10.469	$\mathbf{14.07} \pm 0.171$	92.76 ± 2.796	28.59 ± 0.234	15.52 ± 0.324	26.54 ± 0.128
SAT2021 (p)	2217.61 ± 0.627	3645.33 ± 11.963	$\mathbf{1701.62} \pm 10.411$	1856.39 ± 10.138	1939.54 ± 4.926	1786.68 ± 6.005	1763.12 ± 6.214
SAT2021-up (p)	1886.87 ± 4.472	2948.5 ± 25.254	$\mathbf{1439.5} \pm 2.972$	1538.71 ± 8.639	1650.68 ± 3.577	1508.91 ± 2.327	1481.28 ± 3.556

Figure 5 Benchmark results on various sets of large, practical graphs (**randomly permuted**), timeout is 60s. The benchmarks compare solver configurations using the preprocessor ("SY+") to state of the art SAUCY and TRACES. Shown values are the sum over all instances in the set in seconds. The average and standard deviation of 3 consecutive runs is used. Bold entries indicates the fastest running time for the given set.

section. To make up **pract**, we picked the 5 largest instances (if available) of all the SAUCY benchmark sets, and for the sets arising from computational tasks (MIP and SAT) we picked 5 instances uniformly at random.

Set "comb". The goal of this set is to measure the overhead of applying the preprocessor on graphs where there is no or very little exploitable structure (i.e., where the preprocessor is expected to have no effect). For this purpose, we chose a large variety of graphs from combinatorics, on which solvers are routinely evaluated [26]. The subset we chose contains a graph from almost every graph class of the benchmark library from [2] (cfi, grid, grid-sw, had, had-sw, hypercubes, kef, latin, latin-sw, lattice, mz, paley, pp, ran10, ransq, sts, sts-sw, ranreg, tran, triang and shrunken multipedes). Whenever applicable, we chose a graph of around 1000 vertices: note that here, we apply a size restriction, since combinatorial graphs are generally difficult for their size. We choose an even smaller graph or left out sets entirely whenever a solver had trouble finishing the instance quickly. Note that, since we want to measure the preprocessing overhead, only instances for which the solvers finish in a reasonable amount of time are of interest. If solvers take a long time solving an instance to begin with, the overhead of the preprocessor is always negligible. Note that these restrictions **only apply to comb**: all the other sets tested in this paper have no restriction on the size of instances and instances were not chosen manually.

Results. The results are summarized in Figure 4. We conclude for BLISS, NAUTY and DEJAVU that the preprocessor increases performance dramatically on most instances, while the overhead of the preprocessor is negligible. For TRACES, performance also improves, in particular there are fewer timeouts. However, the improvement is not as dramatic.

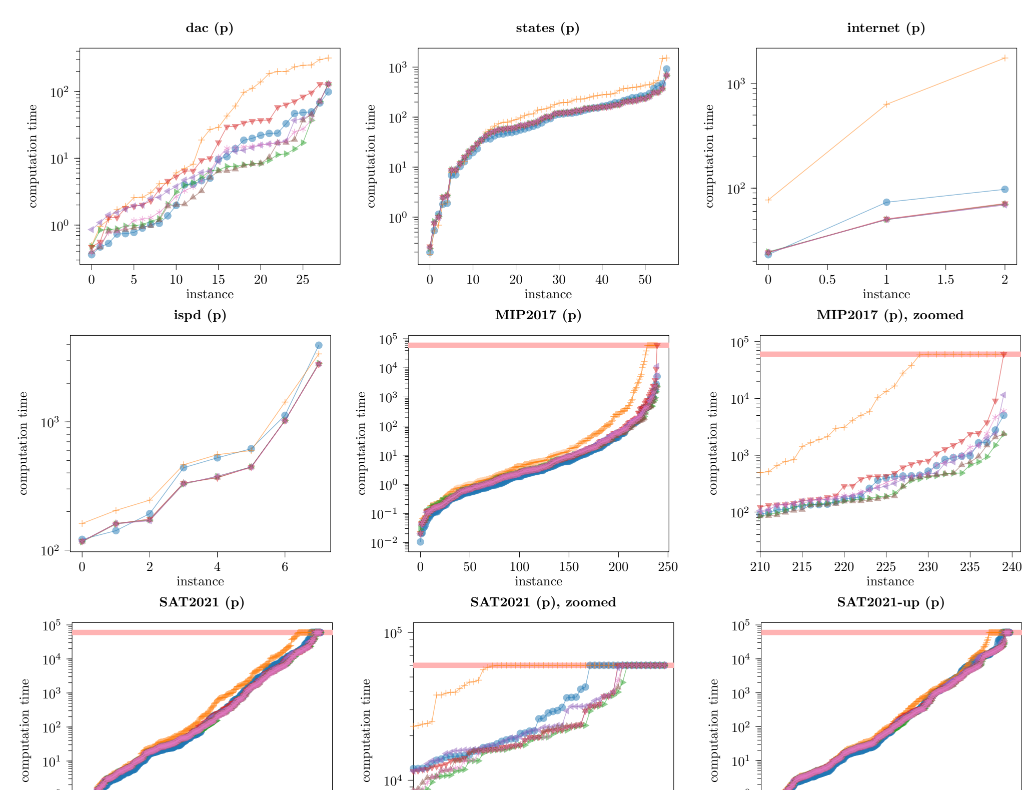


Figure 6 Detailed plots for the various sets of Figure 5. The red bar illustrates timeouts. Instances are sorted according to running time.

There are however two eye-catching instances: first, there is an instance with very high standard deviation for DEJAVU. The instance is a Kronecker eye flip graph, which DEJAVU is known to struggle with [6]. Secondly, there is a particular expensive outlier for TRACES. We analyze and discuss the instance in detail in Appendix C. There, we conclude that the outlier is caused through an undesired interaction with a heuristic of TRACES.

9.2 Comparison to state-of-the-art

The state-of-the-art solver on large practical graphs is SAUCY. Furthermore, TRACES also contains low-degree techniques. Thus, we compare all the solvers with the preprocessor to SAUCY and TRACES. The timeout used is $60s$ (also if a solver runs out of memory).

We test all sets of the SAUCY distribution. We also test 3 contemporary sets of practical graphs: the MIP2017 set contains graphs stemming from the mixed integer programming library (see [1]). The SAT2021 library contains graphs stemming from SAT instances from the SAT competition 2021 [4]. In the SAT2021-up set, SAT instances were first preprocessed using the unit and pure literal rule (see [5]). We want to remark that the SAT sets contain the largest graphs out of all the tested sets, with up to tens of millions of vertices.

The results are summarized in Figure 5, Figure 6, and Appendix B. We observe that the previous state-of-the-art (saucy) is outperformed on all but one set by several solvers using the preprocessor. This demonstrates that the approach of using our universal preprocessor in conjunction with different solvers can outperform state-of-the-art. Moreover, both saucy and Traces also visibly speed up by applying the preprocessor on all but one set.

On a few of the very large graphs in the SAT sets, dejavu and Traces run out of memory. Hence, depending on how this is weighed into the evaluation, other solvers may be preferable. In all cases where dejavu runs out of memory, all other solvers time out. In any case, on all these sets, sy+nauty and sy+saucy also outperform saucy.

10 Conclusion and Future Development

We introduced the new sassy preprocessor for symmetry detection. We demonstrated that sassy indeed speeds up state-of-the-art solvers on large, practical graphs. Future additions to the preprocessor could include more heuristics for degree 2 removal, stronger invariants or even more efficient implementations and tuning of the existing heuristics. Since we have observed a high sensitivity of state-of-the-art solvers to their choice of cell selectors, a more extensive study into the topic would be of interest.

References

1 MIPLIB 2017 - The Mixed Integer Programming Library. `https://https://miplib.zib.de/`.

2 nauty and Traces. `http://pallini.di.uniroma1.it`.

3 sassy. `https://github.com/markusa4/sassy`.

4 SAT Competition 2021. `https://satcompetition.github.io/2021/`.

5 Markus Anders. SAT preprocessors and symmetry. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 1:1–1:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.SAT.2022.1`.

6 Markus Anders and Pascal Schweitzer. Engineering a fast probabilistic isomorphism test. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 73–84. SIAM, 2021. `doi:10.1137/1.9781611976472.6`.

7 Markus Anders and Pascal Schweitzer. Parallel computation of combinatorial symmetries. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 6:1–6:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ESA.2021.6`.

8 Markus Anders, Pascal Schweitzer, and Florian Wetzels. Comparative design-choice analysis of color refinement algorithms beyond the worst case. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 15:1–15:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ICALP.2021.15`.

9 Christoph Berkholz, Paul S. Bonsma, and Martin Grohe. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theory Comput. Syst.*, 60(4):581–614, 2017. `doi:10.1007/s00224-016-9686-0`.

10 Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In Nikolaj S. Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23 – 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 – August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2011. `doi:10.1007/978-3-642-22438-6_10`.

11 Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, pages 530–534. ACM, 2004. doi:10.1145/996566.996712.

12 David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In Nikolaj S. Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23 – 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 – August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2011. doi:10.1007/978-3-642-22438-6_18.

13 Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for SAT. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017 – 20th International Conference, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 83–100. Springer, 2017. doi:10.1007/978-3-319-66263-3_6.

14 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005. doi:10.1007/11499107_5.

15 Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier, 2006. doi:10.1016/S1574-6526(06)80014-3.

16 Mark K. Goldberg. A nonfactorial algorithm for testing isomorphism of two graphs. *Discret. Appl. Math.*, 6(3):229–236, 1983. doi:10.1016/0166-218X(83)90078-1.

17 Christopher Hojny and Marc E. Pfetsch. Symmetry handling via symmetry breaking polytopes. In Ekrem Duman and Ali Fuat Alkaya, editors, *13th Cologne Twente Workshop on Graphs and Combinatorial Optimization, Istanbul, Turkey, May 26-28, 2015*, pages 63–66, 2015.

18 Derek F. Holt, Bettina Eick, and Eamonn A. O'Brien. *Handbook of Computational Group Theory*. Discrete Mathematics and Its Applications. Chapman and Hall/CRC, 2005.

19 Tommi A. Junttila and Petteri Kaski. Conflict propagation and component recursion for canonical labeling. In Alberto Marchetti-Spaccamela and Michael Segal, editors, *Theory and Practice of Algorithms in (Computer) Systems – First International ICST Conference, TAPAS 2011, Rome, Italy, April 18-20, 2011. Proceedings*, volume 6595 of *Lecture Notes in Computer Science*, pages 151–162. Springer, 2011. doi:10.1007/978-3-642-19754-3_16.

20 Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Symmetry and satisfiability: An update. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2010. doi:10.1007/978-3-642-14186-7_11.

21 Manuel Kauers and Martina Seidl. Symmetries of quantified boolean formulas. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018 – 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 2018. doi:10.1007/978-3-319-94144-8_13.

22 Sandra Kiefer, Ilia Ponomarenko, and Pascal Schweitzer. The weisfeiler-leman dimension of planar graphs is at most 3. *J. ACM*, 66(6):44:1–44:31, 2019. doi:10.1145/3333003.

23 Sandra Kiefer, Pascal Schweitzer, and Erkal Selman. Graphs identified by logics with counting. *ACM Trans. Comput. Log.*, 23(1):1:1–1:31, 2022. doi:10.1145/3417515.

24 Tuukka Korhonen, Jeremias Berg, Paul Saikko, and Matti Järvisalo. Maxpre: An extended maxsat preprocessor. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017 – 20th International Conference, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 449–456. Springer, 2017. doi:10.1007/978-3-319-66263-3_28.

25 François Margot. Symmetry in integer linear programming. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 – From the Early Years to the State-of-the-Art*, pages 647–686. Springer, 2010. doi:10.1007/978-3-540-68279-0_17.

26 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014. doi:10.1016/j.jsc.2013.09.003.

27 Marc E. Pfetsch and Thomas Rehn. A computational comparison of symmetry handling methods for mixed integer programs. *Math. Program. Comput.*, 11(1):37–93, 2019. doi:10.1007/s12532-018-0140-y.

28 Adolfo Piperno. Isomorphism test for digraphs with weighted edges. 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 30:1–30:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.SEA.2018.30.

29 Pascal Schweitzer and Daniel Wiebking. A unifying method for the design of algorithms canonizing combinatorial objects. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 1247–1258. ACM, 2019. doi:10.1145/3313276.3316338.

30 Ákos Seress. *Permutation Group Algorithms*. Cambridge Tracts in Mathematics. Cambridge University Press, 2003. doi:10.1017/CBO9780511546549.

A Probing for Sparse Automorphisms

A.1 The Individualization Refinement Framework

The individualization-refinement framework is a general framework for algorithms computing isomorphisms, automorphisms and canonical labellings (see [26]). These algorithms generally work on a special tree, the so-called *IR tree*. We give a brief description of how IR trees are constructed. For a more extensive description, in particular for the numerous strategies needed to perform the search more efficiently, see [26, 7].

Each node x of the tree has a corresponding equitable coloring π_x of the input graph. The leaves correspond to discrete colorings. The most important property of IR trees is that they are isomorphism-invariant, meaning that on G and $\varphi(G)$ (where φ is an isomorphism) we obtain isomorphic IR trees.

Let (G, π) be the input graph. Let π' be the coarsest equitable refinement of π. We let the root of the IR tree correspond to π'. In each node x of an IR tree, a non-trivial color class from the corresponding coloring π_x is chosen (i.e., a $C = \pi_x^{-1}(c)$ with $|C| > 1$, C must be chosen isomorphism-invariantly). If there is no non-trivial color class, then x is a leaf and its corresponding coloring is discrete. Otherwise, for each $v \in C$, we define x_v as a child of x in the IR tree. Let π_{x_v} denote the coloring corresponding to x_v. We may obtain π_{x_v} from π_x as follows. Starting from π_x, we first artificially single out v (i.e., *individualize* v). This means we set $\pi_{x_v}(v) := c'$ where $c' \notin \pi(V(G))$ (again, c' is chosen isomorphism-invariantly). Then, we refine the coloring using color refinement, obtaining the equitable coloring π_{x_v}.

We can derive automorphisms from IR trees. If π_1, π_2 are leaves of the tree, i.e., discrete colorings, then $\varphi := \pi_1^{-1} \circ \pi_2$ defines a permutation on $V(G)$. While φ is not guaranteed to be an automorphism, we can efficiently test whether it is (by checking whether $\varphi(G) = G$).

With this method all of $\mathrm{Aut}(G)$ can be computed. This follows essentially from the fact that comparing all pairs of leaves in this way will give us all automorphisms of G (or rather a generating set of $\mathrm{Aut}(G)$ when automorphism pruning is applied; see [26]).

A.2 The Probing Algorithm

Our probing strategy only searches for automorphisms which can be used directly to reduce the graph. The idea is as follows. For a color class that we want to reduce, we attempt to collect automorphisms that transitively permute all the vertices in the entire color class. This certifies that the color class is an orbit. We can then individualize an arbitrary vertex of the color class. In contrast, if we only have some automorphisms that together do not act transitively on the color class, it is not clear how to manipulate the graph favorably. In particular, since some of the vertices may not be in the same orbit, we do not know which vertex to individualize. We now describe the bounded IR probing algorithm.

(Description of Algorithm 1.) (See Algorithm 1 for the pseudocode.) The algorithm expects as input a colored graph $G = (V, E, \pi)$, a color class $C_{probe} = \pi^{-1}(c)$ as well as a length bound L. It outputs a set of automorphisms Φ and a coloring π' refining π. If the probing was unsuccessful then $\Phi = \{\}$ and $\pi' = \pi$. Otherwise $\langle\Phi\rangle$ acts transitively on C_{probe} and π' is obtained from π by individualizing a vertex and refining.

We compute arbitrary IR paths (i.e., a rooted path in the IR tree) starting with an individualization of a vertex in C_{probe}. The path is only computed up to a length of L.

Initially, the algorithm examines two of these paths concurrently, starting in two different vertices $v_1, v_2 \in C_{probe}$. It checks after each individualization whether the automorphism φ_{π_1,π_2} (defined, as above, mapping corresponding singletons) is an automorphism. If this happens to be the case after having performed, say, L' individualizations, we bound all subsequent paths by L'.

Afterwards for each vertex $w \in C_{probe} \setminus \{v_1, v_2\}$ we compute an IR path starting with the individualization of w. We hope to find an automorphism mapping v_1 to w. If we discover an automorphism for each w, we return the set of automorphisms Φ, individualize v_1 in π, refine to obtain π' and return Φ and π'.

(Correctness of Algorithm 1.) Correctness of the algorithm follows simply from the fact that we certify all automorphisms. That is, every map claimed to be an automorphism is indeed an automorphism. Since this certification is done for each automorphism, this certifies the fact that C_{probe} is an orbit of $\mathrm{Aut}(G, \pi)$. Since we return all automorphisms required to construct the orbit (i.e., we return Φ), we have $\langle\Phi \cup \mathrm{Aut}(G, \pi')\rangle = \mathrm{Aut}(G, \pi)$ by the orbit-stabilizer theorem (see [18]).

(Implementation of Algorithm 1.) We want to make some further remarks on the implementation of the algorithm. In fact, even though it can be implemented very efficiently, it generally has to be used sparingly. Overall we need to decide when and how often to employ the probing strategy and also which depth bound L to use. The preprocessor essentially uses three strategies: 1-IR probing, ∞-IR probing with class size 2 and ∞-IR probing up to class size 8 (in order of descending frequency).

B Non-permuted Benchmarks

Figure 7, Figure 8 and Figure 9 show benchmark results for *non-permuted* graphs. While overall times are faster across all graph classes and solvers than on the randomly permuted graphs, the interpretation of results given in Section 9 also applies to these benchmarks. Hence, our results agree on both randomly permuted and non-permuted graphs.

Algorithm 1 Bounded IR probing in a color class C_{probe} up to a path of length L.

```
1  function BoundedProbeIR(G, π, C_probe, L)
      Input : graph G = (V, E, π) where π is equitable, color class C_probe of π, length
              bound L
      Output: (equitable) coloring π′, set of automorphisms Φ
2     Φ := {} ;                                          // set of automorphisms
3     Pick vertices v_1, v_2 ∈ π^{-1}(C_probe);
4     for i ∈ {1, 2} do
5        π_i := π;
6        individualize v_i in π_i;
7        ColorRefinement(G, π_i);
8     L_C := [C_probe] ;                                 // list of color classes
9     while |L_C| < L do
10       if φ_{π_1,π_2}(G, π) = (G, π) then
11          break;                                       // automorphism found
12       C := non-trivial color class of π_1;
13       L_C += [C] ;                                    // append C to L_C
14       individualize some v ∈ π_1^{-1}(C) in π_1;
15       ColorRefinement(G, π_1);
16       individualize some v ∈ π_2^{-1}(C) in π_2;
17       ColorRefinement(G, π_2);
18    if φ_{π_1,π_2}(G, π) ≠ (G, π) then
19       return π, ∅;                                    // probing failed
20    else
21       Φ := Φ ∪ {φ};
22    for w ∈ C_probe ∖ {v_1, v_2} do
23       reset π_2 to π ;                                // essentially π_2 := π
24       individualize w in π_2;
25       ColorRefinement(G, π_2);
26       for C ∈ L_C do
27          individualize some v ∈ π_2^{-1}(C) in π_2;
28          ColorRefinement(G, π_2);
29       if φ_{π_1,π_2}(G, π) ≠ (G, π) then
30          return π, ∅;                                 // probing failed
31       else
32          Φ := Φ ∪ {φ};
33    individualize v_1 in π ;           // success; individualize v_1 in (G, π)
34    return ColorRefinement(G, π), Φ;
```

C The Outlier in Combinatorial Graphs

There is one particular outlier in the evaluation of Traces comparing preprocessed vs. unprocessed instances. The instance is a shrunken multipede on 408 vertices. Without preprocessing, it is solved in $0.75s$, while with preprocessing it is solved in $6.3s$. This is at first glance confusing: the preprocessor finishes within less than 0.5ms, while not changing the graph other than coloring it with its coarsest equitable coloring. This is however almost the same coloring Traces would also compute for the graph.

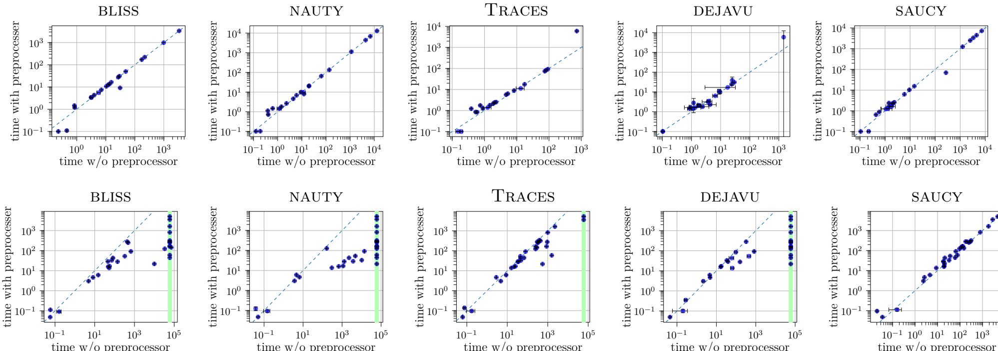


Figure 7 Solvers with SASSY vs. solvers without SASSY on **comb** (top) and **pract** (bottom), not permuted. Timeout is $60s$. The green bar shows instances that timed out without the preprocessor.

	state-of-the-art		this paper				
set	SAUCY	TRACES	SY+DEJAVU	SY+TRACES	SY+BLISS	SY+SAUCY	SY+NAUTY
dac	0.35 ± 0.008	2.47 ± 0.005	0.28 ± 0.001	0.81 ± 0.002	0.37 ± 0.002	$\mathbf{0.27} \pm 0.002$	0.34 ± 0.001
states	$\mathbf{2.89} \pm 0.054$	7.58 ± 0.158	3.85 ± 0.048	3.85 ± 0.04	3.85 ± 0.041	3.85 ± 0.038	3.85 ± 0.043
internet	0.15 ± 0.002	2.23 ± 0.022	$\mathbf{0.13} \pm 0.000$	$\mathbf{0.13} \pm 0.001$	$\mathbf{0.13} \pm 0.001$	$\mathbf{0.13} \pm 0.000$	$\mathbf{0.13} \pm 0.000$
ispd	3.83 ± 0.028	4.84 ± 0.059	3.7 ± 0.057	3.7 ± 0.061	3.7 ± 0.057	3.7 ± 0.054	$\mathbf{3.68} \pm 0.039$
MIP2017	10.96 ± 0.158	774.09 ± 0.578	$\mathbf{9.12} \pm 0.165$	84.42 ± 0.201	21.46 ± 0.331	10.66 ± 0.109	21.04 ± 0.163
SAT2021	1292.76 ± 1.641	2855.57 ± 10.636	$\mathbf{881.69} \pm 0.982$	1058.97 ± 3.283	1149.04 ± 3.73	990.96 ± 3.323	988.15 ± 2.662
SAT2021-up	1144.06 ± 3.648	2393.85 ± 4.799	$\mathbf{780.02} \pm 6.009$	903.93 ± 2.517	1027.61 ± 3.815	876.77 ± 2.535	865.38 ± 2.578

Figure 8 Benchmark results on various sets of large, practical graphs (**not permuted**), timeout is $60s$. Running out of memory also counts as a timeout. The benchmarks compare solver configurations using the preprocessor ("SY+") to state of the art SAUCY and TRACES. Shown values are the sum over all instances in the set in seconds. The average and standard deviation of 3 consecutive runs is used. Bold entries indicates the fastest running time for the given set.

The only difference is that TRACES might name the colors differently internally, e.g., color 3 might be named color 6 instead. While this does not structurally make the graph harder or easier, heuristics internally might always, for example, choose the "first largest color" (this is similar to, e.g., variable ordering in SAT solvers). Thus, renaming the colors might influence the decisions made by the solver. Using the "first" color is however usually not a deliberate decision. In fact, if we simply reverse the order of colors, the graph is indeed solved in $0.12s$. In [7], it is also argued that cell selector choice has a significant impact on the set of shrunken multipedes. We believe that the solution to this issue is to make structurally better choices, and has indeed little to do with the role of the preprocessor.

D Ablation study

In Figure 10 we evaluate for DEJAVU on the MIP2017 set the effect of each of the preprocessing techniques separately. We do so by running the configuration SY+DEJAVU, but performing a separate run for each technique, deactivating the respective technique. For example, SY+DEJAVU-DEG2 runs SY+DEJAVU without the degree 2 removal techniques. The data shows that each of the techniques has a beneficial impact on running time. By far the most impactful technique is the removal of degree 0 and 1, followed by the removal of vertices of degree 2, and lastly probing.

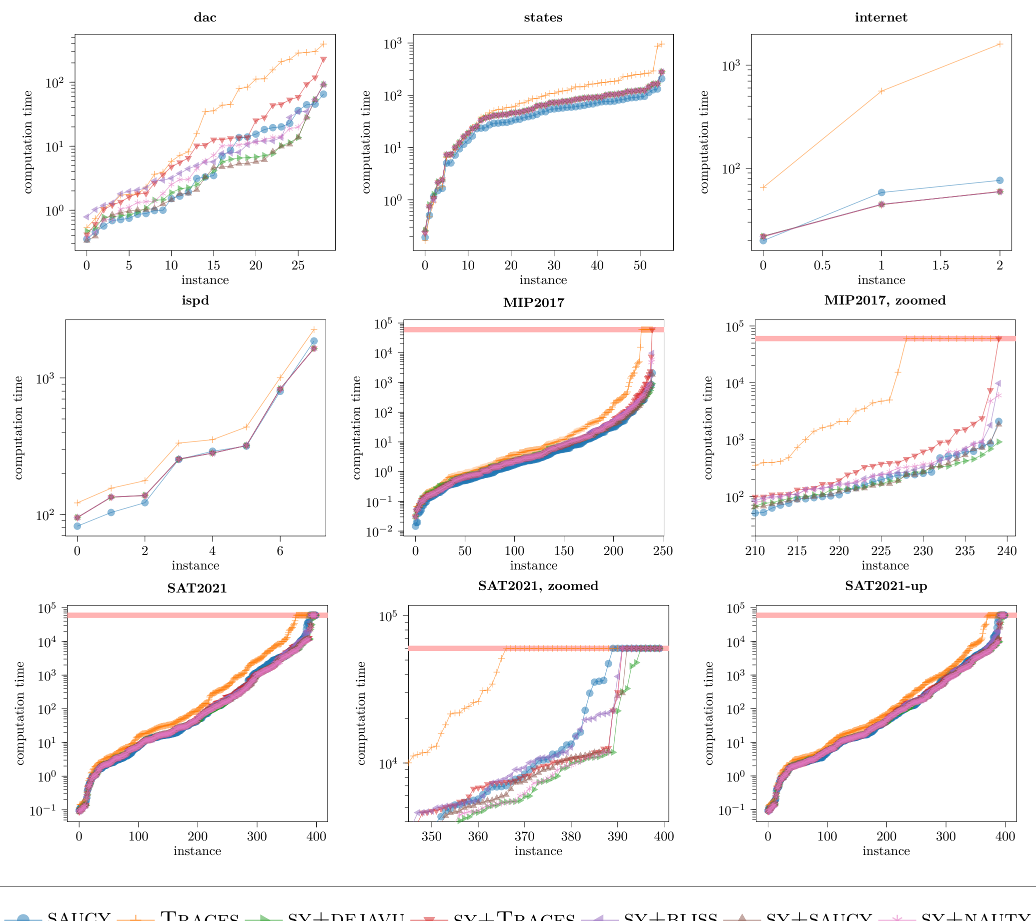


Figure 9 Detailed plots for the various sets of Figure 8. The red bar illustrates timeouts. Instances are sorted according to running time.

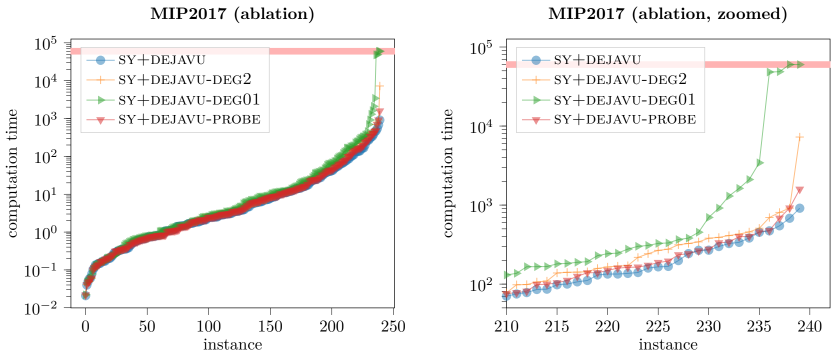


Figure 10 Ablation study for SY+DEJAVU on the MIP2017 graphs (times: 9.15s, 17.87s, 234.47s, 10.65s), timeout is 60s.

Fast Reachability Using DAG Decomposition

Giorgos Kritikakis
Univeristy of Crete, Heraklion, Greece

Ioannis G. Tollis
Univeristy of Crete, Heraklion, Greece

Abstract

We present a fast and practical algorithm to compute the transitive closure (TC) of a directed graph. It is based on computing a reachability indexing scheme of a *directed acyclic graph* (DAG), $G = (V, E)$. Given any path/chain decomposition of G we show how to compute in parameterized linear time such a reachability scheme that can answer reachability queries in constant time. The experimental results reveal that our method is significantly faster in practice than the theoretical bounds imply, indicating that path/chain decomposition algorithms can be applied to obtain fast and practical solutions to the transitive closure (TC) problem. Furthermore, we show that the number of non-transitive edges of a DAG G is $\leq width * |V|$ and that we can find a substantially large subset of the transitive edges of G in linear time using a path/chain decomposition. Our extensive experimental results show the interplay between these concepts in various models of DAGs.

2012 ACM Subject Classification Theory of computation → Theory and algorithms for application domains; Theory of computation → Design and analysis of algorithms

Keywords and phrases graph algorithms, hierarchy, directed acyclic graphs (DAG), path/chain decomposition, transitive closure, transitive reduction, reachability, reachability indexing scheme

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.2

Related Version *Full Version*: https://arxiv.org/abs/2212.03945

Supplementary Material *Software (Source Code)*: https://github.com/GiorgosKritikakis/OnGraphHierarchies

1 Introduction

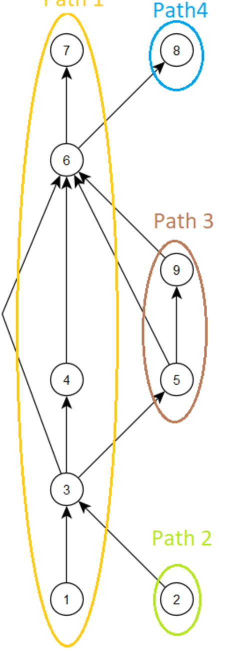
The problem of computing reachability information or a transitive closure of a directed graph is fundamental in computer science and has a wealth of applications. Formally, given a directed graph $G = (V, E)$, the transitive closure of G, denoted as G^*, is a graph (V, E^*) such that E^* contains all edges in E, and for any pair of vertices u, $v \in V$, if there exists a directed path from u to v in G, then there is a directed edge from u to v in E^*. An edge (v_1, v_2) of a DAG G is transitive if there is a path longer than one edge that connects v_1 and v_2. Given a directed graph with cycles, we can find the strongly connected components (SCC) in linear time and collapse all vertices of a SCC into a supernode. Hence, any reachability query can be reduced to a query in the resulting *Directed Acyclic Graph* (DAG). Additionally, DAGs are very important in many applications in several areas of research and business because they often represent hierarchical relationships between objects in a structure. Any DAG can be decomposed into vertex disjoint *paths* or *chains*. In a path every vertex is connected to its successor by an edge, while in a chain any vertex is connected to its successor by a directed path, which may be an edge. A *path/chain decomposition* is a set of vertex disjoint paths/chains that cover all the vertices of a DAG.

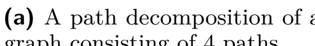
The *width* of a DAG $G = (V, E)$ is the maximum number of mutually unreachable vertices of G [8]. An *optimum chain decomposition* of a DAG G contains the minimum number of chains, k, which is equal to the width of G. In Section 2 we present experimental results that show the behavior of the width of DAGs as they become larger and/or denser. Due to

© Giorgos Kritikakis and Ioannis G. Tollis;
licensed under Creative Commons License CC-BY 4.0
21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis; Article No. 2; pp. 2:1–2:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

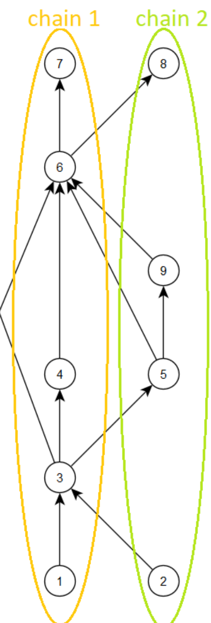
the multitude of applications there are several algorithms to find a chain decomposition of a DAG, see for example [16, 9, 7, 22, 4, 5, 18, 26]. Some of them find the optimum and some are heuristics. Generally speaking the algorithms that compute the optimum take more than linear time and use flow techniques which are often heavy and complicated to implement. On the other hand, for several practical applications it is not necessary to compute an optimum chain decomposition.

We consider reachability mainly for the static case, i.e., when the graph does not change. The question of whether an arbitrary vertex v can reach another arbitrary vertex u can be answered in linear time by running a breadth-first or depth-first search from v, or it can be answered in constant time after a reachability indexing scheme, or transitive closure of the graph has been computed. The transitive closure of a graph can be computed in $O(nm)$ time by starting a breadth-first or depth-first search from each vertex. Alternatively, one can use the Floyd-Warshall algorithm [12] which runs in $O(n^3)$, or solutions based on matrix multiplication [24]. Currently, the best known bound on the asymptotic complexity of a matrix multiplication algorithm $O(n^{2.3728596})$ time [2]. An algorithm with complexity $O(n^{2.37188})$ was very recently announced in a preprint [10]. However, this and similar improvements to Strassen's Algorithm are not used in practice because the constant coefficient hidden by the notation are extremely large. Here we focus on computing a reachability indexing scheme in almost linear time. Notice that we do not explicitly compute the transitive closure matrix of a DAG. The matrix can be easily computed from the reachability indexing scheme in $O(n^2)$ time (constant time per entry).





(a) A path decomposition of a graph consisting of 4 paths.



(b) A chain decomposition of a graph consisting of 2 chains.

Figure 1 Path and chain decomposition of an example graph.

In this paper we present a practical algorithm to compute a reachability indexing scheme (or the transitive closure information) of a DAG $G = (V, E)$, utilizing a given path/chain decomposition (i.e., the DAG and a path/chain decomposition are given as input to the

algorithm). The scheme can be computed in parameterized linear time, where the parameter is the number, k_c, of paths/chains in the given decomposition. The scheme can answer any reachability query in constant time. Let E_{tr}, $E_{tr} \subset E$, denote the set of transitive edges and E_{red}, $E_{red} = E - E_{tr}$, denote the set of non-transitive edges of G. We show that $|E_{red}| \leq width * |V|$ and that we can compute a substantially large subset of E_{tr} in linear time (see Section 3). This implies that any DAG can be reduced to a smaller DAG that has the same TC in linear time. Consequently, several hybrid reachability algorithms will run much faster in practice. The time complexity to produce the scheme is $O(|E_{tr}| + k_c * |E_{red}|)$, and its space complexity is $O(k_c * |V|)$ (see Section 4). Our experimental results reveal the practical efficiency of this approach. In fact, the results show that our method is substantially better in practice than the theoretical bounds imply, indicating that path/chain decomposition algorithms can be used to solve the transitive closure (TC) problem. Clearly, given the reachability indexing scheme the TC matrix can be computed in $O(|V|^2)$ time.

2 Width of a DAG and Decomposition into Paths/Chains

In this section, we briefly describe some categories of path and chain decomposition techniques and show experimental results for the width in different graph models. We focus on fast and practical path/chain decomposition heuristics. There are two categories of path decomposition algorithms, Node Order Heuristic, and Chain Order Heuristic, see [16]. The first constructs the paths one by one, while the second creates the paths in parallel. The chain-order heuristic starts from a vertex and extends the path to the extent possible. The path ends when no more unused immediate successors can be found. The node-order heuristic examines each vertex (node) and assigns it to an existing path. If no such path exists, then a new path is created for the vertex. In addition to path-decomposition algorithm categorization, Jagadish in [16] describes chain decomposition heuristics. Those heuristics run in $O(n^2)$ time using a pre-computed transitive closure, which is not linear, and we will not discuss them further.

In [19], a chain decomposition technique was introduced that runs in $O(|E| + c * l)$ time, where c is the number of path concatenations, and l is the length of a longest path of the DAG. This approach relies on path concatenation. We can concatenate two paths/chains into a single chain if there is a path between the last vertex of one chain and the first vertex of another chain. This algorithm produces decompositions that are very close to the optimum, and its worst-case time complexity is the same as the algorithms that construct simple path decomposition. The above techniques have been tested in practice, and we can utilize any of these approaches to build a chain decomposition in linear or almost linear time, see [19]. In the next sections, we describe how fast chain decomposition algorithms can enhance transitive closure solutions, and present in detail an indexing scheme.

In the rest of this section, we present results that reveal the behavior of the width as the graph density increases. We use three different random graph models implemented in networkx : Erdős-Rényi [11], Barabasi-Albert [3], and Watts-Strogatz [28] models. The generated graphs are made acyclic, by orienting all edges from low to high ID number, see the documentation of networkx [14] for more information about the generators. For every model, we created 12 types of graphs: Six types of 5000 nodes and six types of 10000 nodes, both with average degrees 5, 10, 20, 40, 80, and 160. We used different average degrees in order to have results for various sizes and densities. All experiments were conducted on a simple laptop PC (Intel(R) Core(TM) i5-6200U CPU, with 8 GB of main memory). Our algorithms have been developed as stand-alone java programs and were run on multiple copies of graphs. We observed that the graphs generated by the same generator with the same parameters

Table 1 The width of the graph in three different networkx models as the density increases for graphs of 5000 nodes.

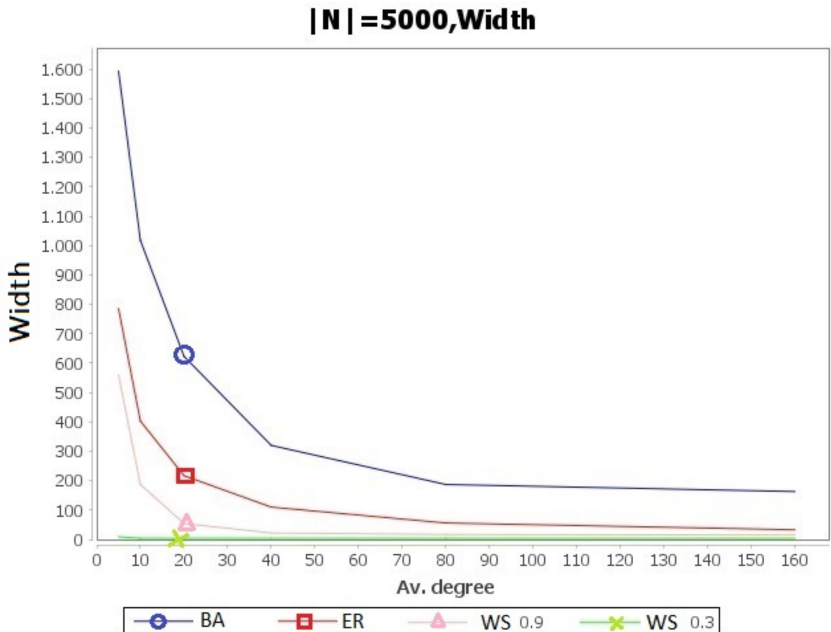
$\lvert V \rvert = 5000$						
Av. Degree	5	10	20	40	80	160
	BA					
Width	1593	1018	623	320	187	163
	ER					
Width	785	403	217	110	56	33
	WS, b=0.9					
Width	560	187	54	22	17	15
	WS, b=0.3					
Width	9	4	4	4	4	4

have small width deviation. For example, the percentage of deviation on ER is about 5% and for the BA model is less than 10%. The width deviation of the graphs in the WS model is a bit higher, but this is expected since the width of these graphs is significantly smaller. The aim of our experiments is to understand the behavior of the width of DAGs created in different models. Tables 1 and 2 show the width (computed by Fulkerson's method) for graphs of 5000 nodes and 10000 nodes, respectively.

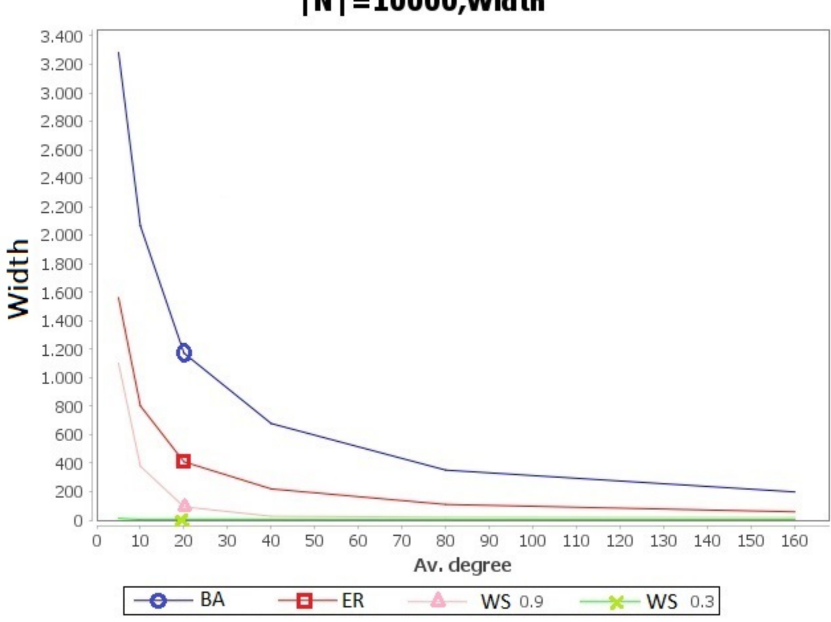
Random Graph Generators.

- **Erdős-Rényi (ER) model** [11]: The generator returns a random graph $G_{n,p}$, where n is the number of nodes and every edge is formed with probability p.
- **Barabási–Albert (BA) model** [3]: preferential attachment model: A graph of n nodes is grown by attaching new nodes each with m edges that are preferentially attached to existing nodes with high degree. The factors n and m are parameters to the generator.
- **Watts–Strogatz (WS) model** [28]: small-world graphs: First it creates a ring over n nodes. Then each node in the ring is joined to its k nearest neighbors. Then shortcuts are created by replacing some edges as follows: for each edge (u, v) in the underlying "n-ring with k nearest neighbors" with probability b replace it with a new edge (u, w) with uniformly random choice of an existing node w. The factors n, k, b are the parameters of the generator.

Understanding the width in DAGS. In order to understand the behavior of the width of DAGs of these random graph models we observe: (i) the BA model produces graphs with a larger width than ER, and (ii) the ER model creates graphs with a larger width than WS. For the WS model, we created two sets of graphs: The first has probability $b = 0.9$ and the second has $b = 0.3$. Clearly, if the probability b of rewiring an edge is 0, the width would be one, since the generator initially creates a path that goes through all vertices. As the rewiring probability b grows, the width grows. That is the reason we choose a low and a high probability. Figures 2a and 2b, are derived from Tables 1 and 2, and demonstrate the behavior of the width for each model on the graphs of 5000 and 10000 nodes. Please notice that in almost all model graphs (except for WS with $b = 0.3$) the width of a DAG decreases fast as the density of the DAG increases. As a matter of fact, it is interesting to observe that the width of the ER model graphs is proportional to $\frac{\text{Number of nodes}}{\text{average degree}}$. The width of the BA model graphs is clearly higher, but it follows a similar trend.



(a) The width curve on graphs of 5000 nodes.



(b) The width curve on graphs of 10000 nodes.

Figure 2 The width curve on graphs of 5000 and 10000 nodes using three different models.

Table 2 The width of the graph in three different networkx models as the density increases on graphs of 10000 nodes.

$\|V\| = 10000$						
Av. Degree	5	10	20	40	80	160
	BA					
Width	3282	2066	1172	678	351	198
	ER					
Width	1561	802	409	219	110	58
	WS, b=0.9					
Width	1101	378	93	27	20	18
	WS, b=0.3					
Width	12	4	4	4	4	4

3 DAG Reduction for Faster Transitivity

The importance of removing transitive edges in order to create an abstract graph utilizing paths and chains was first described in [20]. Their focus was on graph visualization techniques, while in this paper we apply a similar abstraction to solve the transitive closure problem. This concept of abstraction or reduction of a DAG may be useful in several applications beyond transitive closure or reachability. Therefore we state the following useful lemmas and Theorem 3:

▶ **Lemma 1.** *Given a chain decomposition D of a DAG $G = (V, E)$, each vertex $v_i \in V$, $0 \leq i < |V|$, can have at most one outgoing non-transitive edge per chain.*

Proof. Given a graph $G(V, E)$, a decomposition $D(C_1, C_2, ..., C_{k_c})$ of G, and a vertex $v \in V$, assume vertex v has two outgoing edges, (v, t_1) and (v, t_2), and both t_1 and t_2 are in chain C_i. The vertices are in ascending topological order in the chain by definition. Assume t_1 has a lower topological rank than t_2. Thus, there is a path from t_1 to t_2, and accordingly a path from v to t_2 through t_1. Hence, the edge (v, t_2) is transitive. See Figure 3a. ◀

▶ **Lemma 2.** *Given a chain decomposition D of a DAG $G = (V, E)$, each vertex $v_i \in V$, $0 \leq i < |V|$, can have at most one incoming non-transitive edge per chain.*

Proof. Similar to the proof of Lemma 1, see Figure 3b. ◀

▶ **Theorem 3.** *Let $G = (V, E)$ be a DAG with width w. The non-transitive edges of G are less than or equal to $width * |V|$, in other words $|E_{red}| = |E| - |E_{tr}| \leq width * |V|$.*

Proof. Given any DAG G and its width w, there is a chain decomposition of G with w number of chains. By Lemma 1, every vertex of G could have only one outgoing, non-transitive edge per chain. The same holds for the incoming edges, according to Lemma 2. Thus the non-transitive edges of G are bounded by $width * |V|$. ◀

An interesting application of the above is that we can find a significantly large subset of E_{tr} in linear time as follows: Given any chain (or path) decomposition with k_c chains, we can trace the vertices and their outgoing edges and keep the edges that point to the lowest point of each chain, rejecting the rest as transitive. We do the same for the incoming edges keeping the edges that come from the highest point (i.e., the vertex with the highest topological rank) of each chain. In this fashion we find a superset of E_{red}, call it E'_{red}, in

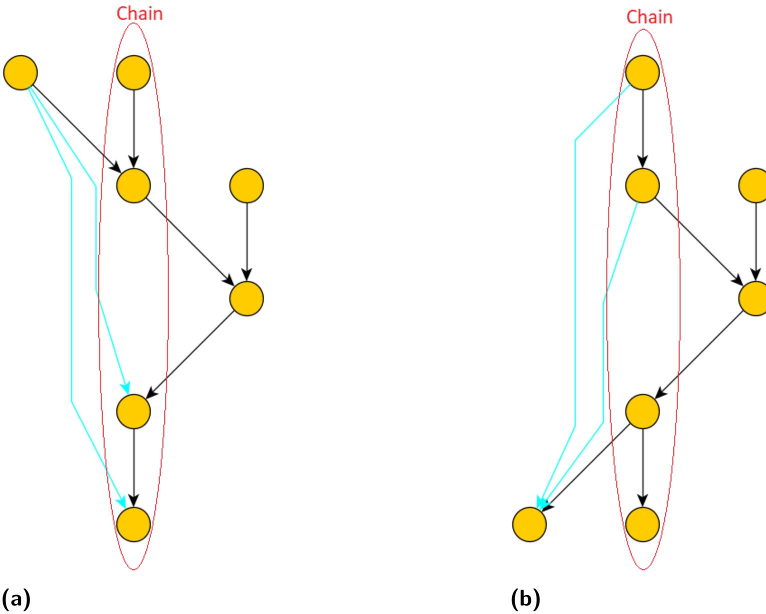


Figure 3 The light blue edges are transitive. (a) shows the outgoing transitive edges that end in the same chain. (b) shows the incoming transitive edges that start from the same chain.

linear time. Equivalently, we can find $E'_{tr} = E - E'_{red}$. E'_{tr} is a significantly large subset of E_{tr} since $|E - E'_{tr}| = |E'_{red}| \leq k_c * |V|$. Clearly, this approach can be used as a linear-time preprocessing step in order to substantially reduce the size of any DAG while keeping the same transitive closure as the original DAG G. Consequently, this will speed up every transitive closure algorithm bounding the number of edges of any input graph, and the indegree and outdegree of every vertex by k_c. For example, algorithms based on tree cover, see [1, 6, 25, 27], are practical on sparse graphs and can be enhanced further with such a preprocessing step that removes transitive edges. Additionally, this approach may have practical applications in dynamic or hybrid transitive closure techniques: If one chooses to answer queries online by using graph traversal for every query, one could reduce the size of the graph with a fast (linear-time) preprocessing step that utilizes chains. Also, in the case of insertion/deletion of edges one could quickly decide if the edges to add or remove are transitive. Transitive edges do not affect the transitive closure, hence no updates are required. This could be practically useful in dynamic insertion/deletion of edges.

4 Reachability Indexing Scheme

In this section, we present an important application that uses a chain decomposition of a DAG. Namely, we solve the transitive closure problem by creating a reachability indexing scheme that is based on a chain decomposition and we evaluate it by running extensive experiments. Our experiments shed light on the interplay of various important factors as the density of the graphs increases.

Jagadish described a compressed transitive closure technique in 1990 [16] by applying an indexing scheme and simple path/chain decomposition techniques. His method uses successor lists and focuses on the compression of the transitive closure. Thus his scheme does not answer queries in constant time. Simon [23], describes a technique similar to [16]. His technique is based on computing a path decomposition, thus boosting the method presented

in [13]. The linear time heuristic used by Simon is similar to the Chain Order Heuristic of [16]. A different approach is a graph structure referred to as path-tree cover introduced in [17], similarly, the authors utilize a path decomposition algorithm to build their labeling.

In the following subsections, we describe how to compute an indexing scheme in $O(|E_{tr}|+k_c*|E_{red}|)$ time, where k_c is the number of chains (in any given chain decomposition) and $|E_{red}|$ is the number of non-transitive edges. Following the observations of Section 3, the time complexity of the scheme can be expressed as $O(|E_{tr}|+k_c*|E_{red}|) = O(|E_{tr}|+k_c*width*|V|)$ since $|E_{red}| \leq width*|V|$. Using an approach similar to Simon's [23] our scheme creates arrays of indices to answer queries in constant time. The space complexity is $O(k_c*|V|)$.

For our experiments, we utilize the chain decomposition approach of [19], which produces smaller decompositions than previous heuristic techniques, without any considerable run-time overhead. Additionally, this heuristic, called NH_conc, will perform better than any path decomposition algorithms as will be explained next. Thus the indexing scheme is more efficient both in terms of time and space requirements. Furthermore, the experimental work shows that, as expected, the chains rarely have the same length. Usually, a decomposition consists of a few long chains and several short chains. Hence, for most graphs it is not even possible to have $|E_{red}| = width*|V|$, which assumes the worst case for the length of each chain. In fact, $|E_{red}|$ is usually much lower than that and the experimental results presented in Tables 3 and 4 confirm this observation in practice.

Given a directed graph with cycles, we can find the strongly connected components (SCC) in linear time. Since any vertex is reachable from any other vertex in the same SCC (they form an equivalence class), all vertices in a SCC can be collapsed into a supernode. Hence, any reachability query can be reduced to a query in the resulting directed acyclic graph (DAG). This is a well-known step that has been widely used in many applications. Therefore, without loss of generality, we assume that the input graph to our method is a DAG. The following general steps describe how to compute the reachability indexing scheme:

1. Compute a Chain decomposition
2. Sort all Adjacency Lists
3. Create an Indexing Scheme

In Step 1, we use our chain decomposition technique that runs in $O(|E|+c*l)$ time. In Step 2, we sort all the adjacency lists in $O(|V|+|E|)$ time. Finally, we create an indexing scheme in $O(|E_{tr}|+k_c*|E_{red}|)$ time and $O(k_c*|V|)$ space. Clearly, if the algorithm of Step 1 computes fewer chains then Step 3 becomes more efficient in terms of time and space.

4.1 The Indexing Scheme

Given any chain decomposition of a DAG G with size k_c, an indexing scheme will be computed for every vertex that includes a pair of integers and an array of size k_c of indexes. A small example is depicted in Figure 4. The first integer of the pair indicates the node's chain and the second its position in the chain. For example, vertex 1 of Figure 4 has a pair $(1,1)$. This means that vertex 1 belongs to the $1st$ chain, and it is the $1st$ element in it. Given a chain decomposition, we can easily construct the pairs in $O(|V|)$ time using a simple traversal of the chains. Every entry of the k_c-size array represents a chain. The i-th cell represents the i-th chain. The entry in the i-th cell corresponds to the lowest point of the i-th chain that the vertex can reach. For example, the array of vertex 1 is $[1,2,2]$. The first cell of the array indicates that vertex 1 can reach the first vertex of the first chain (can reach itself, reflexive property). The second cell of the array indicates that vertex 1 can reach the second vertex of the second chain (There is a path from vertex 1 to vertex 7). Finally, the third cell of the array indicates that vertex 1 can reach the second vertex of the third chain.

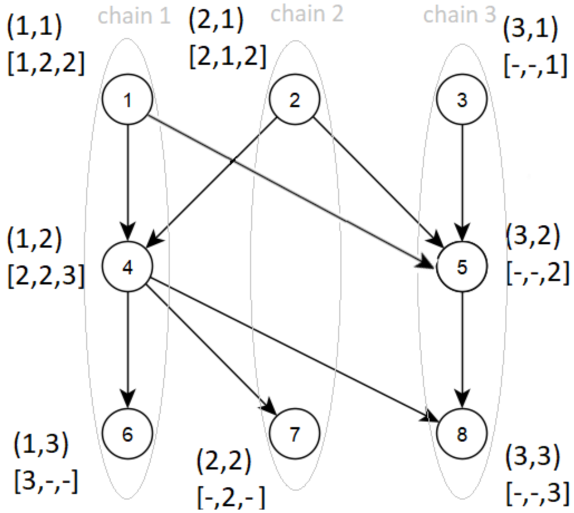


Figure 4 An example of an indexing scheme.

Notice that we do not need the second integer of all pairs. If we know the chain a vertex belongs to, we can conclude its position using the array. We use this presentation to simplify the understanding of the users.

The process of answering a reachability query is simple. Assume, there is a source vertex s and a target vertex t. To find if vertex t is reachable from s, we first find the chain of t, and we use it as an index in the array of s. Hence, we know the lowest point of t's chain vertex s can reach. s can reach t if that point is less than or equal to t's position, else it cannot.

4.2 Sorting Adjacency lists

Next, we use a linear time algorithm to sort all the adjacency lists of immediate successors in ascending topological order. See Algorithm 2 in Appendix A.2. The algorithm maintains a stack for every vertex that indicates the sorted adjacency list. Then it traverses the vertices in reverse topological order, $(v_n, ..., v_1)$. For every vertex v_i, $1 \leq i \leq n$, it pushes v_i into all immediate predecessors' stacks. This step can be performed as a preprocessing step, even before receiving the chain decomposition. To emphasize its crucial role in the efficient creation of the indexing scheme, if the lists are not sorted then the second part of the time complexity would be $O(k_c * |E|)$ instead of $O(k_c * |E_{red}|)$.

4.3 Creating the Indexing Scheme

Now we present Algorithm 1 that constructs the indexing scheme. The first for-loop initializes the array of indexes. For every vertex, it initializes the cell that corresponds to its chain. The rest of the cells are initialized to infinity. The indexing scheme initialization is illustrated in Figure 5. The dashes represent the infinite values. Notice that after the initialization, the indexes of all sink vertices have been calculated. Since a sink has no successors, the only vertex it can reach is itself.

The second for-loop builds the indexing scheme. It goes through vertices in descending topological order. For each vertex, it visits its immediate successors (outgoing edges) in ascending topological order and updates the indexes. Suppose we have the edge (v, s), and we have calculated the indexes of vertex s (s is an immediate successor of v). The process

Algorithm 1 Indexing Scheme.

1: **procedure** CREATE INDEXING SCHEME(G, T, D)
INPUT: A DAG $G = (V, E)$, a topological sorting T of G, and the decomposition D of G.
2: **for each vertex:** $v_i \in G$ **do**
3: v_i.indexes $\leftarrow$ new table[size of D]
4: v_i.indexes.fill(∞)
5: $ch_no \leftarrow v_i$'s chain index
6: $pos \leftarrow v_i$'s chain position
7: v_i.indexes[ch_no] $\leftarrow pos$
8: **end for**
9: **for each vertex** v_i **in reverse topological order do**
10: **for each adjacent target vertex** t **of** v_i **in ascending topological order do**
11: $t_ch \leftarrow$ chain index of t
12: $t_pos \leftarrow$ chain position of t
13: **if** $t_pos < v_i$.indexes[t_ch] **then** ▷ (v_i, t) is not transitive
14: v_i.updateIndexes($t.indexes$)
15: **end if**
16: **end for**
17: **end for**
18: **end procedure**

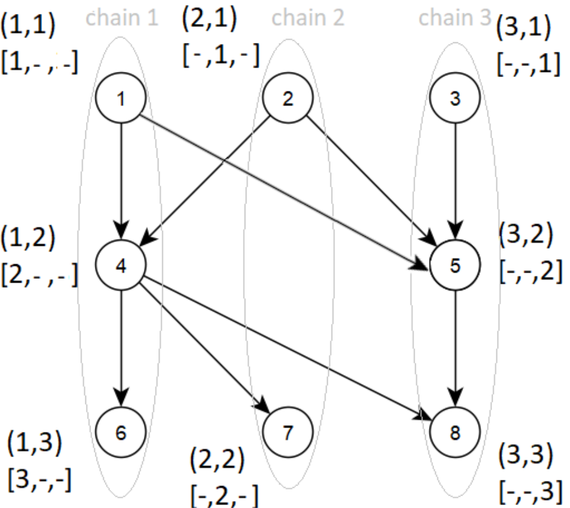


Figure 5 Initialization of indexes.

of updating the indexes of v with its immediate successor, s, means that s will pass all its information to vertex v. Hence, vertex v will be aware that it can reach s and all its successors. Assume the array of indexes of v is $[a_1, a_2, ..., a_{k_c}]$ and the array of s is $[b_1, b_2, ..., b_{k_c}]$. To update the indexes of v using s, we merely trace the arrays and keep the smallest values. For every pair of indexes (a_i, b_i), $0 \leq i < kc$, the new value of a_i will be $\min\{a_i,\, b_i\}$. This process needs k_c steps.

▶ **Lemma 4.** *Given a vertex v and the calculated indexes of its successors, the while-loop of Algorithm 1 (lines 10-17) calculates the indexes of v by updating its array with its non-transitive outgoing edges' successors. (Proof in Appendix A.1).*

Combining the previous algorithms and results we conclude this section with the following:

▶ **Theorem 5.** *Let $G = (V, E)$ be a DAG. Algorithm 1 computes an indexing scheme for G in $O(|E_{tr}| + k_c * |E_{red}|)$ time. (Proof in Appendix A.1).*

As described in the introduction, a parameterized linear-time algorithm for computing the minimum number of chains was recently presented in [5]. Its time complexity is $O(k^3|V|+|E|)$ where k is the minimum number of chains, which is equal to the width of G. If we use this chain decomposition as input to Algorithm 1 it computes an indexing scheme for G in parameterized linear time. This implies that the transitive closure of G can be computed in parameterized linear time. Hence we have the following:

▶ **Corollary 6.** *Let $G = (V, E)$ be a DAG. Algorithm 1 can be used to compute an indexing scheme for G in parameterized linear time. Hence the transitive closure of G can be computed in parameterized linear time.*

4.4 Experimental Results

We conducted experiments using the same graphs of 5000 and 10000 nodes as we described in Section 2 that were produced by the four different models of Networkx [14] and the Path-Based model of [21]. We computed a chain decomposition using the algorithm introduced in [19], called NH_conc, and created an indexing scheme using Algorithm 1. For simplicity, we assume that the adjacency lists of the input graph are sorted, using Algorithm 2, as a preprocessing step. We report our experimental results in Tables 3 and 4 for graphs with 5000 nodes and graphs with 10000 nodes, respectively.

In theory, the phase of the indexing scheme creation needs $O(|E_{tr}| + k_c * |E_{red}|)$ time. However, the experimental results shown in the tables reveal some interesting (and expected) findings in practice: As the average degree increases and the graph becomes denser, (a) the cardinality of E_{red} remains almost stable; and (b) the number of chains decrease. The observation that the number of non-transitive edges, E_{red}, does not vary significantly as the average degree increases, implies that the number of transitive edges, $|E_{tr}|$, increases proportionally to the increase in the number of edges, since $(E_{tr} = E - E_{red})$. Since the algorithm merely traces in linear time the transitive edges, the growth of $|E_{tr}|$ affects the run time only linearly. As a result, the run time of our technique does not increase significantly as the the size (number of edges) of the input graph increases. In order to demonstrate this fact visually, we show the curves of the running time for the graphs of 10000 nodes produced by the ER model in Figure 6 (see Appendix A). The flat (blue line) represents the run time to compute the indexing scheme, and the curve (red line) the run time of the DFS-based algorithm for computing the transitive closure (TC). Clearly, the time of the DFS-based algorithm increases as the average degree increases, while the time of the indexing scheme is a straight line almost parallel to the x-axis. All models of Tables 3 and 4 follow this pattern.

Table 3 Experimental results for the indexing scheme for graphs of 5000 nodes.

$\|V\| = 5000$								
Average Degree	Number of Chains	$\|E_{tr}\|$	$\|E_{red}\|$	$\|E_{tr}\|/\|E\|$	NH_conc Time (ms)	Indexing Scheme Time (ms)	Total time (ms)	TC
	BA							
5	1630	8054	18921	0.32	3	101	104	137
10	1055	28230	21670	0.57	12	79	91	333
20	664	75801	23799	0.76	6	54	60	638
40	335	180815	22504	0.89	10	48	58	1418
80	207	382422	20854	0.95	122	118	240	3018
160	163	770771	17660	0.98	25	107	132	5464
	ER							
5	923	3440	21466	0.14	6	67	73	172
10	492	24761	25425	0.49	10	51	61	487
20	252	75312	24646	0.75	5	26	31	1079
40	139	175809	22634	0.89	46	51	97	2896
80	70	378015	19435	0.95	16	50	66	5260
160	38	769919	16843	0.98	98	138	236	8609
	WS, b=0.9							
5	687	7742	17258	0.30	13	71	84	393
10	212	37992	12008	0.76	11	18	29	817
20	60	89272	10728	0.89	23	22	45	1530
40	25	186486	13514	0.93	47	45	92	3704
80	20	386294	13706	0.97	115	103	218	6172
160	17	787066	12934	0.98	253	207	460	9173
	WS, b=0.3							
5	9	18421	6579	0.74	11	8	19	910
10	4	43505	6495	0.87	8	11	19	1107
20	4	93490	6510	0.93	18	18	36	2176
40	5	193416	6584	0.97	17	18	35	4753
80	4	393348	6652	0.98	98	82	180	7949
160	5	793430	6570	0.99	250	166	416	11757
	PB, Paths=70							
5	86	14155	10809	0.57	8	7	15	206
10	101	36801	13102	0.74	7	12	19	313
20	107	84168	15419	0.85	7	15	22	890
40	93	181388	16988	0.91	49	216	265	2584
80	73	376220	17303	0.96	128	163	291	4603
160	51	758207	16566	0.98	55	141	196	9358

Table 4 Experimental results for the indexing scheme for graphs of 10000 nodes.

$\lvert V\rvert = 10000$								
Average Degree	Number of Chains	$\lvert E_{tr}\rvert$	$\lvert E_{red}\rvert$	$\lvert E_{tr}\rvert/\lvert E\rvert$	NH_conc Time (ms)	Indexing Scheme Time (ms)	Total time (ms)	TC
	BA							
5	3341	14544	35431	0.29	7	278	285	441
10	2159	53503	46397	0.54	14	231	245	1379
20	1264	147791	51809	0.74	15	218	233	3347
40	752	355854	52465	0.85	28	188	216	7700
80	400	764926	48350	0.94	271	322	593	14632
160	228	1560464	42967	0.97	81	264	345	24601
	ER							
5	1837	5595	44401	0.11	12	200	212	600
10	1003	44813	55366	0.45	9	161	170	1935
20	516	144276	55310	0.72	16	110	126	6031
40	271	347323	52620	0.87	25	101	126	13522
80	139	749781	46666	0.94	40	145	185	23052
160	72	1548153	39710	0.97	73	249	322	37613
	WS, b=0.9							
5	1332	13353	36647	0.27	12	175	187	1213
10	447	74782	25218	0.75	9	53	62	3829
20	100	178930	21070	0.89	13	32	45	9279
40	29	373054	26946	0.93	24	60	84	13144
80	24	771374	28626	0.96	266	247	513	25585
160	22	1571957	28043	0.98	80	232	312	36507
	WS, b=0.3							
5	12	36816	13184	0.73	27	19	46	3468
10	4	86804	13196	0.86	18	45	63	5063
20	4	186756	13244	0.93	10	42	52	12156
40	4	386751	13249	0.97	19	48	67	21055
80	4	786840	13160	0.98	237	187	424	31016
160	4	1586896	13104	0.99	62	167	229	40704
	PB, Paths=100							
5	125	8182	16810	0.33	12	16	28	240
10	141	74182	25722	0.74	11	30	41	937
20	153	168839	30728	0.85	13	43	56	5015
40	142	363753	34606	0.91	27	78	105	13797
80	120	756578	36918	0.96	56	142	198	27904
160	89	1538101	36496	0.98	77	265	342	41235

Apparently, there is a trade-off to consider when building an indexing scheme deploying the technique of [19]. The heuristic performs concatenations between paths. For every successful concatenation, the extra runtime overhead is $O(l)$, where l is the longest path between the two concatenated paths. The unsuccessful concatenations do not cause any overhead. Assume that we have a path decomposition, and then we perform chain concatenation. If there is no concatenation between two paths, the concatenation algorithm will run in linear time.

On the other hand, if there are concatenations, for each one of them, then the cost is $O(l)$ time, but the savings in the indexing scheme creation is $\Theta(|V|)$ in space requirements and $\Theta(|E_{red}|)$ in time, since every concatenation reduces the needed index size for every vertex by one. Hence, instead of computing a simple path decomposition (in linear time) the use of a path concatenation procedure in order to create a more compact indexing scheme faster is preferred for almost all applications. Another interesting and to some extent surprising observation that comes from the results of Tables 3 and 4 is that the transitive edges for almost all models of the graphs of 5000 and 10000 nodes with average degree above 20 are above 85%, i.e., $|E_{tr}|/|E| \geq 85\%$, see the appropriate columns in both tables. In some cases where the graphs are a bit denser, the percentage grows above 95%. This observation has important implications in designing practical algorithms for faster transitive closure computation in both the static and the dynamic case.

5 Conclusions and Extensions

Our extensive experiments expose the practical behavior of (1) the *width*, (2) E_{red}, and (3) E_{tr} as the density and size of graphs grow. Furthermore, we show that the set E_{red} is bounded by $width * |V|$ and show how to find a substantially large subset of E_{tr} in linear time given any path/chain decomposition. These facts have important practical implications to the reachability problem and show the potential applications of these techniques in a dynamic setting where edges and nodes are inserted and deleted from a (very large) graph. Although our techniques were not developed for the dynamic case, the picture that emerges is very interesting.

According to our experimental results, see Tables 3 and 4, the overwhelming majority of edges in a DAG are transitive. The insertion or deletion of a transitive edge clearly requires a constant time update since it does not affect transitivity, and can be detected in constant time. On the other hand, the insertion or removal of a non-transitive edge may require a minor or major recomputation in order to reestablish a correct chain decomposition. Similarly, since the nodes of the DAG are topologically ordered, the insertion of an edge that goes from a high node to a low node signifies that the SCCs of the graph have changed, perhaps locally. However, even if the insertion/deletion of new nodes/edges causes significant changes in the reachability index (transitive closure) one can simply recompute a chain decomposition in linear or almost linear time, and then recompute the reachability scheme in parameterized linear time, $O(|E_{tr}| + k_c * |E_{red}|)$, and $O(k_c * |V|)$ space, which is still very efficient in practice, see [15] for a very recent comparison of practical fully dynamic transitive closure techniques. We plan to work on the problems that arise in the computation of dynamic path/chain decomposition and reachability indexes in the future.

References

1. Rakesh Agrawal, Alexander Borgida, and Hosagrahar Visvesvaraya Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *ACM SIGMOD Record*, 18(2):253–262, 1989.
2. Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. SIAM, 2021.
3. Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.
4. Manuel Cáceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I Tomescu. A linear-time parameterized algorithm for computing the width of a dag. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 257–269. Springer, 2021.
5. Manuel Cáceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I Tomescu. Sparsifying, shrinking and splicing for minimum path cover in parameterized linear time. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 359–376. SIAM, 2022.
6. Li Chen, Amarnath Gupta, and M Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *Proceedings of the 31st international conference on Very large data bases*, pages 493–504. Citeseer, 2005.
7. Yangjun Chen and Yibin Chen. On the dag decomposition. *British Journal of Mathematics and Computer Science*, 2014. 10(6): 1-27, 2015, Article no.BJMCS.19380, ISSN: 2231-0851. URL: `https://www.researchgate.net/publication/285591312_Pre-Publication_Draft_2015_BJMCS_19380`.
8. R. P. DILWORTH. A decomposition theorem for partially ordered sets. *Ann. Math.*, 52:161–166, 1950.
9. Fulkerson DR. Note on dilworth's embedding theorem for partially ordered sets. *Proc. Amer. Math. Soc.*, 52(7):701–702, 1956.
10. Ran Duan, Hongxun Wu, and Renfei Zhou. Faster matrix multiplication via asymmetric hashing, 2023. `arXiv:2210.10173`.
11. P Erdös and A Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 1959.
12. Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
13. Alla Goralčíková and Václav Koubek. A reduct-and-closure algorithm for graphs. In *International Symposium on Mathematical Foundations of Computer Science*, pages 301–307. Springer, 1979.
14. Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.
15. Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Faster fully dynamic transitive closure in practice. *CoRR*, abs/2002.00813, 2020. `arXiv:2002.00813`.
16. H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, December 1990. `doi:10.1145/99935.99944`.
17. Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 595–608, 2008.
18. Shimon Kogan and Merav Parter. Beating matrix multiplication for n^{1/3}-directed shortcuts. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
19. Giorgos Kritikakis and Ioannis G Tollis. Fast and practical dag decomposition with reachability applications. *arXiv e-prints*, 2022. `arXiv:2212.03945`.
20. Panagiotis Lionakis, Giacomo Ortali, and Ioannis Tollis. Adventures in abstraction: Reachability in hierarchical drawings. In *Graph Drawing and Network Visualization: 27th International Symposium, GD 2019, Prague, Czech Republic, September 17–20, 2019, Proceedings*, pages 593–595, 2019.

21 Panagiotis Lionakis, Giacomo Ortali, and Ioannis G Tollis. Constant-time reachability in dags using multidimensional dominance drawings. *SN Computer Science*, 2(4):1–14, 2021.

22 Veli Mäkinen, Alexandru I Tomescu, Anna Kuosmanen, Topi Paavilainen, Travis Gagie, and Rayan Chikhi. Sparse dynamic programming on dags with small width. *ACM Transactions on Algorithms (TALG)*, 15(2):1–21, 2019.

23 K. SIMON. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58(1-3):325–346, 1988.

24 Volker Strassen et al. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

25 Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 845–856, 2007.

26 Jan Van Den Brand, Yin Tat Lee, Yang P Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and ℓ1-regression in nearly linear time for dense instances. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 859–869, 2021.

27 Haixun Wang, Hao He, Jun Yang, Philip S Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 75–75. IEEE, 2006.

28 Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *nature*, 393(6684):440–442, 1998.

A Appendix

A.1 Proofs

Proof of Lemma 4. Updating the indexes of vertex v with all its immediate successors will make v aware of all its descendants. The while-loop of Algorithm 1 does not perform the update function for every direct successor. It skips all the transitive edges. Assume there is such a descendant t and the transitive edge (v, t). Since the edge is transitive, we know by definition that there exists a path from v to t with a length of more than 1. Suppose that the path is $(v, v_1, .., t)$. Vertex v_1 is a predecessor of t and immediate successor of v. Hence it has a lower topological rank than t. Since, while-loop examines the incident vertices in ascending topological order, then vertex t will be visited after vertex v_1. The opposite leads to a contradiction. Consequently, for every incident transitive edge of v, the loop firstly visits a vertex v_1 which is a predecessor of t. Thus vertex v will be firstly updated by v_1 and it will record the edge (v, t) as transitive. Hence there is no reason to update the indexes of vertex v with those of vertex t since the indexes of t will be greater than or equal to those of v. ◀

Proof of Theorem 5. In the initialization step, the indexes of all sink vertices have been computed as we described above. Taking vertices in reverse topological order, the first vertex we meet is a sink vertex. When the for-loop of line 9 visits the first non-sink vertex, the indexes of its successors are computed (all its successors are sink vertices). According to Lemma 5.1, we can calculate its indexes, ignoring the transitive edges. Assume the for-loop has reached vertex v_i in the ith iteration, and the indexes of its successors are calculated. Following Lemma 5.1, we can calculate its indexes. Hence, by induction, we can calculate the indexes of all vertices, ignoring all $|E_{tr}|$ transitive edges in $O(|E_{tr}| + k_c * |E_{red}|)$ time. ◀

Algorithm 2 Sorting Adjacency lists.

```
1:  procedure Sort(G, t)
    INPUT: A DAG G = (V, E)
2:      for each vertex: v_i ∈ G do
3:          v_i.stack ← new stack()
4:      end for
5:      for each vertex v_i in reverse topological order do
6:          for every incoming edge e(s_j, v_i) do
7:              s_j.stack.push(v_i)
8:          end for
9:      end for
10: end procedure
```

A.2 Sorting Adjacency lists Algorithm

▶ **Lemma 7.** *Algorithm 2 sorts the adjacency lists of immediate successors in ascending topological order, in linear time.*

Proof. Assume that there is a stack $(u_1, ..., u_n)$, u_1 is at the top of the stack. Assume that there is a pair (u_j, u_k) in the stack, where u_j has a bigger topological rank than u_k and u_j precedes u_k. This means that the for-loop examined u_j before u_k. Since the algorithm processes the vertices in reverse topological order, this is a contradiction. Vertex u_j cannot precede u_k if it were examined first by the for-loop. The algorithm traces all the incoming edges of every vertex. Therefore, it runs in linear time. ◀

A.3 Figures

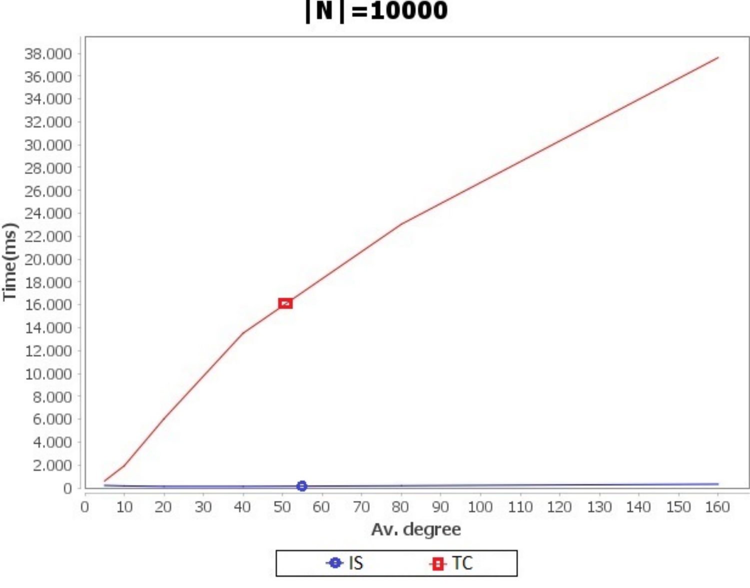


Figure 6 Run time comparison between the Indexing Scheme (blue line) and TC (red line) for ER model on graphs of 10000 nodes, see Table 4.

Partitioning the Bags of a Tree Decomposition into Cliques

Thomas Bläsius
Karlsruhe Institute of Technology, Germany

Maximilian Katzmann
Karlsruhe Institute of Technology, Germany

Marcus Wilhelm
Karlsruhe Institute of Technology, Germany

Abstract

We consider a variant of treewidth that we call *clique-partitioned treewidth* in which each bag is partitioned into cliques. This is motivated by the recent development of FPT-algorithms based on similar parameters for various problems. With this paper, we take a first step towards computing clique-partitioned tree decompositions.

Our focus lies on the subproblem of computing clique partitions, i.e., for each bag of a given tree decomposition, we compute an optimal partition of the induced subgraph into cliques. The goal here is to minimize the product of the clique sizes (plus 1). We show that this problem is NP-hard. We also describe four heuristic approaches as well as an exact branch-and-bound algorithm. Our evaluation shows that the branch-and-bound solver is sufficiently efficient to serve as a good baseline. Moreover, our heuristics yield solutions close to the optimum. As a bonus, our algorithms allow us to compute first upper bounds for the clique-partitioned treewidth of real-world networks. A comparison to traditional treewidth indicates that clique-partitioned treewidth is a promising parameter for graphs with high clustering.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases treewidth, weighted treewidth, algorithm engineering, cliques, clustering, complex networks

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.3

Related Version *Full Version*: https://arxiv.org/abs/2302.08870

Supplementary Material *Software (Source Code)*: https://github.com/marcwil/cptw_code
archived at swh:1:dir:d611a7236b805d6eb409147b28b514f9132f96c1
Software (Source Code: Archive, Docker Image): https://doi.org/10.5281/zenodo.7816985

1 Introduction

The treewidth is a measure for how treelike a graph is in terms of its separators. It is defined via a tree decomposition, a collection of vertex separators called *bags* that are arranged in a tree structure. The size of the largest bag determines the width of the decomposition and the treewidth of a graph is the minimum width over all tree decompositions.

The concept of treewidth has its origins in graph theory with some deep structural insights [22, 24]. Additionally, there are algorithmic implications. Intuitively speaking, the separators of a tree decomposition split the graph into pieces that can be solved independently except for minor dependencies at the separators. This is often formalized using a dynamic program over the tree decomposition, yielding an FPT-algorithm (fixed-parameter tractable) with the treewidth as parameter [9]. As this is a versatile framework that can be applied to many problems, it comes to no surprise that there has been quite a bit of effort to develop algorithms for computing low-width tree decompositions (see, e.g., [15, 16]).

© Thomas Bläsius, Maximilian Katzmann, and Marcus Wilhelm;
licensed under Creative Commons License CC-BY 4.0

A major obstruction for low treewidth are large cliques, which inevitably lead to large separators. This is particularly true for so-called complex networks, i.e., graphs with strong community structure and heterogeneous degree distribution, which appear in various domains such as communication networks, social networks, or webgraphs. One could, however, hope for two aspects that together mitigate this negative effect of large cliques. First, though some separators need to be large, these separators are structurally simple, e.g., they form a clique or can be covered with few cliques. Second, separators that are large but structurally simple still let us solve the separated pieces individually with low dependence between them. The first hope is supported by the fact that the treewidth is asymptotically equal to the clique number in hyperbolic random graphs [5]; a popular model for complex networks [21]. This indicates that cliques are indeed the main obstruction for low treewidth in these kinds of networks. The second hope is supported by the results of de Berg et al. [14], who introduced the concept $\mathcal{P}$-flattened tree decompositions. There, the graph is partitioned into cliques and the width of the tree decomposition is measured in terms of the (weighted) number of cliques in a bag. Thus, the width does measure the complexity of separators rather than their size. Based on this definition, the authors then show that these structurally simple separators help to solve various graph problems efficiently.

To the best of our knowledge, these extended concepts have not yet been studied from a practical perspective. With this paper, we want to initiate this line of research by addressing two questions. First, can such clique-partitioned tree decompositions lead to substantially smaller width values than classical tree decompositions? Second, how can such tree decompositions be computed? For the second question, we design and evaluate different algorithmic strategies for computing a novel yet closely related variant of tree decompositions. Our experiments yield some interesting algorithmic insights and provide a good starting point for further development. On networks that do exhibit clique structures, the constructed tree decompositions indeed have sufficiently low width to answer the first question affirmatively. We believe that there is plenty of room for improvement in our approaches, which may yield even better insights into the applicability of the new parameter. In the following, we discuss related work before stating our contribution more precisely.

1.1 Related Work

There are multiple lines of research that investigate variants of treewidth where additional structural properties are taken into account. As mentioned above, De Berg et al. [14] propose a variant of tree decompositions where the initial graph is partitioned into cliques (or unions of constantly many connected cliques) that are contracted into weighted vertices. The weight of a clique of size s is $\log(s + 1)$ and the weight of a bag of the tree decomposition is the sum of its weights. Using this technique, they give subexponential algorithms for a range of problems on geometric intersection graphs, including Independent Set, Steiner Tree and Feedback Vertex Set. For some of these problems, the algorithms are also representation agnostic, while for most others, the geometric representation is required. They also prove that the running time of the algorithms is tight under the exponential-time-hypothesis (ETH). Kisfaludi-Bak [20] applied the same algorithmic framework to intersection graphs of constantly sized objects in the hyperbolic plane.

A similar parameter called *tree clique width* has been proposed by Aronis [2]. Here, the idea is to consider tree decompositions where each bag is annotated with an edge clique cover (ecc) and where the size of the cover determines the width of a bag. The paper shows several hardness results and adapts common treewidth algorithms to the newly proposed parameter.

Another approach to capture graph structures that lead to high treewidth despite being structurally simple has been proposed by Dallard, Milanič, and Štorgel. They define the *independence number* of a tree decomposition as the size of the largest independent set of any of its bags and the *tree-independence number* of a graph as the minimum independence number of any tree decomposition [13]. This parameter connects to the more theoretical study of (tw, ω)-bounded graphs, i.e., graph classes in which the treewidth depends only on the clique number [11, 12]. This line of research is mostly concerned with the classification and characterization of the considered graph classes both in terms of graph theory and algorithmic exploitability. However, apart from a factor 8 approximation with running time $2^{O(k^2)} \cdot n^{O(k)}$ due to Dallard, Fomin, Golovach, Korhonen, and Milanič [10], we are not aware of any work that tries to actually build algorithms for this or similar parameters.

1.2 Contribution

In this paper, we propose *clique-partitioned* treewidth as a parameter that captures structurally simple separators in graphs. It can be seen as a close adaptation of $\mathcal{P}$-flattened treewidth [14], where we *first* compute a tree decomposition and *then* determine clique partitions of the subgraphs induced by the bags. Thus, instead of using a global clique partition of the whole graph, we consider clique partitions that are local to a single bag.

The remainder of this paper is structured as follows. In Section 2, we formalize our definition for clique-partitioned treewidth and prove several statements comparing it with $\mathcal{P}$-flattened treewidth. In Section 3, we present multiple approaches to compute low-weight clique partitions for the bags of a tree decomposition. They include various heuristic methods, as well as an exact branch-and-bound algorithm for which we propose several adjustments with the potential to improve its running time in practice. Afterwards, in Section 4 we combine an implementation of our approaches with existing methods for computing tree decompositions and study the upper bounds on the clique-partitioned treewidth of real-world networks. Furthermore, we evaluate the performance of the exact and heuristic clique partition solvers proposed in Section 3. Due to space limitations, some proofs are in Appendix B.

2 Clique-partitioned treewidth

We first introduce some basic notation and give the definition for traditional tree decompositions. We write $[n] = \{1, \ldots, n\}$ for the first n natural numbers. Throughout the paper, we assume graphs $G = (V, E)$ to be simple and undirected and write $V(G)$ and $E(G)$ for the sets of vertices and edges, respectively. For a subset $X \subseteq V$ we write $G[X]$ for the subgraph of G induced by X.

A *tree decomposition* of G is a pair (T, B), for a tree T and a function B mapping vertices of T to subsets of V called *bags* such that T and B have the following three properties: (1) every vertex of G is contained in some bag, (2) for every edge, there is a bag containing both endpoints, and (3) for any vertex v of G, the set of bags containing v forms a connected subtree of T. The *width* of a tree decomposition is the size of the largest bag minus 1. The *treewidth* $\mathrm{tw}(G)$ is the smallest width obtainable by any tree decomposition of G.

We define a *clique-partitioned* tree decomposition of G as a tree decomposition where for every $t \in V(T)$ we have a partition $\mathcal{P}_t$ of the subgraph induced by the corresponding bag (i.e., the graph $G[B(t)]$) into cliques. Following de Berg et al. [14], we define the *weight* of a clique C as $\log(|C| + 1)$ and the weight of a bag $B(t)$ as the sum of weights of the cliques in its partition $\mathcal{P}_t$. Throughout this paper we assume 2 to be the default base of logarithms.

The weight of a clique-partitioned tree decomposition is the maximum weight of any of its bags and the *clique-partitioned treewidth* (short: cp-treewidth) of G, denoted by $\text{cptw}(G)$, is the minimum weight of any clique-partitioned tree decomposition.

As mentioned before, the clique-partitioned treewidth is closely related to the parameter defined by de Berg et al. [14]. For a clique partition $\mathcal{P}$ of the whole graph G, we say that a *$\mathcal{P}$-flattened tree decomposition* is a clique-partitioned tree decomposition of G where the partition into cliques within a bag is induced by the global partition $\mathcal{P}$. As before, the weight of a $\mathcal{P}$-flattened tree decomposition is the maximum total weight of the cliques in any of its bags. In reference to the authors [14], we call the minimum weight over all $\mathcal{P}$ the BBKMZ-treewidth.

We note that our parameter can also be seen as an adaptation of *tree clique width* [2], where instead of considering the size of an *edge clique cover* of each bag, we consider the logarithmically weighted sum of clique sizes of a clique partition. That is, we are using the weight function of the $\mathcal{P}$-flattened treewidth to define a parameter which considers individual clique partitions, similar to tree clique width.

In the following, we compare the clique-partitioned treewidth to the more closely related BBKMZ-treewidth. First, as a global partition $\mathcal{P}$ can also be used locally in each bag of a clique-partitioned tree decomposition, we obtain that the clique-partitioned treewidth of a graph is at most its BBKMZ-treewidth. Additionally, the clique-partitioned treewidth can also be substantially smaller than the BBKMZ-treewidth, as shown in the following lemma.

▶ **Lemma 1.** *There is an infinite family of graphs $\mathcal{G}$ such that a graph $G \in \mathcal{G}$ with n vertices, has clique-partitioned treewidth in $\mathcal{O}(\log \log n)$ and BBKMZ-treewidth in $\Omega(\log n)$.*

Proof. The family $\mathcal{G}$ contains for every $h \in \mathbb{N}$ one graph G_h. The Graph G_h is a complete binary tree of height h, where additionally for every leaf ℓ we connect all h vertices that lie on a path between the root r and ℓ into a clique. Note that we have $h \in \Theta(\log n)$.

Let $\mathcal{P}_h$ be a clique partition of G_h. Then, via a simple induction over h, it is easy to see that in G_h there is a path between the root r and some leaf ℓ of G_h such that every vertex on the path belongs to a different partition class. These vertices form a clique in G_h that has to be prosent in some bag of any $\mathcal{P}_h$-flattened tree decomposition of G_h. This bag thus contains all h partition classes on the path and has weight $h \cdot \log(1+1) \in \Omega(\log n)$.

At the same time we can construct a clique-partitioned tree decomposition (T, σ), that has one bag for every path between the root r and each leaf ℓ. Then, T forms a path. As every bag consists of a single clique on h vertices, there is a clique partition of this tree decomposition with weighted width $\log(h+1) \in O(\log \log n)$. ◀

Finally, we show the algorithmic usefulness of clique-partitioned treewidth in the following lemma, which is an extension of the one proposed by de Berg et al. [14].

▶ **Lemma 2.** *Let G be a graph with a clique-partitioned tree decomposition (T, σ) of weight τ. Then a largest independent set of G can be found in $O(2^\tau \cdot \text{poly}(n))$ time.*

By the above argumentation, it follows that the clique-partitioned treewidth introduced in this paper is upper bounded by the version of de Berg et al. and can be exponentially lower. Additionally, it retains some power in solving NP-hard problems in FPT-time.

3 The weighted clique partition problem

We split the task of computing a clique-partitioned tree decomposition in two phases. First, we compute a tree decomposition, minimizing the traditional tree width. Secondly, fixing the structure and bags of this decomposition, we compute a clique partition for every bag. We

note that we already lose optimality by this separation, i.e., the result may be suboptimal even if we get optimal solutions in each of the two phases. However, we expect that small bags should also allow for low-weight clique partitions.

In the first phase, we use established algorithms for the computation of tree decompositions. Consequently, we focus on the second step in this section. To this end, we define the WEIGHTED CLIQUE PARTITION problem, short CLIQUE PARTITION. For a given graph G and an integer w, decide if there is a partition of $V(G)$ into cliques $P_1, \dots, P_k$ such that $\prod_{i\in[k]}(|P_i|+1) \leq w$. Note that this function differs from the one in the definition of clique-partitioned treewidth, but is equivalent, as $\sum_{i\in[k]} \log(|P_i|+1) = \log(\prod_{1\leq i\leq k}(|P_i|+1))$ and the logarithm is monotonic.

In the following, we prove some technical lemmas that are useful throughout the section, before showing that WEIGHTED CLIQUE PARTITION is NP-complete (Section 3.1). Afterwards, we give different heuristic approaches (Section 3.2) and an optimal branch-and-bound algorithm in (Section 3.3). We start with following lemma, which intuitively states that the weight of a partition is smaller the more imbalanced the individual weights are, i.e., moving a vertex from a smaller to a larger clique reduces the total weight.

▶ **Lemma 3.** *Let $a, b, c, d \in \mathbb{N}_0$ such that $a + b = c + d$ and $a \geq b$, $c \geq d$, $d > b$. Then $(a+1)(b+1) < (c+1)(d+1)$.*

With the above lemma (i.e., repeated applications thereof) we can compare the weight of two partitions.

▶ **Lemma 4.** *Let $\langle s_1, \dots, s_k\rangle$ and $\langle r_1, \dots, r_\ell\rangle$ be different non-increasing sequences of natural numbers such that $2 \leq k \leq \ell$, $\sum_{i\in[k]} s_i = \sum_{i\in[\ell]} r_i$, and $s_i \geq r_i$ for all $i \in [k-1]$. Then $\prod_{i\in[k]}(s_i+1) < \prod_{i\in[\ell]}(r_i+1)$.*

Proof. This follows from repeatedly applying Lemma 3 to go from $R = \langle r_1, \dots, r_\ell\rangle$ to $S = \langle s_1, \dots, s_k\rangle$ while reducing the product in each step. To make this precise let i be the first index where $s_i > r_i$. We adjust R by adding 1 to r_i and subtracting 1 from r_ℓ. Note that this maintains the sum. We apply Lemma 3 with $a = r_i + 1$, $b = r_\ell - 1$, $c = r_i$, and $d = r_\ell$. Then, we have $(a+1)(b+1) < (c+1)(d+1)$, i.e., the product of the adjusted sequence is smaller than that of the original sequence R. Moreover, after a finite number of steps, we reach S and thus the product for S is smaller than the product for R. ◀

3.1 Hardness

To prove that WEIGHTED CLIQUE PARTITION is NP-complete, we perform a reduction in two steps. We start with the NP-hard problem 3-COLORING. It asks for a given graph whether each vertex can be colored with one of three colors such that no two neighbors have the same color. As an intermediate problem in the reduction, we introduce WEIGHTED INDEPENDENT SET PARTITION. It is defined equivalently to WEIGHTED CLIQUE PARTITION, but instead of partitioning the graph into cliques, we partition it into independent sets, i.e., sets of pairwise non-adjacent vertices. Note that independent sets are cliques in the complement graph and vice versa. Thus, WEIGHTED INDEPENDENT SET PARTITION and WEIGHTED CLIQUE PARTITION are computationally equivalent. Thus, to obtain the following theorem, it remains to reduce 3-COLORING to WEIGHTED INDEPENDENT SET PARTITION.

▶ **Theorem 5.** *WEIGHTED CLIQUE PARTITION is NP-complete.*

Proof. Membership in NP is easy to see as polynomial time verification of a solution is straightforward. For hardness, we reduce from 3-COLORING to WEIGHTED INDEPENDENT SET PARTITION. Thus, we now assume that we are given a graph G and need to transform

it into a graph G' and integer w such that G can be colored with three colors if and only if G' has a partition into independent sets of weight at most w. We construct G' as follows. For every vertex v of G, we add two new vertices v_1 and v_2 that form a triangle together with v, but have no other edges. We denote $n = |V(G)|$ and set $w = (n+1)^3$. Note that any independent set in G' contains at most n vertices, because every appended triangle admits only one independent vertex.

Assume that G admits a proper three-coloring. This coloring directly translates to a three-coloring of G' as follows. Every vertex v of G keeps its color in G'. moreover, v_1 and v_2 each get one of the two other colors. Thus, the coloring classes in G' have size exactly n each and form an independent set partition with weight $(n+1)^3$.

If otherwise G does not admit a proper three-coloring, then neither does G' and there is no partition of G' into at most three independent sets. Any partition of $V(G')$ into more than three independent sets has a weight larger than $(n+1)^3$ by Lemma 4, as no independent set in G' can have more than n vertices. Consequently, G is three-colorable if and only if there is a partition of $V(G')$ into independent sets with weight at most $(n+1)^3$. ◀

3.2 Heuristic approaches

We now explain different approaches to solving the optimization variant of Weighted Clique Partition both optimally and heuristically.

Throughout this section we make use of the fact that enumerating all maximal cliques of a graph is not only output polynomial [19], but also highly feasible in practice as shown by Eppstein, Löffler, and Strash [17]. We use an implementation of their algorithm from the `igraph`[1] library.

Maximal clique heuristic. Recall from Lemma 3 that the weight function favors imbalanced clique sizes over more balanced ones. It therefore makes sense to try to find few large cliques that cover all vertices. A basic greedy heuristic that tries to achieve this works as follows. First, we enumerate all maximal cliques $\mathcal{C}$ of the graph. Then we iteratively add one clique to the partition by greedily selecting the clique with the largest number of remaining uncovered vertices. We call this the *maximal clique heuristic*.

In order to efficiently implement this heuristic, we use a priority queue to fetch the largest clique and keep track of the cliques $\mathcal{C}_v \subseteq \mathcal{C}$ that a vertex v is part of. This way, after choosing the remaining vertices of a clique $C \in \mathcal{C}$ as a partition, we have to update the sizes of $\mathcal{O}(\sum_{v \in C} |\mathcal{C}_v|)$ cliques. The total number of such updates throughout the whole algorithm is at most the sum of clique sizes in $\mathcal{C}$. Thus, using a Fibonacci Heap, a total running time of $\mathcal{O}(|V| \log |\mathcal{C}| + \sum_{C \in \mathcal{C}} |C|)$ can be achieved. In our implementation we use a binary heap due to it being faster in practice. This costs an additional factor of $\log |\mathcal{C}|$ for the second term.

Repeated maximal clique heuristic. Note that the MC heuristic does not recompute the maximal cliques of the remaining graph after selecting a clique. As deleting the vertices of one clique can have the effect that a non-maximal clique becomes maximal, the MC heuristic might miss a clique we would want to select. The *repeated maximal clique heuristic* recomputes the set of maximal cliques after each decision, i.e., it selects a maximum clique of the remaining graph in each step.

[1] `https://igraph.org/`

Set Cover heuristics. Observe that for the WEIGHTED CLIQUE PARTITION problem, we have to choose a set of cliques of minimum weight that cover all vertices. Thus, we essentially have to solve a weighted SET COVER problem. As there are reasonably efficient solvers for SET COVER (or the equivalent HITTING SET problem), it seems like a promising approach to use those. However, this has the disadvantage, that we would need to list all cliques and not only the maximal cliques. Nonetheless, it seems like a good heuristic to just consider maximal cliques and find a minimum set cover (unweighted or weighted).

The heuristic consists of two steps. First, we compute a minimum set cover, using the maximum cliques as sets and the vertices as elements. We consider two variants for this steps; weighted (a set of size k has weight $\log(k+1)$) and unweighted (each set has weight 1). Afterwards, in the second step, we convert the cover into a partition by assigning the overlap between selected cliques to only one clique. We call the resulting two approaches the *(maximal clique) set cover* and *(maximal clique) weighted set cover* heuristics.

For the first step, i.e., solving SET COVER, we use a state of the art branch-and-bound solver [6] for the unweighted case. Additionally, for the weighted case, we use the straightforward formulation of set cover as an ILP and solve it with Gurobi [18]. To the best of our knowledge, ILP solvers are currently the state-of-the-art for weighted set cover.

For the second step, we have to compute clique partitions from the resulting set covers by assigning each vertex that is covered by multiple cliques to a single one of these cliques. The goal is to minimize the weight of the resulting cliques, i.e., by Lemma 3, we want to distribute them as unevenly as possible. We employ a simple greedy heuristic, assigning each vertex to the largest clique it is part of and braking ties arbitrarily in case of ambiguity.

At a first glance it seems possible that doing both steps optimally (solving set cover and resolving the overlaps) could yield an overall optimal solution. However, this is not the case, as briefly discussed in Appendix A.

3.3 Exact branch-and-bound solver

Our branch-and-bound branches on which clique to select next. How to branch is described in Section 3.3.1 where we show that we can, in each step, select a maximal clique and that the cliques of the resulting sequence are non-increasing in size. In Section 3.3.2 and Section 3.3.3, we describe lower bounds for pruning the search space, i.e., if the best solution found so far is better than the lower bound in the current branch, we can prune that branch.

3.3.1 Branching

The following structural insight enables us to branch on the maximal cliques.

▶ **Lemma 6.** *Let $\mathcal{P}$ be a minimum weight clique partition of a graph G and let $C \in \mathcal{P}$ be the largest clique of $\mathcal{P}$. Then C is maximal clique in G.*

Thus, even though not all cliques of an optimal solution might be maximal, we at least know that the largest one is. We can use the decision of which maximal clique to select as the largest one as the branching decision of our algorithm. This way, we can solve the optimization variant of WEIGHTED CLIQUE PARTITION, i.e., the algorithm takes a graph G and finds a minimum weight clique partitioning.

After a clique C has been selected as the largest one, the remaining problem is to find a clique partition of $G[V \setminus C]$ that does not use any clique larger than C. This means that we can view our algorithm as a simple recursive subroutine that solves the same problem at every node of the recursion tree. As input it gets the graph G and the cliques $\langle C_1, \ldots, C_i \rangle$

that have already been selected by previous recursive calls. It then tries to compute an optimal clique partition of the remaining graph $G' := G \setminus \bigcup_{j \in [i]} C_j$. This is done by either returning a trivial solution if G' can be covered with a single clique or by branching on the decision of which maximal clique to select as the largest one for the partition of G'. Note that for this decision, only maximal cliques that are at most as large as any of the previously selected cliques $\langle C_1, \ldots, C_i \rangle$ need to be considered. The result of the subroutine call is then the cheapest solution found in any of the branches.

In order to quickly obtain a good upper bound, we explore branches corresponding to larger cliques first. This way, the first leaf of the search tree constructs the same solution as the repeated maximal clique heuristic.

3.3.2 Size lower bound

We call the lower bound given by the following lemma the *size lower bound*.

▶ **Lemma 7.** *Let G be a graph with n vertices and $\mathcal{P}$ be a clique partition of G consisting of cliques of size at most s. Then $\mathcal{P}$ has weight at least $(s+1)^{\lfloor n/s \rfloor} \cdot ((n \mod s) + 1)$.*

Proof. The stated minimum weight is achieved by a partitioning $\mathcal{P}'$ that uses as many cliques of size s as possible and one clique with all remaining vertices. Any other partitioning $\mathcal{P}$ using only cliques of size at most s is at least as expensive, as it can be transformed into $\mathcal{P}'$ by Lemma 4. ◀

Note that the size lower bound can trivially be evaluated in constant time. Even though it is rather basic, we expect this lower bound to be effective at pruning branches in which very small cliques are selected early on.

3.3.3 Valuable sequence lower bound

Note that the size lower bound optimistically assumes that there are $\lfloor n/s \rfloor$ non-overlapping cliques of size s. This yields a bad lower bound if, e.g., there is only one clique of size s while all other cliques are much smaller. In the following, we describe an improved bound based on this observation. We note that we have to be careful when considering what clique sizes are available for the following reason. Assume the branching has already picked a clique of size s, i.e., subsequent selected cliques have to have size at most s. Then it seems natural to derive a lower bound by summing over the sizes of all maximal cliques of size at most s. However, we have to account for the fact that selecting (and deleting) one clique can shrink a maximal clique that was larger than s to become a clique of size s. Thus, there might me more cliques of size s available than initially thought. In order to formalize this, we first introduce a different problem that considers only sizes of the cliques without making any assumptions on the overlap between the cliques.

In the Valuable Sequence problem, we are given a multiset A of natural numbers and a natural number n. The task is to construct a sequence of total value n and minimum weight. Such a sequence $S = \langle s_1, s_2, \ldots, s_k \rangle$ consists of elements $s_i \in A$ such that each number is repeated at most as often as it appears in A. In the following, we define value and weight of a sequence and give additional restrictions to what constitutes a valid sequence. To this end, let $S_i = \langle s_1, \ldots, s_i \rangle$ for $i \leq k$ denote a prefix of S. We define a value $\text{val}(s_i)$ for each s_i in the sequence as follows. The first element s_1 has value $\text{val}(s_1) = s_1$. For subsequent elements s_{i+1}, we have $\text{val}(s_{i+1}) = \min\{s_{i+1}, \text{val}(s_i), n - \text{val}(S_i)\}$, where $\text{val}(S_i) = \sum_{j \in [i]} \text{val}(s_j)$ is

the total value of the prefix S_i.[2] If $\mathrm{val}(s_i) = s_i$, we say that the element contributes *fully* to the sequence. Otherwise, it contributes *partially*. The *weight* of S is $\prod_{i \in [k]}(\mathrm{val}(s_i) + 1)$. For the subsequence S_i, we call the next element s_{i+1} *eligible* if $s_{i+1} - \mathrm{val}(S_i) \leq \mathrm{val}(s_i)$; s_1 is always eligible. The sequence S is *valid* if each element is eligible.

To make the connection back to Weighted Clique Partition, interpret the numbers in A as the clique sizes. The total value n corresponds to the number of vertices that have to be covered. The value $\mathrm{val}(s_i)$ corresponds to the number of vertices from the maximal clique of size s_i in G that have not been covered by previous cliques, i.e., the number of vertices that are newly covered in step i. Note that in step $i+1$, at least $s_i - \mathrm{val}(S_i)$ new vertices are covered as only $\mathrm{val}(S_i)$ have been covered previously. Thus, the eligibility requirement ensures that the number of vertices covered in step $i+1$ is not larger than the number of vertices covered in step i (recall, that we can assume the chosen cliques to form a non-increasing sequence). Moreover, for the definition of $\mathrm{val}(s_{i+1})$, note that the minimum with $\mathrm{val}(s_i)$ ensures that the sequence of values is non-increasing and the minimum with $n - \mathrm{val}(S_i)$ ensures that the total value is n.

The following two lemmas formalize this connection between Valuable Sequence and Weighted Clique Partition. Afterwards, we discuss how Valuable Sequence can be solved optimally.

▶ **Lemma 8.** *Let $\mathcal{P}$ be a minimum weight clique partition of a graph G and let $\mathcal{C}$ be the set of maximal cliques in G. Then, any mapping $f : \mathcal{P} \to \mathcal{C}$ with $P \subseteq f(P)$ for each $P \in \mathcal{P}$ is injective and there exists at least one such mapping.*

Proof. There are mappings from $\mathcal{P}$ to $\mathcal{C}$, because each clique $P \in \mathcal{P}$ is either a maximal clique or a subset of a larger maximal clique. Assume that a mapping $f : \mathcal{P} \mapsto \mathcal{C}$ is not injective. Then, there are two partition classes P and P' that are mapped to the same clique $C \in \mathcal{C}$. Thus, these partition classes could be merged, contradicting the optimality of $\mathcal{P}$. ◀

▶ **Lemma 9.** *Let G be a graph with maximal cliques $\mathcal{C} = \{C_1, \dots, C_k\}$ and let (A, n) with $A = \{|C_1|, \dots, |C_k|\}$ and $n = |V(G)|$ be an instance of Valuable Sequence. The weight of a minimum solution of (A, n) is a lower bound for the weight of every clique partition of G.*

Proof. We now show that for a minimum clique partition $\mathcal{P}$ of G, we find a solution S of (A, n) whose weight is at most the weight of $\mathcal{P}$.

Let $P_1, \dots, P_{k'}$ be the cliques of $\mathcal{P}$ sorted by size in decreasing order. We can think of $\mathcal{P}$ as constructed iteratively in that order, so that each P_i is a maximal clique in $G[V \setminus (\bigcup_{j \in [i-1]} P_j)]$.

We construct S iteratively until $\mathrm{val}(S) = n$. For each element s_i in the sequence, we prove by induction that $\mathrm{val}(s_i) \geq |P_i|$ except for the last element. This then lets us use Lemma 4 to obtain that the weight of S is at most the weight of $\mathcal{P}$. For $i = 1$, we simply choose $s_1 = |P_1| \in A$. Since the first element always contributes fully, we have $\mathrm{val}(s_1) = |P_1|$.

Assuming we constructed the sequence until i, we continue with step $i+1$ as follows. If P_{i+1} is one of the initial maximal cliques in $\mathcal{C}$, then we can simply choose $s_{i+1} = |P_{i+1}| \in A$. Note that s_{i+1} is eligible, as $s_{i+1} = |P_{i+1}| \leq |P_i| \leq \mathrm{val}(s_i)$, which in particular implies $s_{i+1} - \mathrm{val}(S_i) \leq \mathrm{val}(s_i)$. In this case, s_{i+1} contributes fully, i.e., $\mathrm{val}(s_{i+1}) = |P_{i+1}|$, which implies the claim.

Otherwise, if P_{i+1} is not in $\mathcal{C}$, it is at least a subset of some clique $C \in \mathcal{C}$ such that $P_{i+1} = C \setminus \bigcup_{j \in [i]} P_j$. We choose $s_{i+1} = |C|$. The eligibility of s_{i+1} follows from the facts that the cliques in $\mathcal{P}$ are ordered non-increasingly, i.e. $|P_i| \geq |P_{i+1}|$, and that $\mathrm{val}(s_j) \geq |P_j|$ holds by induction for all $j < i+1$:

[2] Note that $\mathrm{val}(s_{i+1})$ only depends on values of previous elements in S, i.e., the definition is not cyclic.

$$\mathrm{val}(s_i) \geq |P_i| \geq |P_{i+1}| = \left| C \setminus \bigcup_{j \in [i]} P_j \right| \geq |C| - \sum_{j \in [i]} |P_j| \geq s_{i+1} - \sum_{j \in [i]} \mathrm{val}(s_j) = s_{i+1} - \mathrm{val}(S_i).$$

Note that s_{i+1} contributes partially, i.e., $\mathrm{val}(s_{i+1}) = \mathrm{val}(s_i)$ unless this is the last item in S. As we just argued, we have $\mathrm{val}(s_i) \geq |P_{i+1}|$ and thus $\mathrm{val}(s_{i+1}) \geq |P_{i+1}|$, proving the claim.

To conclude, observe that our construction of S implicitly defines a mapping from $\mathcal{P}$ to $\mathcal{C}$ as in Lemma 8. As such a mapping is injective, no number in A is chosen twice. Moreover as we have $\mathrm{val}(s_i) \geq |P_i|$ for $i < k$, but both sum to n, the weight of $\mathcal{P}$ is at least the weight of S by Lemma 4. ◀

Valuable Sequence can be solved optimally with a simple greedy algorithm. We call the resulting lower bound the *valuable sequence* bound.

▶ **Theorem 10.** *An instance (A, n) of Valuable Sequence can be solved in $O(|A| + n)$ time.*

3.3.4 Sufficient weight reduction

To speed-up the computation of clique partitions for all bags of a tree decomposition, we additionally apply the following reduction rule. In the *sufficient weight* reduction, we immediately accept the first solution that is lighter or equally light as the largest weight of any of the already considered bags.

4 Evaluation

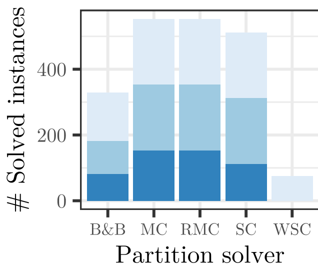
With our evaluation, we aim to answer the following questions.

1. How do the different algorithms compare in regards to run time and quality?
2. How do the algorithms scale?
3. What is the impact of the lower bounds and the reduction rules on the performance of the exact branch-and-bound solver?
4. How do different network properties influence the performance of the algorithms?
5. How do the resulting upper bounds on the clique-partitioned treewidth compare to traditional treewidth?

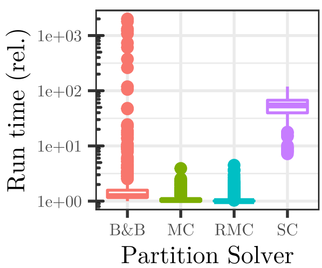
Experimental setup. Our implementation is written in Python. The source code along with all evaluation scripts and results is available on our public GitHub repository[3]. The experiments were run with Python 3.10.1 on a Gigabyte R282-Z93 (rev. 100) server (2250MHz) with 1024GB DDR4 (3200MHz) memory.

For each input graph, we perform the following two steps. First, we compute a tree decomposition using the heuristics implemented in the HTD library [1]. Specifically, we use the min-fill-in heuristic, which is known to provide a good tradeoff between run time and solution quality [23]. Secondly, we solve the Weighted Clique Partition problem for each bag of the tree decomposition using all algorithms proposed in Section 3.

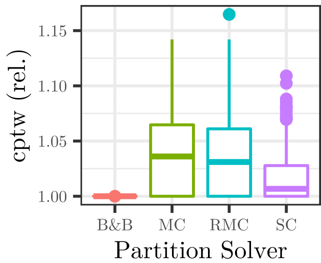
[3] `https://github.com/marcwil/cptw_code`



(a) Number of solved instances with 500 (bright) 5 k (medium) and 50 k (dark) vertices.



(b) Distribution of run time relative to fastest solver within time limit on a given graph.



(c) Distribution of obtained width relative to best found solution on a given graph.

Figure 1 Comparison of run time and solution quality of the different exact (red), greedy (green, blue) and set cover based (violet) solvers for the Weighted Clique Partition problem on GIRGs.

We use a time limit of five minutes for the heuristic computation of low-weight tree decompositions with the HTD library. For the Weighted Clique Partition algorithms, we set a time limit of three minutes per bag and five minutes in total.

To discern the different solvers from Sections 3.2 and 3.3, in our plots, we use the following abbreviations: branch and bound solver (B&B), maximal clique set cover heuristic (SC), maximal clique weighted set cover heuristic (WSC), maximal clique heuristic (MC), and repeated maximal clique heuristic (RMC).

Input instances. For the input, we use a large collection of real-world networks as well as generated networks. For the latter, we use geometric inhomogeneous random graphs (GIRGs) [8], which resemble real-world networks in regards to important properties and have been shown to be well suited for the evaluation of algorithms [3]. GIRGs can be generated efficiently [4] and allow to vary the power-law exponent (ple) of the degree distribution controlling its heterogeneity, as well as a parameter α controlling the locality by either strengthening the influence of the geometry (high values of α) or increasing the probability for random edges not based on the geometry (low values of α). We mainly use the following two datasets, where each graph has been reduced to its largest connected component.

- A collection of 2967 real-world networks [7] that essentially consists of all networks with at most 1 M edges from Network Repository [25]; see [3] for details.
- GIRGs with $n \in \{500, 5000, 50000\}$ vertices, expected average degree 10, dimension 1, ple $\in \{2.1, 2.3, 2.5, 2.7, 2.9\}$, and $\alpha \in \{1.25, 2.5, 5, \infty\}$. For each parameter configuration, we generate ten networks with different random seeds, to smooth out random variations.

4.1 Performance comparison

Here, we evaluate the performance of our Clique Partition approaches on the two datasets.

Generated instances. In Figure 1, we compare the run times as well as the solution quality of the different considered Clique Partition algorithms on the dataset of generated networks. In Figure 1a, we show how many of the 600 instances were solved within the time limit by each solver. While the greedy heuristics are able to finish on almost all instances, the set cover heuristic and the branch-and-bound solver get timed on some of the larger networks with 5 k and 50 k vertices. The weighted set cover heuristic performs much worse, finishing only on few networks. We therefore exclude it from the other comparisons.

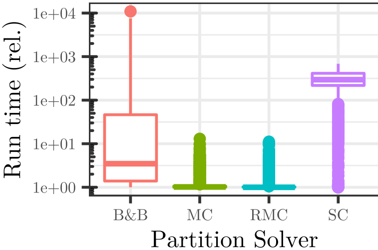


Figure 2 Distribution of run time relative to fastest solver within time limit on our set of real-world networks.

Table 1 Distribution of obtained cp-treewidth relative to optimum clique partition on our set of real-world networks.

Measure	MC	RMC	SC
Mean	1.008	1.009	1.002
Median	1	1	1
90th percentile	1.035	1.036	1.000
99th percentile	1.108	1.121	1.046
Maximum	1.254	1.192	1.113

In Figures 1b and 1c, we compare the performance for all instances that were solved within the time limit by all other algorithms. Figure 1b shows the run time of each algorithm relative to the fastest one on each instance. Figure 1c shows the obtained upper bound on the cp-treewidth relative to the optimal solution computed by the branch-and-bound solver.

Our findings are as follows. The branch-and-bound solver solves the fewest instances of the four considered algorithms, but is quick on most of the instances it is able to solve within the time limit. Both greedy heuristics (MC and RMC) are similarly fast, significantly outcompeting the other approaches. In terms of quality, all three heuristics perform well, achieving solutions within few percent of the optimum. The set cover heuristic slightly outperforms the greedy heuristics in terms of quality, but pays for this with substantially higher running time.

Real-world networks. We complement the above evaluation of our Clique Partition algorithms, by comparing their performance on the collection of real-world networks. As above, we exclude the weighted set cover heuristic. The other four approaches were able to finish on 1243 (B&B), 2619 (MC), 2622 (RMC), and 2204 (SC) of the 2967 networks within the time limit. We compare our algorithms on the 1237 networks that were solved by all four approaches. Figure 2 shows the run time of each solver relative to the fastest solver on each instance. In Table 1 we describe the distribution of the obtained upper bounds on the cp-treewidth relative to the optimal solution found by the branch-and-bound solver.

Our results are the following. In general, our observations on generated networks are replicated on the real-world networks. Even though the branch-and-bound algorithm solved fewer instances than the set cover heuristic, it is comparatively faster on the networks it is able to solve. Both approaches are, however, considerably slower than the greedy heuristics and this difference is more pronounced than on the generated networks. Regarding the solution quality, all three heuristic solvers perform even better than on the generated networks, with only a tiny fraction of instances not being solved almost optimally.

Discussion. We find that the proposed algorithms show good performance both on generated and real-world instances. Although, the branch-and-bound solver was only able to solve about half of the considered networks, its run time typically beats the set cover heuristic on the networks it can solve. In addition, it is a valuable tool for evaluating the solution quality of the other approaches. We find that especially the set cover heuristic, but also the greedy heuristics (MC and RMC) often find close to optimal clique partitions. Due to their excellent trade-off between speed and solution quality, the greedy heuristics are probably the best approach in most practical settings. In general, we do not expect that there is substantial

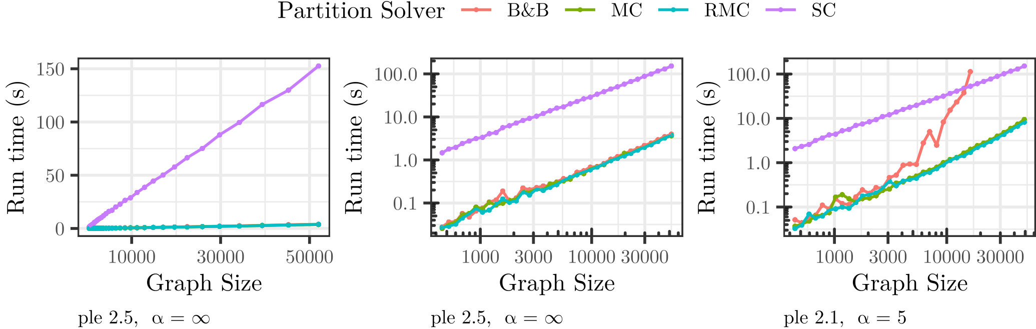


Figure 3 Scaling behavior of Clique Partition algorithms on GIRGs with different parameters.

room for improvement in the engineering of Clique Partition solvers for the computation of cp-treewidth. Instead, in order to achieve better upper bounds, we suggest future research to optimize the tree decomposition and the partition into cliques at the same time.

4.2 Run time scaling

Next, we consider the scaling behavior of our solvers. For this, we generated GIRGs of varying sizes up to around 50 k vertices for various parameters. As in Section 4.1, we did not evaluate the weighted set cover heuristic. Figure 3 shows the run times for GIRGs with two different parameter configurations. On the networks with high locality ($\alpha = \text{inf}$), all four approaches seem to have close to linear run time, despite enumerating all maximal cliques present in each bag of the tree decomposition. However, as we decrease the locality ($\alpha = 5$) the performance of the branch-and-bound solver deteriorates while the greedy heuristics and especially the set cover heuristic are only slightly affected. In the logarithmic plot, we observe clearly super-polynomial scaling behavior only for the branch-and-bound solver. Further experiments on a larger grid of parameter settings confirm the above findings.

4.3 Branch-and-bound: lower bounds and reduction rule

In the following, we evaluate the effectiveness of the lower bounds and the reduction rule in speeding up our branch-and-bound solver. For this, we use the dataset of generated networks. As the performance without lower bounds does not allow for the timely evaluation on larger instances, we consider only graphs generated with 5 k vertices. Figure 4 shows the average run time without lower bounds (none), with only the size lower bound (S) and with the valuable sequence bound in addition to the size bound (S+V) as well as with and without the sufficient weight reduction for different network parameters. We only show $\alpha \in \{5, \infty\}$, as for lower values the variant without lower bounds did not finish within the time limit.

We find that especially for smaller power-law exponents, the lower bounds bring large speed-ups of up to multiple orders of magnitude. The additional gain of using the size lower bound is much larger than that of the much simpler valuable sequence bound. The sufficient weight reduction yields similar speed-ups for all settings. Overall, we conclude that the lower bounds are effective in speeding up the branch-and-bound solver. On a more general note, it is striking how strongly all variants of the solver are affected by lower values of α, especially also below the values shown in Figure 4. In additional experiments we found that the above observations also apply to the remainder of the dataset, even though for 50 k vertices the time limit is reached even more frequently.

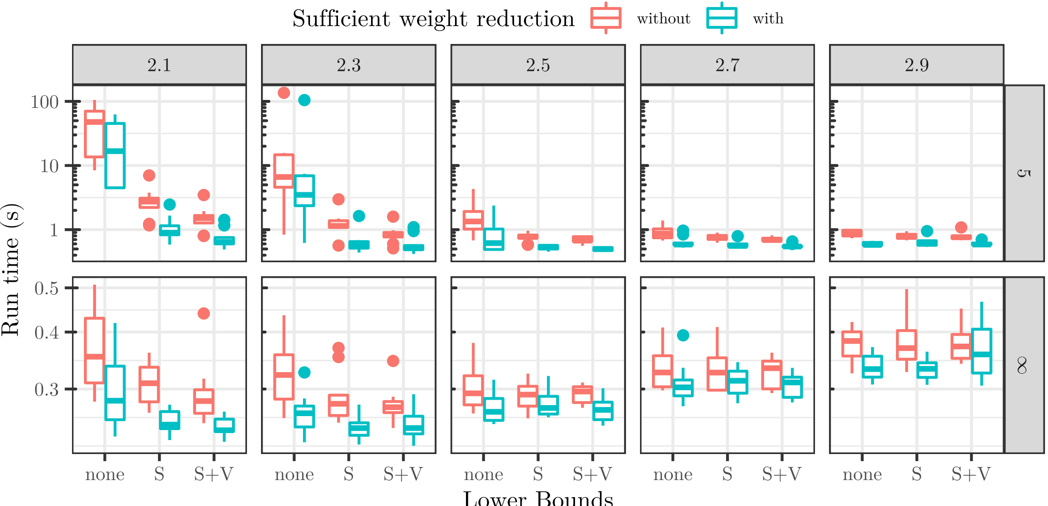


Figure 4 Run time of different variants of the branch-and-bound solver on GIRGs with 5 k vertices and different values for the power-law exponent (left to right) and α (top / bottom).

4.4 Impact of network properties

At multiple points throughout the last sections, we found that, especially for the branch-and-bound algorithm, the performance strongly depended on the parameter α controlling the locality of the generated networks.

In order to better understand this, we study the structure of cliques in the generated networks depending on their parameters. Specifically, for each network we count the number of maximal cliques in the graph, we count the number of maximal cliques in each bag of the tree decomposition and take the maximum, we count the number of cliques used per bag in the clique-partitioned tree decomposition and take the maximum, and consider the width of the clique-partitioned tree decomposition. The clique-partitioned tree decompositions are obtained using the MC and MCR heuristic. Figure 5 shows these values for GIRGs with varying power-law exponent and α.

We see that with decreasing values of α, all considered measures increase. However, while the total number of maximal cliques in the network only increases by a factor of roughly 4 to 10, the highest number of cliques intersecting some bag of the tree decomposition as well as the highest number of cliques in a lowest-weight clique partition increase by multiple orders of magnitude. Intuitively, this can be explained by cliques starting to fray if the locality is too low. This explains, why the Clique Partition problem is harder on GIRGs with lower values of α, which slows down the branch-and-bound algorithm. We also observe, that the obtained upper bounds on the cp-treewidth are not much lower than the highest number of cliques per bag of a solution, explaining the good performance of the set cover heuristic.

4.5 Clique-partitioned treewidth compared to traditional treewidth

Here we consider the data set of real-world networks. As we have seen in Section 4.1, the maximal clique and repeated maximal clique heuristics are efficient and tend to perform well in terms of quality. Thus, we use these two heuristics to find an upper bound on the clique-partitioned treewidth.

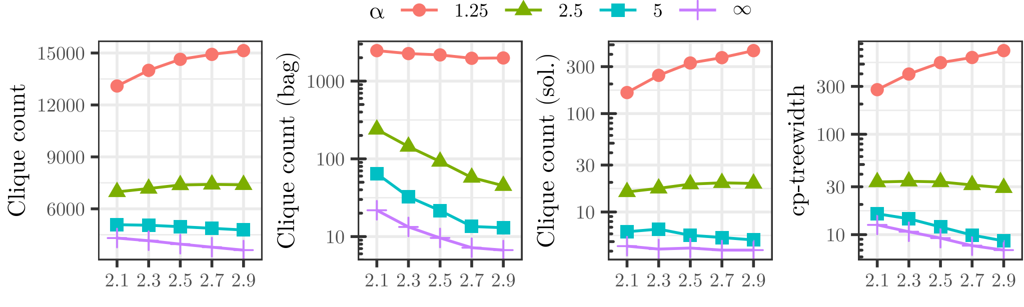
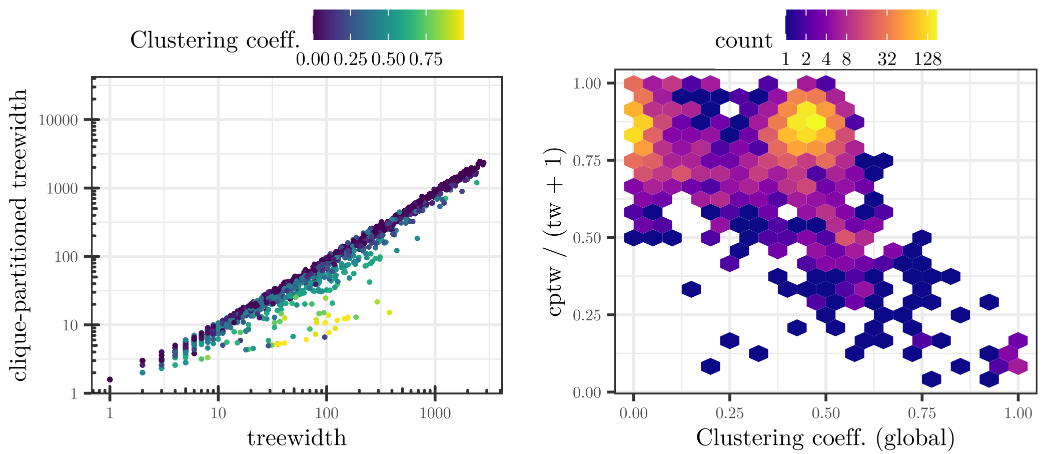


Figure 5 Total clique count (number of maximal cliques) per network, and highest clique count in any bag of a greedy tree decomposition as well in the lowest weight clique partition of any bag, and clique-partitioned treewidth (lowest upper bound) of the entire instance on GIRGs with 5 k vertices and varying parameters. Note the logarithmic y-axes on all except the first plot.



(a) Dependency between clustering coefficient and heuristic upper bounds on clique-partitioned treewidth and treewidth.

(b) Dependency between clustering coefficient and relative difference between clique-partitioned treewidth and treewidth.

Figure 6 Upper bounds for clique-partitioned treewidth on large real-world networks.

In Figure 6a we compare the obtained upper bounds for the weighted treewidth and the treewidth. Even though the parameter does not decrease much for the majority networks, there are some networks on which substantial reductions are achieved. This is particularly true for networks with high clustering coefficient, where for some instances our clique-partitioned tree decomposition has width 10 while the corresponding traditional tree decomposition has width above 100. This correspondence with the clustering coefficient fits well to the observations in Section 4.4. For the networks for which we do not yet see a big improvement, it would be interesting to see whether adjusting the computation of the initial tree decomposition can yield better bounds; see also the discussion in Section 4.1.

References

1. Michael Abseher, Nysret Musliu, and Stefan Woltran. htd - A free, open-source framework for (customized) tree decompositions and beyond. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, pages 376–386. Springer, 2017. doi:10.1007/978-3-319-59776-8_30.
2. Chris Aronis. The algorithmic complexity of tree-clique width. *CoRR*, abs/2111.02200, 2021. arXiv:2111.02200.
3. Thomas Bläsius and Philipp Fischbeck. On the external validity of average-case analyses of graph algorithms. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022, September 5-9, 2022, Berlin/Potsdam, Germany*, volume 244 of *LIPIcs*, pages 21:1–21:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ESA.2022.21.
4. Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. Efficiently generating geometric inhomogeneous and hyperbolic random graphs. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 21:1–21:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.21.
5. Thomas Bläsius, Tobias Friedrich, and Anton Krohmer. Hyperbolic random graphs: Separators and treewidth. In Piotr Sankowski and Christos D. Zaroliagis, editors, *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, volume 57 of *LIPIcs*, pages 15:1–15:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ESA.2016.15.
6. Thomas Bläsius, Tobias Friedrich, David Stangl, and Christopher Weyand. An efficient branch-and-bound solver for hitting set. In Cynthia A. Phillips and Bettina Speckmann, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*, pages 209–220. SIAM, 2022. doi:10.1137/1.9781611977042.17.
7. Thomas Bläsius and Philipp Fischbeck. 3006 Networks (unweighted, undirected, simple, connected) from Network Repository, May 2022. doi:10.5281/zenodo.6586185.
8. Karl Bringmann, Ralph Keusch, and Johannes Lengler. Geometric inhomogeneous random graphs. *Theor. Comput. Sci.*, 760:35–54, 2019. doi:10.1016/j.tcs.2018.08.014.
9. Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Treewidth*, pages 151–244. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-21275-3_7.
10. Clément Dallard, Fedor V. Fomin, Petr A. Golovach, Tuukka Korhonen, and Martin Milanic. Computing tree decompositions with small independence number. *CoRR*, abs/2207.09993, 2022. doi:10.48550/arXiv.2207.09993.
11. Clément Dallard, Martin Milanic, and Kenny Storgel. Treewidth versus clique number. i. graph classes with a forbidden structure. *SIAM J. Discret. Math.*, 35(4):2618–2646, 2021. doi:10.1137/20M1352119.
12. Clément Dallard, Martin Milanic, and Kenny Storgel. Treewidth versus clique number. III. tree-independence number of graphs with a forbidden structure. *CoRR*, abs/2206.15092, 2022. doi:10.48550/arXiv.2206.15092.
13. Clément Dallard, Martin Milanič, and Kenny Štorgel. Treewidth versus clique number. ii. tree-independence number, 2021. doi:10.48550/arXiv.2111.04543.
14. Mark de Berg, Hans L. Bodlaender, Sándor Kisfaludi-Bak, Dániel Marx, and Tom C. van der Zanden. A framework for eth-tight algorithms and lower bounds in geometric intersection graphs. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 574–586. ACM, 2018. doi:10.1145/3188745.3188854.

15 Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The First Parameterized Algorithms and Computational Experiments Challenge. In Jiong Guo and Danny Hermelin, editors, *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:9, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.IPEC.2016.30`.

16 Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In Daniel Lokshtanov and Naomi Nishimura, editors, *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, volume 89 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.IPEC.2017.30`.

17 David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM J. Exp. Algorithmics*, 18, 2013. `doi:10.1145/2543629`.

18 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL: `https://www.gurobi.com`.

19 David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988. `doi:10.1016/0020-0190(88)90065-8`.

20 Sándor Kisfaludi-Bak. Hyperbolic intersection graphs and (quasi)-polynomial time. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1621–1638. SIAM, 2020. `doi:10.1137/1.9781611975994.100`.

21 Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Phys. Rev. E*, 82:036106, September 2010. `doi:10.1103/PhysRevE.82.036106`.

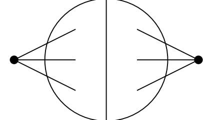
22 László Lovász. Graph minor theory. *Bulletin of the American Mathematical Society*, 43(1):75–86, October 2005. `doi:10.1090/S0273-0979-05-01088-8`.

23 Silviu Maniu, Pierre Senellart, and Suraj Jog. An experimental study of the treewidth of real-world graph data. In Pablo Barceló and Marco Calautti, editors, *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, volume 127 of *LIPIcs*, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ICDT.2019.12`.

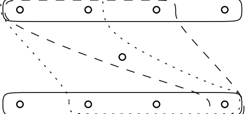
24 Neil Robertson and Paul D. Seymour. Graph minors. IV. tree-width and well-quasi-ordering. *J. Comb. Theory, Ser. B*, 48(2):227–254, 1990. `doi:10.1016/0095-8956(90)90120-O`.

25 Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 4292–4293. AAAI Press, 2015. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9553`.

A Limits of the set cover heuristics



(a) Unweighted set cover.



(b) Weighted set cover.

Figure 7 Counter-examples for the optimality of the set cover heuristics.

We want to briefly discuss why the set cover solutions are not always optimal clique partitions. First, we give an instance on which the unweighted set cover approach fails.

▶ **Observation 11.** *There are graphs on which the minimum size clique cover cannot give an optimal clique partition.*

Proof. Consider a clique on k vertices for even k where half of the vertices are connected to one additional vertex and the other half to another additional vertex, as illustrated in Figure 7a. Then, for $k \geq 6$ the partition into three cliques of sizes 1, 1, and k has lower weight than the partition into two cliques of size $\frac{k}{2} + 1$, which corresponds to the optimal solution of the set cover instance. ◀

For the minimum weight set cover, we can use the fact that the weights of the set cover instance correspond to the size of the whole clique and do not reflect the potential overlap between multiple selected cliques.

▶ **Observation 12.** *There are graphs on which the minimum size clique cover cannot give an optimal clique partition.*

Proof. For the weighted approach, consider the instance depicted in Figure 7b. The small circles represent the vertices of a graph and the regions mark maximal cliques. The optimal clique partitioning uses cliques of sizes 6, 2, and 1 (the dotted clique plus the remainders of the two solid cliques). In the set cover instance these cliques have (partly overlapping) sizes 6, 4, and 4, which is more expensive than the set cover solution with sizes 5, 4, and 4 (using the dashed clique instead of the dotted one), which results in a solution with sizes $5, 3, 1$. ◀

The above problem could be avoided by extending the set cover instance to also include all non-maximal subsets of each clique that can be obtained by removing vertices shared with some subset of overlapping cliques. This would, however, lead to an exponential blowup of the set cover instances, which is not feasible even with state of the art solvers.

B Omitted proofs

▶ **Lemma 2.** *Let G be a graph with a clique-partitioned tree decomposition (T, σ) of weight τ. Then a largest independent set of G can be found in $O(2^\tau \cdot \mathrm{poly}(n))$ time.*

Proof. We use a standard dynamic programming approach on tree decompositions based on *introduce*, *forget*, and *join* nodes (see for example Cygan et. al [9]). For each node $t \in V(T)$, we store a number of partial solutions for the subgraph of G induced by the bags of nodes in the subtree below t.

A partial solution consists of a subset of the vertices in the current bag as well as the size of the total partial independent set for the subgraph induced by the subtree below the current bag. This makes it easy to initialize partial solutions for leaf nodes in the tree decomposition.

In an introduce node, two new partial solutions are created, one where the new vertex is in the independent set and one where it is not. In a forget node, the removed vertex is removed from each partial solution. In a join node, the partial solutions from the child-nodes are combined by taking their union.

In a traditional tree decomposition of width k, this leads to at most 2^k partial solutions per bag. In a clique-partitioned tree decomposition, this is even smaller, as there are only $k+1$ ways an independent set can intersect a clique of size k. Thus, assuming $\{\mathcal{P}_t \mid t \in V(T)\}$ denotes the clique partition of weight τ, the number of partial solutions that need to be considered per bag t are at most

$$\prod_{C \in \mathcal{P}_t} (|C| + 1) = 2^{\sum_{C \in \mathcal{P}_t} \log(|C|+1)} = 2^{\tau}.$$

As the number of bags and time spent per bag is polynomial, this concludes the proof. ◀

▶ **Lemma 3.** *Let $a, b, c, d \in \mathbb{N}_0$ such that $a + b = c + d$ and $a \geq b$, $c \geq d$, $d > b$. Then $(a + 1)(b + 1) < (c + 1)(d + 1)$.*

Proof. There is an $x > 0$ such that $c = a - x$ and $d = b + x$. As $c \geq d$, x can be at most $(a - b)/2$. We derive

$$\begin{aligned}(c + 1)(d + 1) &= (a - x + 1)(b + x + 1) \\ &= ab - bx + b + ax - x^2 + x + a - x + 1 \\ &= (ab + a + b + 1) + ax - bx - x^2 \\ &= (a + 1)(b + 1) + x(a - b - x).\end{aligned}$$

We have $x(a - b - x) > 0$, as $0 < x \leq \frac{a-b}{2}$ and thus the claimed strict inequality follows. ◀

▶ **Lemma 6.** *Let $\mathcal{P}$ be a minimum weight clique partition of a graph G and let $C \in \mathcal{P}$ be the largest clique of $\mathcal{P}$. Then C is maximal clique in G.*

Proof. Assume that C is a non-maximal clique. That is, there is a vertex $v \in V(G) \setminus C$ with $C \subseteq N(v)$. Let $C' \in \mathcal{P}$ be the clique containing v. We construct a clique partition $\mathcal{P}'$ by removing v from C' and adding it to C. As C was the largest clique in $\mathcal{P}$, via Lemma 3 we have $(|C| + 2)(|C'|) < (|C| + 1)(|C'| + 1)$, contradicting the optimality of $\mathcal{P}$. ◀

▶ **Theorem 10.** *An instance (A, n) of VALUABLE SEQUENCE can be solved in $O(|A| + n)$ time.*

Proof. We construct a solution $S = s_1, \ldots, s_i$ by iteratively choosing an eligible and not yet chosen number $a \in A$ with maximum value, until the value of the sum reaches n.

We note that this greedy strategy maximizes how many numbers in A are eligible, as the corresponding upper bound $\text{val}(S) + \text{val}(s_i)$ decreases as slowly as possible. The optimality of the produced sequence $S = s_1, \ldots, s_i$ follows, again, via Lemma 4 as for $j \in [i]$, the value $\text{val}(s_j)$ is at least as large as the value of any other number that can be chosen in round j.

Regarding the running time, the greedy strategy can be implemented by sorting the numbers in A (in $O(|A| + n)$ time) and keeping track of the largest unchosen number that is eligible and contributes fully, as well as the smallest unchosen number that can contribute partially (which is larger than the ones that can contribute fully). Both of these values can be updated in constant time each time a number has been chosen. ◀

Subset Wavelet Trees

Jarno N. Alanko
Helsinki Institute for Information Technology (HIIT), Finland
Department of Computer Science, University of Helsinki, Finland

Elena Biagi
Department of Computer Science, University of Helsinki, Finland

Simon J. Puglisi
Helsinki Institute for Information Technology (HIIT), Finland
Department of Computer Science, University of Helsinki, Finland

Jaakko Vuohtoniemi
Department of Computer Science, University of Helsinki, Finland

Abstract

Given an alphabet Σ of $\sigma = |\Sigma|$ symbols, a degenerate (or indeterminate) string X is a sequence $X = X[0], X[1] \ldots, X[n-1]$ of n subsets of Σ. Since their introduction in the mid 70s, degenerate strings have been widely studied, with applications driven by their being a natural model for sequences in which there is a degree of uncertainty about the precise symbol at a given position, such as those arising in genomics and proteomics. In this paper we introduce a new data structural tool for degenerate strings, called the subset wavelet tree (SubsetWT). A SubsetWT supports two basic operations on degenerate strings: subset-rank(i, c), which returns the number of subsets up to the i-th subset in the degenerate string that contain the symbol c; and subset-select(i, c), which returns the index in the degenerate string of the i-th subset that contains symbol c. These queries are analogs of rank and select queries that have been widely studied for ordinary strings. Via experiments in a real genomics application in which degenerate strings are fundamental, we show that subset wavelet trees are practical data structures, and in particular offer an attractive space-time tradeoff. Along the way we investigate data structures for supporting (normal) rank queries on base-4 and base-3 sequences, which may be of independent interest. Our C++ implementations of the data structures are available at `https://github.com/jnalanko/SubsetWT`.

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

Keywords and phrases degenerate strings, compressed data structures, succinct data structures, string processing, data structures, efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.4

Supplementary Material *Software (Source Code)*: `https://github.com/jnalanko/SubsetWT`
Software (Source Code): `https://github.com/jnalanko/SubsetWT-Experiments`

Funding This work was supported in part by the Academy of Finland via grants 339070 and 351150.

Acknowledgements A brief description of the subset wavelet tree first appeared in a technical report by the authors [3].

1 Introduction

Given an alphabet Σ of σ symbols, a degenerate (or indeterminate) string is a sequence $X = X[0], X[1] \ldots, X[n-1]$ of subsets of Σ. For example, here is a degenerate string of length 15 on the alphabet $\Sigma = A, C, G, T$ (note that empty subsets are allowed):

$$X = \{T\}\{G\}\{A, C, G, T\}\{\}\{\}\{C, G\}\{\}\{A\}\{\}\{A\}\{A, C\}\{\}\{\}\{A\}\{A\}.$$

© Jarno N. Alanko, Elena Biagi, Simon J. Puglisi, and Jaakko Vuohtoniemi;
licensed under Creative Commons License CC-BY 4.0
21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis; Article No. 4; pp. 4:1–4:14
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Since their introduction in a classic paper by Fischer and Paterson [10], degenerate strings have been widely studied in the field of string processing and its application domains. Abrahamson [1] studied online pattern matching for degenerate strings, describing theoretically optimal algorithms. Since then, several authors (see, e.g., [16, 30]) have described practical pattern matching algorithms that are fast in practice. Authors have also considered covering problems [7], data structures for extension queries [17], constrained LCS [18], and the computation of inverted repeats [2] on degenerate strings.

Interest in degenerate strings has been driven by them being a natural model of the uncertainty or flexibility often present in real world sequence data. For example, the IUPAC encoding for biological sequences [19] designates specific symbols, referred to as degenerate, to represent a sequence position corresponding to a set of possible alternative nucleotides. In music sequences, single notes may match chords, or notes separated by an octave may match – properties naturally captured by degenerate strings [5].

In this paper we study two new and seemingly fundamental operations on indeterminate strings - subset rank and select. In particular, for a given degenerate string X, we define (for $i \le n$ and $c \in \Sigma$):

$$\mathsf{subset\text{-}rank}_X(i,c) = \text{number of subsets among the first } i \text{ subsets of } X \text{ that contain } c;$$
$$\mathsf{subset\text{-}select}_X(i,c) = \text{index in } X \text{ of the } i\text{th subset that contains } c.$$

For example, if $X = \{T\}\{G\}\{A,C,G,T\}\{\}\{\}\{C,G\}\{\}\{A\}\{\}\{A\}\{A,C\}\{\}\{\}\{A\}\{A\}$ as before, then we would have $\mathsf{subset\text{-}rank}_X(8,A) = 2$.

Rank and select queries on ordinary (i.e. non-degenerate) strings are now considered fundamental to the field of succinct and compressed data structures [26, 23]. To our knowledge, however, the literature on degenerate and indeterminate strings has not explicitly considered these queries before.

Our own interest in supporting these types of queries on degenerate strings comes from problems in pangenomics, and in particular the spectral Burrows-Wheeler transform (SBWT), a recently described approach for representing the de Bruijn graph of a set of strings [3]. The de Bruijn graph is a central data structure in computational biology, used for a variety of tasks, including genome assembly [6] and pangenomic read alignment [15, 21]. There is exactly one node in the de Bruijn graph for every distinct k-length substring, or k-mer occurring in the set of input strings, and nodes are labelled with these substrings. A k-mer query on the de Bruijn graph asks if there is a node in the graph labelled with a specified query k-mer. In [3] it is shown that these queries can be reduced to a sequence of $2k$ subset-rank queries on a particular degenerate string L produced by the SBWT that encodes the graph. For brevity, we avoid defining the SBWT here, but we note that L has a special property that its length is also equal to the sum of the sizes of the sets, i.e. $|L| = \sum_i |L[i]|$. The allowance of empty sets means this is possible without the resulting sequence becoming an ordinary string. We call such a degenerate string *balanced*. We return to this special case later, but note here that the data structure we describe applies to all degenerate strings, balanced or not.

Contribution. We describe the subset wavelet tree, a new data structure for subset-rank and subset-select queries on degenerate strings. Our experiments on a real-world application show that subset wavelet trees offer attractive space-time tradeoffs for subset-rank queries in real-world genomics applications. A key subproblem in the navigation of a SubsetWT to answer subset-rank is computing normal rank queries on small alphabets sequences (in particular, base-3 and base-4 sequences). With this in mind, we describe and benchmark several efficient methods for that subproblem, which may be of independent interest.

Roadmap. In the next section we cover basic concepts and related work. We also provide details of our experimental setup and the data sets we use in later sections. In Section 3 we describe a simple data structure for subset-rank and subset-select that acts as a baseline against which the practical performance of our data structure can be gauged. In Section 4 we describe the subset wavelet tree and algorithms for supporting subset-rank and subset-select with it. Section 5 describes methods for computing normal rank queries on small alphabets sequences. Section 6 then reports on experiments using the SubsetWT for k-mer queries using SBWT representation discussed above.

2 Preliminaries

Rank and select on binary strings. A key tool in the design of succinct data structures is the support for the *query* operations rank and select on a bit string (or bitvector) X of length n defined as follows (for $i \leq n$ and $c \in \{0, 1\}$):

$$\mathsf{rank}_X(i, c) = \text{number of } c\text{'s among the first } i \text{ bits of } X$$
$$\mathsf{select}_X(i, c) = \text{position of the } i\text{-th } c \text{ in } X$$

Classical techniques [24] (see also [28]) require $n + o(n)$ bits to support each of the above queries in $O(1)$ time. However, the information theoretic lower bound on space usage for a bit string of length n having n_1 1s, is $\mathcal{B}(n, n_1) = \log \binom{n}{n_1} = n_1 \log \frac{n}{n_1}$ bits.

There are data structures that come within a lower order term of this lower bound while still supporting fast rank and select operations. Perhaps the foremost of these, known as "RRR", is due to Raman, Raman, and Rao Satti [29] and takes space $\mathcal{B}(n, n_1) + o(n)$ and answers all queries above in $O(1)$ time. Fast implementations of RRR exist [27, 12, 22].

Rank and select for larger alphabets. There are also solutions for rank and select for sequences on larger alphabets [14, 13, 8, 4]. Perhaps the most versatile and useful of these is the wavelet tree [14, 25], which we now describe.

Consider a (ordinary) string $S = S[0]S[1] \ldots S[n]$ over alphabet Σ. The wavelet tree of S is a balanced binary tree, where each leaf represents a symbol of Σ. The root is associated with the complete sequence S. Its left child is associated with a subsequence obtained by concatenating the symbols $S[i]$ of S satisfying $S[i] < |\Sigma|/2$. The right child corresponds to the concatenation of every symbol $S[i]$ satisfying $S[i] \geq |\Sigma|/2$. This relation is maintained recursively up to the leaves, which are associated with the repetitions of a unique symbol. At each node we store only a binary string of the same length of the corresponding sequence, using at each position a 0 to indicate that the corresponding symbol is mapped to the left child, and a 1 to indicate the symbol is mapped to the right child.

If the bit strings of the nodes support constant-time rank and select queries, then the wavelet tree supports fast rank and select on T. Before describing how those queries are carried out, it is instructive to examine a simpler query, namely accessing a given symbol in the input string using only its wavelet tree.

access: In order to obtain the value of $S[i]$ the algorithm begins at the root, and depending on the value of the root bit string B at position i, it moves down to the left or to the right child. If the bit string value is 0 it goes to the left, and replaces $i \leftarrow \mathsf{rank}_B(i, 0)$. If the bit string value is 1 it goes to the right child and replaces $i \leftarrow \mathsf{rank}_B(i, 1)$. When a leaf is reached, the symbol associated with that leaf is the value of a_i.

rank: To obtain the value of $\mathsf{rank}_S(i, c)$ the algorithm is similar. It begins at the root and goes down updating i as in the previous query, but the path is chosen according to the bits of c instead of looking at $B[i]$. When a leaf is reached, the i value is the answer.

Table 1 Statistics on the raw genomic datasets used in experiments. A k-mer is considered equal to its reverse complement in the k-mer counts. We derived a single degenerate string from each of these data sets using the Spectral Burrows-Wheeler transform.

	Number of sequences	Total length	Unique 31-mers
E. coli	745,409	18,957,578,183	170,648,610
SARS-CoV-2	1,234,695	36,808,137,972	2,407,721
Metagenome	17,336,887	8,703,117,274	2,761,523,935

select: The value of $\mathsf{select}_S(j, c)$ is computed as follows: The algorithm begins in the leaf corresponding to the character c, and then moves upwards until reaching the root. When it moves from a node to its parent, j is updated as $j \leftarrow \mathsf{select}_B(j, 0)$ if the node is a left child, and $j \leftarrow \mathsf{select}_B(j, 1)$ otherwise. When the root is reached, the final j value is the answer.

Experimental Setup. All our experiments were conducted on a machine with four 2.10 GHz Intel Xeon E7-4830 v3 CPUs with 12 cores each for a total of 48 cores, 30 MiB L3 cache, 1.5 TiB of main memory, and a 12 TiB serial ATA hard disk. The OS was Linux (Ubuntu 18.04.5 LTS) running kernel 5.4.0-58-generic. The compiler was `g++` version 10.3.0 and the relevant compiler flags were `-O3 -march=native` and `-DNDEBUG`. All runtimes were recorded by instrumenting the code with calls to the high-resolution clock of `std::chrono` in C++. The sizes of the index structures in memory were calculated by adding together the sizes of each individual component. The code to reproduce the experiments is available at `https://github.com/jnalanko/SubsetWT-Experiments`.

Datasets. We experiment on three different data sets that represent typical targets for k-mer indexing in bioinformatics applications.

1. A pangenome of 3682 E. coli genomes. The data was downloaded during the year 2020 by selecting a subset of 3682 assemblies listed in `ftp://ftp.ncbi.nlm.nih.gov/genomes/genbank/bacteria/assembly_summary.txt` with the organism name "Escherichia coli" with date before March 22, 2016. The resulting collection is available at `zenodo.org/record/6577997` .
2. A set of 17,336,887 Illumina HiSeq 2500 reads of length 502 sampled from the human gut (SRA identifier ERR5035349) in a study on irritable bowel syndrome and bile acid malabsorption [20].
3. A set of 1,234,695 genomes of the SARS-CoV-2 virus downloaded from NCBI datasets.

Table 1 shows a number of key statistics. The constructed index structures include both forward and reverse DNA strands.

3 Simple Subset Rank and Select

We now describe a straightforward way to support subset-rank and subset-select on a degenerate string X of length n over alphabet Σ in $O(1)$ time and uses $O(n\sigma)$ bits of space. For each symbol $c \in \Sigma$ we store a bit string R_c of length n such that $R_c[i] = 1$ if and only if set $X[i]$ contains symbol c. Each bit string is preprocessed for rank and select queries. To answer $\mathsf{subset\text{-}rank}_X(i, c)$ we simply return $\mathsf{rank}_{R_c}(i, 1)$. Select is answered in a similar way. The approach is fast in practice, and will act as a baseline in our experiments.

	T	G	ACGT			CG		A		A	AC			A	A
AC	0	0	1	0	0	1	0	1	0	1	1	0	0	1	1
GT	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0

	ACGT	CG	A	A	AC	A	A
A	1	0	1	1	1	1	1
C	1	1	0	0	1	0	0

	T	G	ACGT	CG
G	0	1	1	1
T	1	0	1	0

Figure 1 SubsetWT of $X = \{T\}\{G\}\{A,C,G,T\}\{\}\{\}\{C,G\}\{\}\{A\}\{\}\{A\}\{A,C\}\{\}\{\}\{A\}\{A\}$. Conceptually there are two bitvectors at each node of the tree, L_v and R_v which are shown on top of each other in the figure. As described in the text, L_v and R_v can be combined into a base-4 sequence at the root and into a base-3 sequence at other nodes. At the root of this example, the sequence would be 113003020220022, and at the left child of the root would be 3122322 (or 2011211 on a minimal base-3 alphabet).

4 Subset Wavelet Tree

We build a tree with $\log \sigma$ levels[1]. Each node of the tree corresponds to a part of the alphabet, defined as follows. We denote with A_v the alphabet of node v. The root node corresponds to the full alphabet. The alphabets of the rest of the nodes are defined recursively such that the left child of a node v corresponds to the first half of A_v, and the right child corresponds to the second half of A_v. Let Q_v be the subsequence of subsets that contain at least one character from A_v. As a special case, the subsequence Q_v also includes the empty sets when v is the root.

Each node v contains two bit vectors L_v and R_v of length $|Q_v|$. We have $L_v[i] = 1$ iff subset $Q_v[i]$ contains a character from the first half of A_v, and correspondingly $R_v[i] = 1$ iff $Q_v[i]$ contains a character from the second half of A_v. Figure 1 illustrates our running example. The bit vectors L_v and R_v can be combined to form a string on the alphabet $\{0, 1, 2, 3\}$, such that the i-th character is defined as $(2 \cdot L_v[i] + R_v[i])$.

Rank queries on L_v can then be implemented by summing the ranks of characters 0 and 2, and rank queries on R_v can be implemented by summing the ranks of characters 1 and 3. To answer our query for a character c and position i, we traverse from the root to the leaf of the tree where A_v is the singleton subset $\{c\}$. While traversing, we compute for each visited node v the length of the prefix in the current subset sequence Q_v that contains all the subsets of $X_1, \dots X_i$ that have at least one character from A_v. This is done by using rank queries on the bit vectors L_v and R_v, analogous to a regular wavelet tree query. Pseudocode is given in Algorithm 1.

To answer a select query for a character c and position i, we traverse the tree from the leaf where A_v is the singleton subset $\{c\}$ to the root. While traversing, we update i for each visited node v and compute the length of the prefix in the current subset sequence Q_v that contains all the subsets of $X_1, \dots X_i$ that together have exactly i c characters. This is done by using select queries on the bit vectors L_v and R_v, analogous to a regular wavelet tree query. Pseudocode is given in Algorithm 2.

Query time for the subset wavelet tree is clearly $O(\log \sigma)$, as constant time is spent at each of the $\log \sigma$ levels. For a general sequence of sets, the data structure requires $2n(\sigma - 1) + o(n\sigma)$ bits of space. The subset wavelet tree can be thought of as a complete binary tree with σ leaves, labeled with the symbols of the alphabet. These are not in Figure 1 since leaves are not actually stored in the subset wavelet tree. If all sets are full, then each set goes both

[1] We assume for simplicity that σ is a power of 2.

Algorithm 1 Subset wavelet tree rank query.

Input: Character c from an alphabet $\Sigma = \{1, \ldots, \sigma\}$ and an index i.
Output: The number of subsets X_j such that $j \leq i$ and $c \in X_j$.

```
function SubsetRank(i, c):
    v ← root
    [ℓ, r] ← [1, σ]
    while ℓ ≠ r do
        if c < (ℓ + r)/2 then
            r ← ⌊(ℓ + r)/2⌋
            i ← rank_{L_v}(i, 1)
            v ← left child of v
        else
            ℓ ← ⌈(ℓ + r)/2⌉
            i ← rank_{R_v}(i, 1)
            v ← right child of v
    return i.
```

Algorithm 2 Subset wavelet tree select query.

Input: Character c from an alphabet $\Sigma = \{1, \ldots, \sigma\}$ and an index i.
Output: The position of subset X_j such that the ith $c \in X_j$.

```
function SubsetSelect(i, c):
    v ← c leaf
    while v ≠ root do
        u ← parent of v
        if v = left child of u then
            i ← select_{L_v}(i, 1)
        else
            i ← select_{R_v}(i, 1)
        v ← u
    return i.
```

to the left and to the right child at each level. This means that every internal node of the subset wavelet tree stores $2n$ bits. The total size of the subset wavelet tree is then given by the number of (internal) nodes multiplied by the size of each of them, thus $(\sigma - 1)2n$.

For a balanced degenerate string, however, less space is needed. In particular, because each element in each set corresponds to at most one symbol in sequence at a given level of the tree, the total length of the sequences is bound by the total sizes of the sets, making the number of bits over all $\log \sigma$ levels of the tree $2n \log \sigma$. We thus have the following theorem.

▶ **Theorem 1.** *The subset wavelet tree of a balanced degenerate string takes $2n \log \sigma + o(n \log \sigma)$ bits of space and supports subset rank and subset select queries in $O(\log \sigma)$ time.*

5 Rank for Base-3 and Base-4 Sequences

A critical operation in answering subset rank queries with the subset WT is to answer (ordinary) rank queries on the small alphabet sequences stored at the nodes of the tree. The sequence at the root is base-4 (with alphabet $\Sigma = \{0, 1, 2, 3\}$) and the sequence at every other node is base-3 (with alphabet $\Sigma = \{0, 1, 2\}$). Actually, the required operation is more specific than a rank query: we always want to know the sum of $\mathsf{rank}(i, \sigma - 1)$ and either of $\mathsf{rank}(i, \Sigma[0])$ or $\mathsf{rank}(i, \Sigma[1])$. We call these combined queries *rank-pair* queries. In particular:

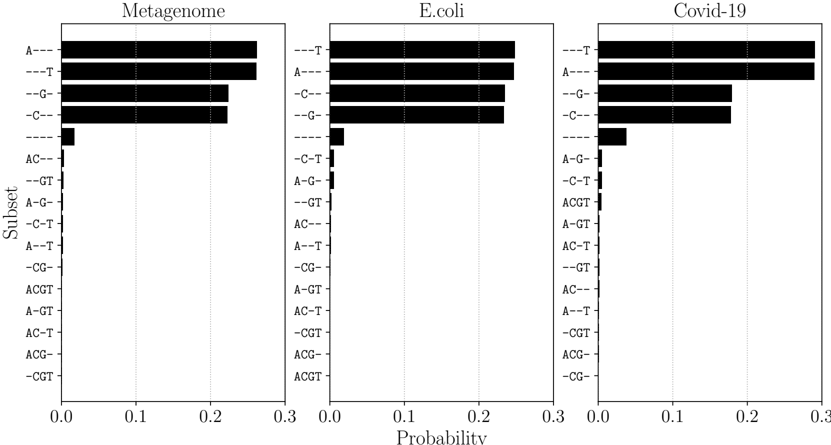


Figure 2 Distributions of subsets in the balanced degenerate string produced by the Spectral Burrows-Wheeler transform on the three genomic data sets described in Section 2. The three plots show that in all cases the set distribution is highly skewed, with the vast majority of the sets being singletons. The entropies for the distributions are: 2.21 (Metagenome), 2.24 (E.coli), and 2.31 (Covid-19).

Base-4	Base-3
$\mathsf{rankpair}(i,1) = \mathsf{rank}(i,1) + \mathsf{rank}(i,3)$	$\mathsf{rankpair}(i,0) = \mathsf{rank}(i,0) + \mathsf{rank}(i,2)$
$\mathsf{rankpair}(i,2) = \mathsf{rank}(i,2) + \mathsf{rank}(i,3)$	$\mathsf{rankpair}(i,1) = \mathsf{rank}(i,1) + \mathsf{rank}(i,2)$

In this section we examine methods for supporting rank and rank-pair queries on small alphabet sequences. All the methods we describe support both types of queries, but, importantly, some structures offer more ready support for rank-pair than do others.

We develop the structures with our spectral BWT application in mind. As stated earlier, the SBWT sequence is a balanced degenerate string. Moreover, singleton sets dominate, as the plots in Figure 2 show very clearly. In developing our rank/*rank-pair* data structures in this section, we specifically target degenerate strings with skewed distributions.

5.1 Wavelet Trees

The current de facto standard for rank queries on sequences over non-binary alphabets is the wavelet tree. We, therefore, use WTs as a baseline for the other methods we develop in this section. Using different bitvector implementations inside WTs leads to different space-time tradeoffs. We experimented with both plain bitvectors, which make the WT faster and larger, and RRR bitvectors, which, as discussed in Section 2 take $nH_0 + o(n)$ bits of space for an input bit string of n bits and are generally slower.

We used implementations from the Succinct Data Structures Library (SDSL), which are the fastest wavelet tree implementations we know of. We remark that wavelet trees as implemented in the SDSL offer no ready support for *rank-pair* queries, and so we implement *rank-pair* by issuing two separate rank queries for the appropriate symbols.

5.2 Scanning Rank

This approach is inspired by fast methods for binary rank queries [12], where the task is to count 1s up to a given position. For brevity, we describe only the structure for base-4 sequences here – the structure for base-3 sequences is essentially the same.

The data structure consists of three layers. At the lowest layer is the sequence X itself, packed into words. Assuming 64-bit words, we can pack 32 base-4 symbols into a single word, and so this layer takes $64 \cdot \lceil n/32 \rceil \approx 2n$ bits. At the highest layer, we divide X into superblocks each of size s. For each superblock we store the answer to $\mathsf{rank}(i, c)$ for all $c \in \Sigma$, where i is the start of the superblock. These answers are stored in a table of size $\sigma n/s$ words so that we can access the answers for the superblock containing a given position j in constant time at column j/s of the table. In the middle layer of the structure we divide X into blocks of size $b < s$, where b is a divisor of s. For a block beginning at position i, we precompute and store, for each symbol $c \in \Sigma$, the number of occurrences of c in $X[s\lfloor i/s \rfloor..i)$ – in other words, the count between the start of the block and the start of its enclosing superblock. If we set $s = 2^{32}$, then the block counts need 32 bits each. In our experiments we set $b = 1024$.

A critical optimization is to interleave the counts stored for each block with the part of the sequence covered by the block. In memory, the format of a block is a 2-word header containing four precomputed counts, followed by a data section of $b/64$ words into which the b symbols themselves are packed. Interleaving the header and data sections in this way, the lower and middle layers as a single array A of $(2n/b + n/32)$ words in memory. Thus, accessing the data sections of a block immediately after its header has good memory locality.

Query $\mathsf{rank}(i, c)$ is answered as follows. The header for the block containing position i starts at position $j = i/(b + 2)$ in A. We retrieve the count for c and add it to the relevant count retrieved from the superblock table. We then proceed to scan $A[j + 2..i \mod b)$, counting occurrences of c. In general this involves inspecting zero or more whole words and possibly one partial word, which together contain part of the input sequence relevant to the query. Counting occurrences of bit patterns 00, 01, 10, and 11 in whole (or partial) words can be made fast by the use of bitwise operations. *rank-pair* affords a particularly fast implementation with relevant symbol occurrences counted inside a word via a single bitwise AND (with an appropriate mask) and a single popcount operation.

5.3 Sequence Splitting

Our next structure aims to exploit the skewed distribution in real subset sequences observable in Figure 2. Because the sets in X are mostly singletons, in the base-4 sequence at the root of the SubsetWT symbols 1 and 2 will dominate[2]. With this in mind, to represent a base-4 sequence X of length n, as follows. Let $X_{a,b}$ be the subsequence of X consisting of only symbols $X[i] \in \{a, b\}$. For $X_{1,2}$ we store a bitvector L of $|X_{1,2}|$ bits where $L[i] = 1$ if $X_{1,2}[i] = 2$ and $L[i] = 0$ if $X_{1,2}[i] = 1$. We store a similar bitvector R for $X_{0,3}$: $R[i] = 1$ if $X_{0,3}[i] = 3$ and $R[i] = 0$ if $X_{0,3}[i] = 1$. Finally, we store positions i such that $X[i] \in \{0, 3\}$ in a predecessor data structure, P. If there is skewness of the subset distribution we can expect P and R to be small. Both bitvectors L and R are indexed for rank queries. In summary, the final data structure for a base-4 sequence consists of P, L and R and their rank support structures. For base-3 sequences there is no need to store the bitvector L, since P stores exclusively the indexes i such that $X[i] = 2$, as those are the only non-singleton sets.

On a base-4 sequence X query $\mathsf{rank}_X(i, c)$ is answered with a predecessor query on P for position i, which returns p, the number of elements in P smaller than i (i.e., the rank of the predecessor of i in P), followed by a binary rank query on L or R. Subtracting the result of the predecessor query p from i gives us the appropriate index for a binary rank query on L if $c \in \{1, 2\}$. In particular $\mathsf{rank}_X(i, 1) = \mathsf{rank}_L(i - p, 0)$ and $\mathsf{rank}_X(i, 2) = \mathsf{rank}_L(i - p, 1)$. For answering rank queries with $c \in \{0, 3\}$, we require a binary rank query on the bitvector

[2] In sequences at lower nodes, which are base-3, it will be symbols 0 and 1 that dominate

R at position p, in particular $\mathsf{rank}_X(i,0) = \mathsf{rank}_R(p,0)$ and $\mathsf{rank}_X(i,3) = \mathsf{rank}_R(p,1)$. Rank queries on a base-3 sequence are the same as for base-4 for singletons, $x \in \{0,1\}$, specifically $\mathsf{rank}(i,0) = \mathsf{rank}_L(i-p,0)$ and $\mathsf{rank}(i,1) = \mathsf{rank}_L(i-p,1)$. As no second binary vector is present, the result of the predecessor query gives us directly the rank of $c = 2$.

rank-pair queries with this structure can be answered faster than two separate single rank queries. Indeed, with rank-pair queries we can save a predecessor query as p is computed only once for both symbols in the query.

5.4 Extending RRR to Base-3 and Base-4 Sequences

Our final method is a generalization – to base-3 and base-4 sequences – of the famous entropy compressed bitvector due to Raman, Raman, and Rao-Satti [29], the so-called RRR data structure. RRR represents a bitstring using at most $nH_0 + o(n)$ bits and supports rank and select operations on the bitstring in $O(1)$ time per query, without needed access to the original input after construction. Practical implementations of generalizations of RRR have been proposed before [8], however our approach is different, drawing on ideas by Navarro and Providel [27] for a particular implementation of the binary RRR scheme.

Let X be a sequence of length n from an alphabet with constant size σ. We index X using a three-level structure similar to the basic binary RRR structure. That is, we segment X into blocks of size $b = O(\log n)$ and superblocks of size $B = O(\log^2 n)$, where B is a multiple of b. We precompute the counts of symbols up to the start of each superblock, and the counts of symbols inside each block. The precomputed values are represented using $O(\log n)$ bits each for superblock, and $O(\log \log n)$ bits for the regular blocks, making the total space for those values $O(n\sigma \log \log n / \log n)$, which is $o(n)$, as σ was assumed constant. A rank query $\mathsf{rank}(i,c)$ is answered in three parts: first, look up the count of c up to the superblock containing i, then, add up the counts of c in blocks preceding index i in the superblock, and lastly, add the count of occurrences of c in the prefix of length $p = i \bmod b$ in the block containing index i.

To compute the count of a symbol within a prefix of a block, we encode some extra information to be able to decode the sequence of symbols in a block, and then loop to count the number of occurrences in the prefix of length $i \bmod b$. Consider the equivalence relation that partitions the space of all the σ^b possible distinct blocks into equivalence classes such that two blocks are in the same class if and only if they contain the same multiset of symbols. We store for each block the rank r of the block in the lexicographically sorted list of blocks in its equivalence class. The class and the lexicographic rank within the class completely determine the sequence of symbols inside the block. That is, there exists a function $\mathsf{unrank}(r, d_0, d_1, \ldots d_{\sigma-1})$ that takes the lexicographic rank r and the counts $d_0, d_1, \ldots d_{\sigma-1}$ of symbols inside the block, and returns the sequence of symbols in the block. It remains to show how to implement $\mathsf{unrank}(r, d_0, d_1, \ldots d_{\sigma-1})$.

One way to implement unrank would be to precompute and store the answers to all queries $\mathsf{unrank}(r, d_0, d_1, \ldots d_{\sigma-1})$. This corresponds to the universal tables in the original RRR data structure. This, however, is space consuming for large b, so we describe a way to compute unrank without using any extra space at all. Our method can be seen as a generalization of the scheme used in the practical RRR implementation of Navarro and Providel [27], from a binary alphabet to an integer alphabet.

We denote by $\binom{n}{d_0 d_1 \ldots d_{\sigma-1}}$ the *multinomial coefficient* $\frac{n!}{d_0! d_1! \cdots d_{\sigma-1}!}$, defined so that the value is 0 if any of the $d_0, \ldots, d_{\sigma-1}$ are negative or their sum is greater than n. Let $lexrank(c_0, c_1, \ldots c_{b-1})$ be the lexicographic rank of a block $c_0, c_1, \ldots c_{b-1}$ in its equivalence class. Let $D_c(i)$ be the number of occurrences of symbol c in the suffix $c_i, \ldots, c_{b-1}$. Now we can write:

$$\mathsf{lexrank}(c_0, \ldots c_{b-1}) = \sum_{i=0}^{b-1} \sum_{j=0}^{c_i - 1} \binom{b-1-i}{D_0(i) \quad \cdots \quad D_j(i)-1, \quad \cdots \quad D_{\sigma-1}(i)},$$

where the -1 in the choices of the multinomial is only applied for choice $D_j(i)$. The formula represents a process that iterates the symbols of the block from left to right, adding up ways to complete the block using the remaining counts such that the completed block is lexicographically smaller than the input block. Computing the unrank function is a matter of inverting the lexrank function. We do this by adding the multinomials in the inner sum until the total would become greater than the target rank r. When this happens, we append the current symbol j to the sequence of the block and proceed to the next round of the outer sum. Algorithm 3 provides the pseudocode of the process for a base-4 sequence.

Algorithm 3 Base-4 block unrank. Prints the sequence of symbols in the block with rank r among the class of blocks with symbol counts d_0, d_1, d_2 and d_3.

```
function Base4BlockUnrank(r, d_0, d_1, d_2, d_3):
    b ← d_0 + d_1 + d_2 + d_3                          ▷ Block size
    s ← 0                                              ▷ Blocks counted so far
    for i = 0, ..., b − 1 do
        for j = 0, ..., 3 do                           ▷ 0 to σ − 1
            d_j ← d_j − 1
            count ← (b−1−i choose d_0, d_1, d_2, d_3)
            d_j ← d_j + 1
            if s + count > r then
                print j
                d_j ← d_j − 1
                break
            else
                s ← s + count
```

5.4.1 Practical considerations

In practice, we use a block size $b = 31$ and superblock size $B = 32b = 992$. With this choice of b, the counts of symbols inside blocks fit into 5 bits each. We omit the count of the last symbol of the alphabet in each block because it can be computed by subtracting the counts of the other symbols from the block size b. This choice of b also guarantees that lexicographic ranks of blocks within their classes always fit in 64-bit integers, assuming that the alphabet size is at most 4. To compute the multinomial coefficients for unrank, we use the formula $\binom{n}{d_0 \ldots d_{\sigma-1}} = \binom{n}{d_0}\binom{n-d_0}{d_1}\binom{n-d_0-d_1}{d_2}\ldots\binom{n-d_0-\ldots-d_{\sigma-2}}{d_{\sigma-1}}$. The expression is evaluated using only $(\sigma - 1)$ multiplications by loading the binomials from a precomputed table and omitting the last term which is always equal to 1. We terminate the block decoding process early after having decoded the prefix of the required length.

These lexicographic ranks within a class are stored compactly using $\lceil \log_2 m \rceil$ bits each, where m is the size of the class of the block. The binary representations of these ranks are concatenated in memory. Since the query algorithm will always access the list of lexicographic ranks of blocks in sequential order starting from a superblock boundary, we do not have to store the widths of all of the binary representations in the concatenation, but instead, we only store the sum of widths up to each superblock, and we can compute the width of the binary representation of the i-th block from the stored symbol counts during query time.

The binomials involved in the computation are again loaded from a precomputed table, and the integer base-2 logarithms are efficiently implemented using a machine instruction to count the number of leading zeroes in a word.

5.5 Microbenchmark

To evaluate the practical performance of the small-alphabet rank data structures developed earlier in this section, we benchmarked 10^7 rank and rank-pair queries at random positions for random characters, in the base-3 and base-4 sequences extracted from the SubsetWT of the SBWT of our metagenomic read dataset.

The smallest data structure was the RRR-based wavelet tree, which was also the slowest. The fastest was the Scanning solution, but it had the largest space. The full results are in Table 2. The WT RRR, Generalized RRR, and Split methods all achieve some level of compression, while WT plain and Scanning methods both expand on the size of the input sequence. Finally, we observe that all methods answer $\mathsf{rank} - pair$ queries in less than twice the time it takes to answer a single rank query. The most impressive *rank-pair* performance (relative to rank performance) is shown by Generalized RRR and Scanning, both of which can save significant computation when computing *rank-pair*.

Table 2 Microbenchmark results on random queries on base-4 and base-3 sequences derived from the SubsetWT of the Spectral Burrows-Wheeler transform of the metagenomic read dataset.

Sequence	Structure	space (bps)	rank time (ns)	*rank-pair* time (ns)
Base-4	WT plain	2.13	247	404
	WT RRR	1.29	1017	1517
	Generalized RRR	1.55	826	829
	Split	1.69	290	328
	Scanning	2.25	142	106
Base-3	WT plain	2.12	199	369
	WT RRR	1.15	718	1006
	Generalized RRR	1.26	681	679
	Split	1.39	224	248
	Scanning	2.25	148	107

Table 3 SBWT k-mer search queries with different subset rank implementations. The space is given in units of bits per indexed k-mer, where a k-mer is considered distinct from its reverse complement. The time is reported in microseconds per queried k-mer.

Dataset	Subset Rank Structure	Space (bpk)	Query Time (μs)
Metagenomic reads	Simple	4.66	1.49
	SubsetWT<Generalized-RRR>	3.07	44.13
	SubsetWT<WT-RRR>	2.67	71.37
	SubsetWT<Split>	3.36	6.03
E. coli genomes	Simple	4.29	1.04
	SubsetWT<Generalized-RRR>	2.84	41.21
	SubsetWT<WT-RRR>	2.48	67.01
	SubsetWT<Split>	3.17	6.84

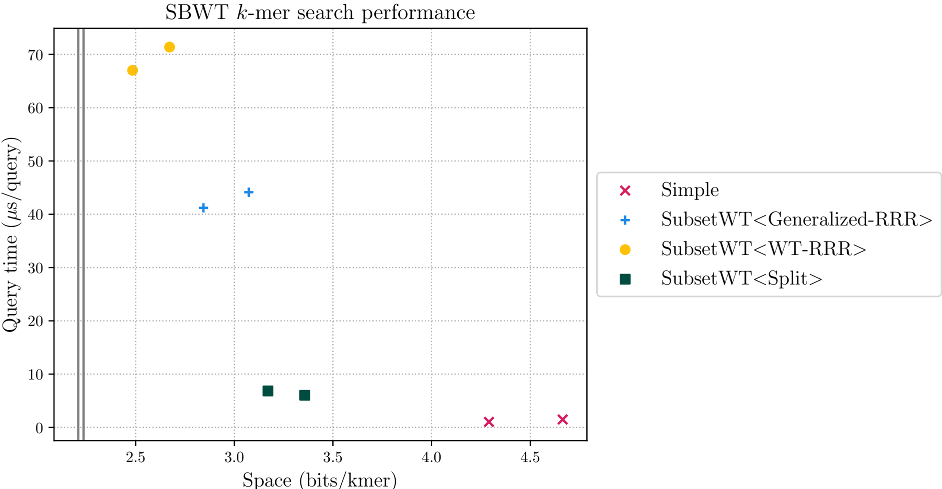


Figure 3 Time and space required for k-mer search on the SBWT using different implementations for subset rank queries. There are two data points per data structure since we have two datasets. For all data structures the metagenome result is the right point and E. coli the left. The two vertical lines mark the entropies of the distribution of subsets in the two datasets.

6 Subset rank query performance on k-mer search of the Spectral BWT

As mentioned at the start of this paper, our main interest in the SubsetWT is for implementing a k-mer search algorithm using the Spectral BWT [3], which reduces a k-mer search query to $2k$ subset rank queries on a degenerate string. In this section, we compare our implementations of the SubsetWT parameterized by different base-3 and base-4 rank structures, to the simple solution of Section 3.

We used the value $k = 31$ in all experiments. The time to load the index into memory was disregarded and the running time includes only the time spent running queries. Table 3 shows the query times against SBWT index structures built for the metagenomic read set and the E. coli genomes. In case of the metagenomic read set, we queried the first 25,000 reads in the dataset, and in case of the E. coli genomes, we queried all k-mers of a single genome in the dataset (assembly id `GCA_000005845`).

The experiments show that the most succinct solution was the SubsetWT with the RRR-encoded wavelet tree for the base-3 and base-4 rank queries, at $2.5 - 2.7$ bits per k-mer, but, on the flipside, its query time was the slowest. The generalized RRR was approximately 15% larger, but had approximately 1.6 times faster queries. The next-largest structure was the Split structure, being 26% larger than the RRR wavelet tree, with dramatically improved query time, up to 12 times faster. The plain matrix solution was the largest, being 73% larger than the RRR wavelet tree, with $48 - 64$ times faster queries. We omit from the results the SubsetWT parameterized by the scanning solution of Section 5.2 and by the plain bitvector wavelet tree of Section 5.1, since on the DNA alphabet, they are dominated in the time-space plane by the simple solution. They may lead to competitive solutions for degenerate strings on larger alphabets. Figure 3 shows the data points in Table 3 in the time-space plane.

7 Concluding Remarks

We have described the subset wavelet tree – a new data structural tool for degenerate strings. On degenerate strings from a real-world large-scale genomics application, subset wavelet trees offer significant space savings over a non-trivial baseline method, at an acceptable slowdown to query times. Along the way we have described and engineered several rank data structures specialized for ternary and quarternary sequences, which are of independent interest.

The main open problem we leave is to find a tighter analysis of the space required by subset wavelet trees when entropy compression is applied to their node sequences. In particular, can the size of the resulting structure be related in some way to the entropy of the subset sequence. Our experimental results suggest this may well be the case.

Another interesting avenue for future work is to apply the new small alphabet rank data structures we have developed to other settings, for example FM indexes [9] for DNA sequence data, or structures currently of a somewhat esoteric nature, such as multiary wavelet trees [8]. Our results suggest some of our structures (e.g., Scanning) are superior to regular wavelet trees, which until now have been the main practical solution for rank on non-binary sequences and are currently in wide use via the Succinct Data Structures Library [11].

References

1. Karl R. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.
2. H. Alamro, M. Alzamel, C.S. Iliopoulos, S. P. Pissis, and S. Watts. IUPACpal: efficient identification of inverted repeats in IUPAC-encoded dna sequences. *BMC Bioinformatics*, 22(51), 2021.
3. Jarno N Alanko, Simon J Puglisi, and Jaakko Vuohtoniemi. Succinct k-mer sets using subset rank queries on the spectral Burrows-Wheeler transform. *bioRxiv*, 2022.
4. J. Barbay, M. He, I. Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4):article 52, 2011.
5. E. Cambouropoulos, T. Crawford, and C.S. Iliopoulos. Pattern processing in melodic sequences: Challenges, caveats and prospects. *Computers and the Humanities*, 35:9–21, 2001.
6. Rayan Chikhi. A tale of optimizing the space taken by de Bruijn graphs. In *Proc. 17th Conference on Computability in Europe (CiE)*, volume 12813 of *LNCS*, pages 120–134. Springer, 2021.
7. Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Covering problems for partial words and for indeterminate strings. *Theor. Comput. Sci.*, 698:25–39, 2017.
8. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.
9. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398. IEEE Computer Society, 2000.
10. Michael J. Fischer and Michael S. Paterson. String-matching and other products. *Complexity of Computation*, 7:113–125, 1974.
11. Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, LNCS 8504, pages 326–337. Springer, 2014.
12. Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Softw. Pract. Exp.*, 44(11):1287–1314, 2014.

13 A. Golynski, I. Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

14 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), !booktitle = "SODA"*, pages 841–850, 2003.

15 Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome biology*, 21(1):1–20, 2020.

16 Jan Holub, William F. Smyth, and Shu Wang. Fast pattern-matching on indeterminate strings. *J. Discrete Algorithms*, 6(1):37–50, 2008.

17 Costas S. Iliopoulos and Jakub Radoszewski. Truly subquadratic-time extension queries and periodicity detection in strings with uncertainties. In Roberto Grossi and Moshe Lewenstein, editors, *Proc. 27th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 54 of *LIPIcs*, pages 8:1–8:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

18 Costas S. Iliopoulos, M. Sohel Rahman, Michal Vorácek, and Ladislav Vagner. The constrained longest common subsequence problem for degenerate strings. In Jan Holub and Jan Zdárek, editors, *Proc. 12th International Conference on Implementation and Application of Automata (CIAA)*, LNCS 4783, pages 309–311. Springer, 2007.

19 IUPAC-IUB Commission on Biochemical Nomenclature. Abbreviations and symbols for nucleic acids, polynucleotides, and their constituents. *Biochemistry*, 9(20):4022–4027, 1970.

20 Ian B Jeffery, Anubhav Das, Eileen O'Herlihy, Simone Coughlan, Katryna Cisek, Michael Moore, Fintan Bradley, Tom Carty, Meenakshi Pradhan, Chinmay Dwibedi, et al. Differences in fecal microbiomes and metabolomes of people with vs without irritable bowel syndrome and bile acid malabsorption. *Gastroenterology*, 158(4):1016–1028, 2020.

21 Mikhail Karasikov, Harun Mustafa, Daniel Danciu, Marc Zimmermann, Christopher Barber, Gunnar Rätsch, and André Kahles. Metagraph: Indexing and analysing nucleotide archives at petabase-scale. *BioRxiv*, 2020.

22 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Proceedings of the Data Compression Conference (DCC)*, pages 302–311, 2014.

23 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

24 J. Ian Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 1180, pages 37–42. Springer, 1996.

25 G. Navarro. Wavelet trees for all. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7354, pages 2–26, 2012.

26 Gonzalo Navarro. *Compact Data Structures – A Practical Approach*. Cambridge University Press, 2016.

27 Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Experimental Algorithms: 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings 11*, pages 295–306. Springer, 2012.

28 Rajeev Raman. Rank and select operations on bit strings. In *Encyclopedia of Algorithms*, pages 1772–1775. Springer, 2016. doi:10.1007/978-1-4939-2864-4_332.

29 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

30 Sun Wu and Udi Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX Winter 1992 Technical Conference*, pages 153–162, 1992.

Engineering Shared-Memory Parallel Shuffling to Generate Random Permutations In-Place

Manuel Penschuck
Goethe Universität Frankfurt, Germany

Abstract

Shuffling is the process of placing elements into a random order such that any permutation occurs with equal probability. It is an important building block in virtually all scientific areas. We engineer, – to the best of our knowledge – for the first time, a practically fast, parallel shuffling algorithm with $\mathcal{O}\left(\sqrt{n}\log n\right)$ parallel depth that requires only poly-logarithmic auxiliary memory (with high probability). In an empirical evaluation, we compare our implementations with a number of existing solutions on various computer architectures. Our algorithms consistently achieve the highest through-put on all machines. Further, we demonstrate that the runtime of our parallel algorithm is comparable to the time that other algorithms may take to acquire the memory from the operating system to copy the input.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Shuffling, random permutation, parallelism, in-place, algorithm engineering, practical implementation

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.5

Supplementary Material *Software (Source Code)*: https://crates.io/crates/rip_shuffle
Software (Source Code and Raw Data): https://zenodo.org/record/7876820

Funding *Manuel Penschuck*: Supported by the Deutsche Forschungsgemeinschaft (DFG) under grant ME 2088/5-1 (FOR 2975 – Algorithms, Dynamics, and Information Flow in Networks).

1 Introduction

Random permutations are heavily studied in many fields of science with numerous applications. They are commonly considered an "easy and fair" arrangement and thus influence many aspects of everyday life ranging from shuffling a deck of cards in a friendly game to determining the fateful order in which soldiers are drafted for war (e.g., [24]).

In computer science, applications include numerical simulations, sampling of complex objects, such as random graphs, machine learning, or statistical tests (e.g., [4, 16, 20, 26]). Especially, if coupled with rejection sampling, shuffling can become a dominating subroutine (e.g., [1] which triggered this work). Further, the assumption that an input is provided in random order (instead of adversarially) allows for practical algorithms that are almost always efficient. Among others, this notion motivates the random-order-model for online algorithms [11]. For the same reason, implementations of offline algorithms may start by shuffling their inputs; for instances, folklore suggests to shuffle the input before sorting it with a simple *Quicksort* implementation.

From an algorithmic point of view, the tasks of *shuffling* and *sorting* are tightly connected since both require an algorithm capable of emitting any permutation. Though, while sorting needs to handle adversarial inputs, shuffling can be optimized for the well-behaved uniform distribution. Shuffling can be implemented in linear-time via integer sorting by augmenting each input element with a uniform variate and sorting by it [7]; we refer to this approach as *SortShuffle*. The famously impractical *BogoSort* demonstrates the other direction, namely sorting by shuffling, but suffers from a "slightly" suboptimal expected runtime of $\Omega(n \cdot n!)$ [14].

© Manuel Penschuck;
licensed under Creative Commons License CC-BY 4.0

The quest for *in-place* algorithms is driven by the various costs of memory. The most obvious aspect is that the maximal data set size that can be handled by a machine roughly halves if the output is produced in a copy. Further, it takes a considerable time to allocate main memory on modern computer systems; in Section 6 we demonstrate that the runtime of our shuffling algorithm is comparable to the time it takes to acquire additional memory of input size. Another hidden cost is the increased code complexity to handle failed allocations of dynamic memory (e.g., because the system ran out of memory). Finally, some programming languages have a concept of non-copyable data; e.g., in C++ the copy-constructor can be deleted, and in Rust data types need to explicitly declare that they can be cloned.

1.1 Our contributions

We design and implement the practical shared-memory parallel algorithm *Parallel In-Place ScatterShuffle* (*PIpScShuf*). Our contributions include:

- The algorithm is an in-place modification of *ScatterShuffle* [29]. Instead of directly putting elements to random positions, *ScatterShuffle* instead assigns the elements to random buckets and recurses on them until eventually a random permutation is achieved.
- We show that our *PIpScShuf* has (whp) a parallel depth of $\mathcal{O}(\log(n)\sqrt{nk/\log k})$ and uses $\mathcal{O}(n \log_k(n))$ work (see Section 2.1 for definitions) where k is small tuning parameter.
- While it is straightforward to implement *ScatterShuffle* in-place using standard techniques (e.g., by sampling the buckets sizes as a multinomial followed by weighted sampling to distribute the elements [26]), we design a multi-staged assignment process for practical performance inspired by *MergeShuffle* [3].
- We first assume that all buckets have the same sizes and randomly assign most elements very efficiently. We then show that an asymptotically negligible and practically cheap repair step can produce the required original random distribution.
- We provide fast shuffle implementations in a free and well-tested plug-and-play Rust library. Our *PIpScShuf* does not use heap allocations and emits reproducible permutations if a seedable pseudo-random number generator is provided.

After a discussion of notation and related work in Sections 2 and 3, we derive the sequential *In-Place ScatterShuffle* (*IpScShuf*) in Section 3 and parallelize it in Section 4. In Section 5, we discuss details of our implementations which we then evaluate in Section 6.

2 Preliminaries and notation

The expression $(x_i)_{i=a}^{b}$ denotes the sequence $x_a, \ldots, x_b$ and may be shortened to $(x_i)_i$ if the limits are implied by context. We indicate an array of n elements as $X[1..n]$ and reference the subrange $X[i], \ldots, X[j]$ as $X[i..j]$. Further, $[n]$ denotes the set $\{1, \ldots, n\}$. Then, a permutation is a bijection $\pi\colon [n] \to [n]$ where $\pi(i)$ encodes the position of the i-th input element in the output. We say that a probabilistic statement holds *with high probability* (whp) if the error probability is at most $1/n$ for some implied parameter n.

2.1 Parallel model of computation

For parallel algorithms, we assume the commonly accepted binary Fork-Join model [8]. This choice fits the `rayon`[1] infrastructure used in our implementation well. An execution starts with a single task on a unit-cost random access machine. Additionally, any task t_0 can

[1] `https://crates.io/crates/rayon`

Algorithm 1 Fisher Yates Shuffle on input $A[1..n]$. The array eventually holds the output.

```
1 for i in 1 to n−1 do
2 |  j ← uniform sample from [i..n]     // A[1..i − 1] already have final values
3 |  swap elements A[i] and A[j]
```

recursively *fork* into two tasks t_1 and t_2. In this case t_0 waits until t_1 and t_2 complete their computation and *join* to resume t_0. In practice, Fork-Join frameworks, such as oneTBB[2], Cilk[3] [19], or rayon, use a worker-pool in combination with a work-stealing scheduler to map tasks to cores. Algorithmic performance measures are the *work*, i.e. the total number of instructions, and the *parallel depth* (*span*), defined as the length of the critical path which corresponds to the execution time assuming an unbounded number of workers.

2.2 Random shuffling

The sequential *Fisher-Yates-Shuffle* (*FY*, also know as *Knuth-Shuffle*) [18] obtains a random permutation of an array $A[1..n]$ in time $\mathcal{O}(n)$. As summarized in Algorithm 1, conceptually, it places all items into an urn, draws them sequentially without replacement, and returns the items in the order they were drawn. The algorithm works in-place and fixes the value of $A[i]$ in iteration $i \in [1...n-1]$ by swapping $A[i]$ with $A[j]$ where j is chosen uniformly at random from the not yet fixed positions $[i..n]$. In other words, in the i-th iteration, the $(i-1)$-prefix of A stores the result obtained so far, while the $(n-i)$-suffix represents the urn.

Shun et al. show that this seemingly inherently sequential algorithm exposes sufficient independence to be processed with logarithmic parallel depth (whp) [30]. Later, Gu et al. propose an in-place variant based on the so-called decomposition property of the parallel *FY* [15]. However, both algorithms are designed to solve a subtly different problem. They *permute* the input in an explicitly prescribed manner. As such, the permutation is part of the input and the implementation[4] of [15] uses two additional pointers per element (i.e. shuffling 32 bit values on a 64 bit machine leads to a five-fold increase of memory).

A random permutation can be computed in parallel by $\mathfrak{P}$ processors by assigning each element to one of $\mathfrak{P}$ buckets uniformly at random and then applying the sequential algorithm to each bucket [29]. We refer to this algorithm as *ScatterShuffle* and discuss it in detail in Section 3. A similar technique yields an I/O-efficient random permutation algorithm [29].

Going the opposite direction also yields an efficient algorithm. *MergeShuffle* first assigns each processor a contiguous section of the input array, shuffles the subproblems pleasingly parallel and finally recursively merges them to obtain a larger random permutation [3]. Here, merging of two input sequences A and B exploits that A and B were previously shuffled. Hence, the relative order of elements from A (and B respectively) can be kept in the output. In other word, the merging phase conceptually produces a $|A|+|B|$ bit vector with exactly $|B|$ ones. If the i-th zero is at position j, we place $A[i]$ to the j-th output position (analogously for ones and B).

This merging can be interpreted as the inverse of *ScatterShuffle*'s scatter with two buckets. In a precursor study, we found it too slow to generalize *MergeShuffle* using k-way merging which is needed to reduce the recursion depth. *MergeShuffle* further uses a sequential merge

[2] previously known as Intel Threaded Building Blocks, `https://github.com/oneapi-src/oneTBB`
[3] see also `https://www.opencilk.org`, `http://cilkplus.org`
[4] `https://github.com/ucrparlay/PIP-algorithms` master at time of writing (`6af1df9`)

procedure and we are unaware of a parallelization that is as efficient as our *PIpScShuf* based on *ScatterShuffle*. However, the authors show that if $|A| \approx |B|$, we can assign the positions of all but expected $\mathcal{O}\left(\sqrt{|A|+|B|}\right)$ elements using a single random bit. A generalization of this insight is a crucial building block for our *RoughScatter* routine in Section 3.3.

Cong and Bader [7] empirically study additional techniques such as shuffling using sorting algorithms (*SortShuffle*) or random dart-throwing (*DartThrowingShuffle*). We are, however, unaware of how to implement these approaches in-place. Yet, it is worth pointing out that our *IpScShuf* algorithm can be interpreted as an optimized in-place RadixSort in which buckets are randomly drawn. As such, there are conceptual similarities to *SortShuffle*. In a precursor study, we found that even highly optimized parallel and in-place sorting algorithms, such as IPS2RA [2], are outperformed by our *PIpScShuf* implementation. This can be attributed to the fact that sorting is a much more constraint problem, while shuffling can algorithmically exploit the features of uniform permutations.

2.3 Sampling from discrete distributions

In the following, we sample from several discrete probability distributions (arguably, shuffling is just that). This is achieved by first obtaining a stream of independent and unbiased random bits that are subsequently reshaped to attain the required distribution. The default way of implementing the first step is using a pseudo-random generator, such as Pcg64Mcg [25].

Sampling an integer from $[0, s)$ with $s = 2^k$ for some $k \in \mathbb{N}$ from random words is very cheap and involves only shifting and masking. We adopt rejection-based algorithms with expected constant time to sample uniform variates from $[0, s]$ for general s (see [20]) and binomial variates (see [9]). Sampling of k-dimensional multinomial variates is implemented by chaining appropriately parametrized binomial samples in expected time $\mathcal{O}(k)$.

3 Sequential in-place shuffling

In this section, we propose *In-Place ScatterShuffle* (*IpScShuf*), a sequential in-place variant of Sanders' parallel *ScatterShuffle* [29]. Building on the performance results obtained, we reintroduce parallelism in Section 4.

3.1 State of the art

It seems that the simple and fast *Fisher-Yates Shuffle* (*FY*, see Section 2.2) is the shuffle algorithm most commonly used in practice. Due to its simplicity, the algorithm typically outperforms more advanced schemes for small inputs. However, *FY*'s unstructured accesses to main memory cause a severe slowdown for larger inputs. This is especially relevant for parallel algorithms where the memory subsystem is shared between cores (see Section 6).

ScatterShuffle (Algorithm 2) is designed to be a parallel algorithm that also fares well in the external memory model [29]. Given an input $(x_i)_{i=1}^n$, the algorithm moves each input element x_i into a bucket drawn independently and uniformly from $B_1, \ldots, B_k$. Afterwards, each bucket constitutes an independent subproblem of expected $\Theta(n/k)$ elements on which we recurse. For small subproblems, we switch to *FY* as the base case algorithm.

While we refer to [29] for a formal correctness proof, the following intuition should suffice to follow this article. Consider that we augment each input element x_i with a random integer r_i chosen uniformly from $[0; 2^\ell)$ where ℓ is sufficiently large such that all r_i are unique. Then, we use RadixSort to order the elements according to these random keys (starting with

Algorithm 2 Sequential variant of *ScatterShuffle*. The buckets' total size is $\Theta(n)$.

```
1  Function ScatterShuffle(X = [x_1, ..., x_n], k)      // X is modified in-place
2    if n is small then                                 // Base case for small inputs
3      FisherYates (X) and return
4    Initialize empty buckets B_1, ..., B_k
5    for x ∈ X do                                       // Assign elements to buckets
6      copy x into B_j where B_j is uniformly chosen from B_1, ..., B_k
7    s ← 1
8    for B_j ∈ {B_1, ..., B_k} do                       // Recurse and overwrite input X
9      ScatterShuffle(B_j, k)
10     X[s..(s+|B_j|)] ← B_j
11     s ← s + |B_j|
```

the most significant k-ary digit); this will yield a uniform permutation by construction. Now observe that the buckets in *ScatterShuffle* and RadixSort are treated analogously with the difference that *ScatterShuffle* samples the digits on-demand.

In the original parallel formulation of *ScatterShuffle*, the number of buckets k equals the number of processing units $\mathfrak{P}$ to expose the maximal degree of parallelism. Sanders, however, already discusses that in the presence of memory hierarchies, the parameter k should be chosen sufficiently small such that the individual processors can cache at least the tail of each bucket. In his implementation[5], the parameter is $k = 32$ for the largest runs in [29]. At time of writing – more than two decades later, using very different hardware to run experiments with more than three orders of magnitude larger data sets – we empirically find $k \leq 64$ to be the best choice for our *In-Place ScatterShuffle* over a wide range of input sizes. Hence, the parameter k should be intuitively treated as a small constant that governs primarily the branching factor of the recursion. In Section 4, we will add parallelism independent of k.

Our main modification to *ScatterShuffle* is *In-place Scatter* (*IpSc*) which scatters the input into k buckets. It effectively replaces lines 4–6 in Algorithm 2. Instead of copying the input into new arrays representing the buckets, the buckets become disjoint memory regions of the input (e.g., represented by two pointers). Then, the recursion (line 9) can directly modify each bucket's memory without copying the elements. Formally, let $X = (x_i)_{i=1}^n$ be the input of *IpSc*. Further, let $A = (a_i)_{i=1}^n$ be independent uniform variables from $[1, k]$ indexing into the aforementioned buckets. Then, *IpSc* groups X by A by rearranging the elements in X with some permutation π that sorts A. We exploit that the order of elements *within* a bucket can be arbitrary as the recursion will shuffle them randomly later on.

Similar problems have been studied in the context of integer sorting. The special case of $k = 2$ (i.e. binary partition) and $k = 3$ (known as the Dutch national flag problem) can be efficiently solved in-place [10, 23]. For $k > 3$, two-pass approaches can be used (e.g., *American flag sort* [22]) but require repeated access each a_i. The parallel implementation [31] of *ScatterShuffle* included in `libstdc++` uses this technique and stores A explicitly requiring $\Theta(n \log k)$ bits. Another way, in the spirit of [13], is to require a pseudo-random generator that can be replayed multiple times by copying and retrieving the generator's internal state.

[5] `https://web.archive.org/web/20050827081959/http://www.mpi-sb.mpg.de/~sanders/programs/`

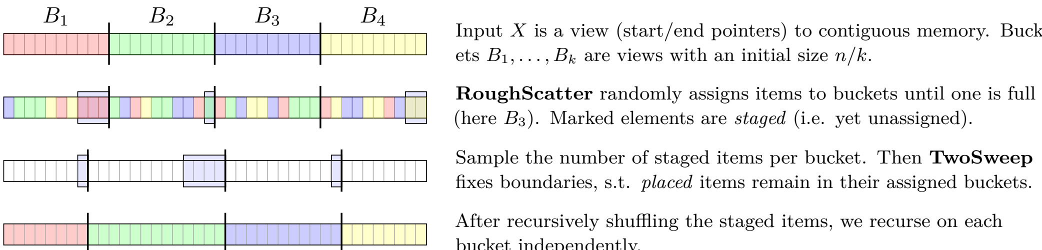


Figure 1 *In-Place ScatterShuffle* (*IpScShuf*). The first three steps constitute *In-place Scatter* (*IpSc*). All operations are either pointer arithmetic or swapping of items. No input element is copied.

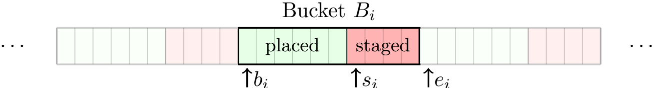


Figure 2 *IpSc* partitions the input into k buckets, each roughly containing n/k elements. Initially, all items are *staged* ($b_i = s_i$) and the bucket is said to be *empty*. Eventually, more and more items are placed (from the left). If $s_i = e_i$ the bucket is said to be *full*.

3.2 IpScShuf – an in-place implementation of ScatterShuffle

In the following, we describe *In-place Scatter* (*IpSc*) that supports true random bits, and, whp, runs in linear time using only $\mathcal{O}(k \log k)$ bits additional storage. Since each of the n items is assigned a uniformly selected bucket, the numbers $(n_i)_{i=1}^k$ of elements assigned to each bucket follow a multinomial distribution and are tightly concentrated around n/k.

For the remainder, we assume that $n \gg k(\log k)^3$ and $k^3 \log k = \mathcal{O}(n)$, since otherwise, the problem is so small that *Fisher-Yates Shuffle* is more appropriate. These assumptions are only needed to bound *IpScShuf*'s complexity and do not affect its correctness. In practice, they translate to a minimal recommended size of roughly 10^5 elements.

A straightforward solution is to draw the sizes of all buckets as a multinomial random variate. We then sample the buckets without replacement weighted by their decreasing target size. This can be implemented in expected linear time using suitable dynamic weighted sampling data structures (e.g., [21]). As discussed further in Section 5, such approaches are outperformed by the following scheme. Inspired by the framework of [3] (but quite different in its details), our assignment task consists of two phases that are illustrated in Figure 1. Firstly, during the *RoughScatter* phase, we very efficiently assign the vast majority of items – but almost certainly not all of them. Secondly, during the *FineScatter* phase, we process the remaining few elements.

3.3 RoughScatter – the opportunistic work horse

RoughScatter exploits the aforementioned concentration of the final bucket sizes around their mean of n/k to assign elements in an opportunistic fashion until we hit said n/k barrier. Let $X[1..n]$ denote the input array. As illustrated in Figure 2, we partition X into k contiguous buckets of equal sizes n/k (up to rounding). Each bucket B_i is stored as a triple of indices (b_i, s_i, e_i) where b_i points to the beginning of B_i and e_i beyond the bucket's end. A bucket is further subdivided into (i) an initially empty segment $X[b_i..s_i)$ of so-called *placed* items and (ii) $X[s_i..e_i)$ of so-called *staged* items. We say that bucket B_i is *full* iff all items are placed, i.e. $s_i = e_i$. Up to the last step in Section 3.4, *staged* items can be freely moved around, whereas the position of *placed* items carries meaning.

In each iteration, the algorithm finalizes the bucket assignment of the element x that s_1 points to, i.e. the first staged element in B_1 at that point in time. To this end, we randomly draw a partner bucket j uniformly from $[1..k]$, swap the elements $X[s_1] \leftrightarrow X[s_j]$ (skipped if $j = 1$), and increment s_j. As a result, element x is moved into the *placed* region of the partner bucket B_j. If B_j is now full, the algorithm stops, otherwise it repeats.

▶ **Lemma 1.** *Let P be the set of elements placed by RoughScatter and let $x \in P$ be an arbitrary placed item. Then x is assigned to bucket B_i with probability $1/k$.*

Proof. Without loss of generality, we assume that initially all items are staged. Then, there is a unique iteration for each item $x \in P$ in which it gets placed. To this end, the then still staged element x is swapped with a staged item y where $y \in B_i$ with probability $1/k$. It then increases s_i and thereby defines x as placed. Since *RoughScatter* only swaps staged items, this placement of x is final. The possible change of position of element y is inconsequential, since each assignment is carried out independently. ◀

In the following, we bound the number elements that remain staged after *RoughScatter*.

▶ **Lemma 2.** *After a RoughScatter run, let $r_i = e_i - s_i$ be the number of items still staged in bucket b_i and $R = \sum_i r_i$ their sum. For $n \gg k(\log k)^3$, we have $R \leq \sqrt{2nk \log k}$ whp.*

Proof. We interpret the input to *RoughScatter* as n *balls* that are independently thrown into k uniform bins. If we run the balls-into-bins experiment to completion, the maximal load of any bucket is at most $M(n, k) = \frac{n}{k} + \sqrt{2\frac{n}{k} \log k}$ whp [27].[6]

Let n' be the number of balls assigned in said game when the maximal load first reached n/k. Algorithmically, this corresponds to the termination of *RoughScatter*. By identifying $M(n', k) = n/k$ and solving for n', we find that whp $n' \geq n - \sqrt{2nk \log k} := n - R$. ◀

▶ Remark 3. The fraction of unprocessed elements R/n vanishes for $n \to \infty$. Even for small inputs with $n = 2^{22}$ and practical $k = 64$, less than 1% of the input remains unassigned whp.

3.4 FineScatter – fixing the small remainder

After the execution of *RoughScatter* only $R = \mathcal{O}(\sqrt{nk \log k})$ items need to be assigned during the *FineScatter* phase whp. If our initial assumption still holds for $R \gg k(\log k)^3$, we can compact the staged items into a contiguous memory area, apply *RoughScatter* and recurse. However, for small inputs the assumption is likely violated, while for large inputs the fraction R/n contributes only negligibly to the total runtime. Thus, we do not consider it worthwhile to devise a merging procedure for this case and instead directly use a dedicated base case algorithm based on the following Lemma. It lays out the route to efficiently obtain the final bucket sizes and independently assign the remaining elements.

▶ **Lemma 4.** *Let $X = (x_i)_{i=1}^n$ be a sequence and $N = (n_i)_{i=1}^k$ be sampled from a multinomial distribution with equal weights $p = 1/k$ such that $\sum_i n_i = n$. Let $f_N : [n] \to [k]$ be an arbitrary partition of X with class sizes N. Finally, let $\pi : [n] \to [n]$ be a random permutation. Then, for fixed i and j, the probability that element x_i is mapped by $f_N(\pi(i))$ to class j is $1/k$.*

Proof. Due to symmetry, it suffices to consider the first partition class $j = 1$. Its size n_1 follows a binomial distribution over n attempts with success probability $p = 1/k$ by definition of the multinomial distribution. Further, let γ be a permutation such that the composition

[6] A similar argument was already used in the analysis of *ScatterShuffle* [29].

$f_N \circ \gamma$ maps the indices $1, \ldots, n_1$ to the first partition class. Due to uniformity, π itself and $\pi' = \pi \circ \gamma$ are equally likely. Thus, it suffices to compute the total probability that π' puts a fixed x_i into the first n_1 ranks over all $0 \le n_1 \le n$:

$$\sum_{j=0}^{n} P[\pi'(i) \le n_1 \mid n_1 = j] \cdot P[n_1 = j] = \sum_{j=0}^{n} \underbrace{\frac{j}{n}}_{=1-(1-\frac{j}{n})} \cdot \binom{n}{j} \left(\frac{1}{k}\right)^j \left(1 - \frac{1}{k}\right)^{n-j} \tag{1}$$

$$= \underbrace{\sum_{j=0}^{n} 1 \cdot \binom{n}{j} \left(\frac{1}{k}\right)^j \left(1 - \frac{1}{k}\right)^{n-j}}_{=1} - \sum_{j=0}^{n} \underbrace{(1 - \frac{j}{n}) \cdot \binom{n}{j}}_{=\begin{cases} \binom{n-1}{j} & \text{if } j < n \\ 0 & \text{if } j = n \end{cases}} \left(\frac{1}{k}\right)^j \left(1 - \frac{1}{k}\right)^{n-j} \tag{2}$$

$$= 1 - \underbrace{\sum_{j=0}^{n-1} \binom{n-1}{j} \left(\frac{1}{k}\right)^j \left(1 - \frac{1}{k}\right)^{(n-1)-j}}_{=1} \left(1 - \frac{1}{k}\right)^1 = 1/k$$ ◀

3.4.1 Finalizing the bucket sizes

Let $N = (n_i)_i$ be the numbers of elements assigned to bucket B_i by *RoughScatter*. Guided by Lemma 4, the base case algorithm first draws a multinomial variant $N' = (n'_i)_i$ where n'_i corresponds to the number of elements that will be placed into bucket B_i by *FineScatter*. Then, the final sizes $N^f = (n^f_i)_i$ are $n^f_i = n_i + n'_i$. Since N and N' follow a multinomial distribution with k equally weighted classes, their sum N^f does too.

By construction, the expected bucket size is n/k. Let $d_i = n^f_i - n/k$ denote the deviation of the size of bucket B_i, i.e. the number of elements it needs to gain over the initial estimation of *RoughScatter*. Analogously to the proof of Lemma 2, we bound $\max_i \{|d_i|\} = \mathcal{O}(\sqrt{n/k \log k})$ whp [27, 29]. Thus, in all likelihood, the bucket boundaries only move slightly.

Luckily, *ScatterShuffle* is oblivious to the order of elements within a bucket prior to recursion. Thus, it suffices to appropriately move a few items near the boundaries of the buckets using our *TwoSweep* algorithm. First, we iterate over the buckets in ascending index order. Meanwhile, we keep a counter $C_i = \sum_{j=1}^{i-1} d_j$ that indicates how many additional items are needed left of the current bucket B_i. If bucket B_i is too large by more than C_i items, we swap the excess staged items into the staging area of bucket B_{i+1}. In a second sweep from B_k to B_1, we move the remaining excess items towards smaller bucket indices.

▶ **Lemma 5.** *Let $N^f = (n^f_i)_i$ be the final bucket sizes, denote their deviation from the mean n/k as $d_i = n^f_i - n/k$, and let $D_i = \sum_{j=1}^{i} d_j$ be the inclusive prefix sum of deviations. Then, TwoSweep executes a total of $M(N^f) = \sum_i |D_i|$ swaps and takes time $\mathcal{O}\big(k + M(N^f)\big)$.*

Proof. *TwoSweep* can exchange a staged item of bucket B_i with its direct neighbors $B_{i\pm1}$ in $\mathcal{O}(1)$ time by executing a single swap and adopting the pointers of the two involved buckets; exchanging an item between buckets B_i and B_j this way implies a chain of $|i - j|$ swaps causing $\mathcal{O}(|i - j|)$ work. Based on this, *TwoSweep* carries out two snow-plow-like motions likely pushing intermediate items along the chain. For the remainder see Appendix C. ◀

▶ **Corollary 6.** *TwoSweep takes time $\mathcal{O}\big(k\sqrt{nk \log k}\big)$ whp.*

Proof. see Appendix C. ◀

3.4.2 Assigning the remaining staged elements

At this point in the execution, all buckets have reached their final sizes, but each bucket B_i still has n'_i staged items with $\sum_i n'_i = R$. Rather than sampling weighted by $N' = (n'_i)_i$, we apply Lemma 4 and instead randomly shuffle all staged items. This can be done by compacting the staged item into $X[1..R]$ and shuffling X. To this end, we swap the staged items with the items originally stored in X; after shuffling, we apply the same swap sequence in reverse to restore the original items and put the staged items into a now random permutation.

3.5 Putting it all together

In-place Scatter (*IpSc*) is the algorithm executed on each recursion layer of *IpScShuf*. It runs *RoughScatter* and *FineScatter* in sequence to randomly assign n elements to k buckets. The input is rearranged such that each bucket corresponds to a contiguous memory region.

▶ **Lemma 7.** *For $n \gg k \log^3 k$ and $k^3 \log k = \mathcal{O}(n)$, IpSc assigns n items in time $\mathcal{O}(n)$ whp.*

Proof. We sum up the four tasks carried out:

1. *RoughScatter* first partitions the input into buckets in time $\mathcal{O}(k)$ and then randomly assigns $n - R = \mathcal{O}(n)$ elements whp. Assuming that obtaining a word of randomness takes constant time, this translates into a time complexity of $\mathcal{O}(k + n) = \mathcal{O}(n)$.
2. Sampling a k-dimensional multinomial random variate takes time $\mathcal{O}(k)$ whp.
3. Running *TwoSweep* to adjust the boundary size takes time $\mathcal{O}\left(k\sqrt{nk \log k}\right) = \mathcal{O}(n)$ whp.
4. Shuffling the staged items with *Fisher-Yates Shuffle* takes time $\mathcal{O}(R) = \mathcal{O}(n)$ time.[7] ◀

In-Place ScatterShuffle consists of recursive applications of *IpSc*. We stop the recursion on a subproblem as soon as it reaches the base case size of $N_0 = \mathcal{O}(1)$ at which point it is finalized using *Fisher-Yates Shuffle*.

▶ **Theorem 8.** *With high probability In-Place ScatterShuffle takes time $\mathcal{O}(n \log_k(n/N_0))$ and $\mathcal{O}(k \log_k(n/N_0))$ additional words of storage where $N_0 = \Omega(k^3)$ is the base case size.*

Proof. *IpScShuf* splits an input of length n into k independent subproblems of size $\Theta(n/k)$ whp. It then calls itself recursively until the base case size of N_0 is reached. Whp, this involves $\mathcal{O}(\log_k(n/N_0))$ recursion layers, each taking time $\mathcal{O}(n)$ and requiring $\mathcal{O}(k)$ words of memory for a depth-first traversal. The base case *FY* uses $\mathcal{O}(1)$ words of memory and takes time $\mathcal{O}(n')$ for a subproblem of size n' and in total $\mathcal{O}(n)$ for all subproblems. ◀

4 Parallel algorithms

In this section, we introduce *Parallel In-Place ScatterShuffle* (***P****IpScShuf*), a parallel variant of *IpScShuf*. It is obvious that after running *IpSc* (i.e. a single recursion level of *IpScShuf*) we can process the k independent subproblems pleasingly in parallel – this is one of the core insights of the original *ScatterShuffle* [29]. Unfortunately, in our case, parallelizing the subproblems alone leads to a linear parallel depth, since the first *IpSc* execution requires $\Omega(n)$ sequential work. Therefore, we also have to parallelize *IpSc* itself. We focus on the parallelization of *RoughScatter* which, in practice, accounts for the vast majority of work.

[7] Based on Theorem 8, we are also free to recurse with *IpScShuf* instead of using *Fisher-Yates Shuffle*.

4.1 Parallelizing RoughScatter

At heart, the parallel *ParRoughScatter* runs the sequential *RoughScatter* concurrently on independent subproblems. To this end, we exploit that *RoughScatter* allows arbitrary gaps *between* buckets. Secondly, we can freely pause and resume after each assignment without additional overhead since the algorithm's state is fully captured by the buckets' pointer triples and the partition of the placed elements.

Analogously to Section 3.3, we first split the input into k buckets of roughly equal size. In order to fork, we further split each bucket into two, and assign either half to one subtask. Then each subtask either recursively continues splitting, or, if the subproblem is sufficiently small, runs the sequential *RoughScatter*. After both subtasks join, we merge the two halves of each bucket. This involves only operations on the buckets' pointers and swapping the staged items of the first half to the second half. Additionally observe that the first subtask ends if there exists a filled bucket $B_i^{(1)}$, and analogously $B_j^{(2)}$ for the second subtask. Only with probability $1/k$, we have $i = j$, and thus, the merged bucket B_j is full. Otherwise, all merged buckets contain at least one staged item and we continue executing *RoughScatter*.

▶ **Observation 9.** *Since ParRoughScatter applies RoughScatter after each join, the number R of remaining staged items according to Lemma 2 also holds for ParRoughScatter.*

▶ **Lemma 10.** *For $n \gg k \log^3 k$ and $k^2 = \mathcal{O}(n)$, whp ParRoughScatter has $\mathcal{O}\big(\sqrt{nk \log k}\big)$ parallel depth and needs $\mathcal{O}(n)$ work.*

Proof. Splitting k buckets into $2k$ takes $\mathcal{O}(k)$ time. By Observation 9, Lemma 2 bounds the number of staged items received from both subtasks to $\mathcal{O}\big(\sqrt{nk \log k}\big)$. This bounds from above the time required to swap elements during merging, as well as the time to run *RoughScatter* on the remaining staged elements after merging. To meet the prerequisites of Lemma 2, we choose a base case size of $N_0 = k^2$ and process smaller subproblem sequentially in time $\mathcal{O}(N_0)$. This leads to the following bound on the parallel depth $D(n)$:

$$D(n) = \begin{cases} D(n/2) + \mathcal{O}\big(k + \sqrt{nk \log k}\big) & \text{if } n \geq N_0 \\ \mathcal{O}(N_0) & \text{if } n < N_0 \end{cases} \tag{3}$$

$$= \mathcal{O}\Big(N_0 + \log(n/N_0)k + \sqrt{nk \log k}\Big) = \mathcal{O}\Big(\sqrt{nk \log k}\Big) \tag{4}$$

Analogously, we bound the work using the Master Theorem [5] for the following recursion:

$$W(n) = \begin{cases} 2W(n/2) + \mathcal{O}\big(k + \sqrt{nk \log k}\big) & \text{if } n \geq N_0 \\ \mathcal{O}(N_0) & \text{if } n < N_0 \end{cases} = \mathcal{O}(n)$$ ◀

4.2 Parallelizing FineScatter

As we discuss in Section 6.4, in practice, it is not necessary to parallelize *FineScatter* due to its negligible impact on the total runtime. Thus, in the following, we sketch just enough adoptions to reduce the parallel depth of *FineScatter* to that of *ParRoughScatter*. By Observation 9, the analysis of *FineScatter* in Section 3.4 remains valid after the execution of *ParRoughScatter*. By comparing with Lemma 10, we find that the parallel depth *ParRoughScatter* dominates all sequential operations but *TwoSweep*.

Recall that *TwoSweep* shifts the boundaries of buckets to match the final bucket sizes in time $\mathcal{O}\big(k\sqrt{nk \log k}\big)$ whp. Thus, we need to shave off only a factor of $\Theta(k)$ which is straightforward using standard parallelization techniques based on the following observation:

The prefix sum D_i defined in Lemma 5 can be interpreted as the number of elements that the end of bucket B_i needs to be shifted. Thus, after computing $(D_i)_i$ and placing one worker per bucket, each worker can shift elements in the appropriate direction. To shift items between distant buckets, we run $\mathcal{O}(k)$ rounds. The time per round is dominated by the largest swap of items over any bucket boundary which, in turn, is upper bounded by the maximal deviation $\mathcal{O}\left(\sqrt{n/k \log k}\right)$ whp in the first round. Thus, *TwoSweep* can be naïvely parallelized with a parallel depth of $\mathcal{O}(\sqrt{nk \log k})$ matching that of *ParRoughScatter*. This results in a trivial upper bound of work of $\mathcal{O}(k\sqrt{nk \log k})$ matching the overestimation of Corollary 6 used for the sequential *TwoSweep*.

▶ **Theorem 11.** *PIpScShuf has (whp) parallel depth* $\mathcal{O}\left(\sqrt{nk/\log k}\log(n)\right)$*, uses* $\mathcal{O}(n \log_k(n))$ *work and* $\mathcal{O}(k[\log_k(n) + \mathfrak{P}])$ *words of memory where* $\mathfrak{P}$ *is the number of parallel subtasks.*

Proof. The proof is analogous to the proof of Theorem 8 by replacing the complexity measures of *IpSc* with Lemma 10 followed by symbolic simplifications. The memory bound additionally accounts for k bucket pointer triples per subtask. ◀

5 Implementation

Our implementations use `Rust`, a programming language with strong memory safety and parallelism guarantees. While the code repository contains a number of prototypes, we consider the publicly exposed algorithms, such as *IpScShuf* (`seq_shuffle`) and *PIpScShuf* (`par_shuffle`), ready to be used in other projects. To monitor the code quality, we rely on the strong static analysis tools and dynamic checks available in the Rust ecosystem. We also use more than 80 tests that include statistical tests of the uniformity of the produced permutations (e.g., the 1 and 2-independence of the output ranks).

PIpScShuf uses the work-stealing scheduler included in the `rayon`[8] crate. We exclusively use binary Fork-Join parallelism by means of the `rayon::join` function which requires no heap allocations after the worker pool was once initialized. Given the widespread usage of `rayon`, it is very likely that the calling application already set up this pool. Then, none of our algorithms cause any heap memory allocation (thereby avoiding potential error sources or hidden synchronizations). A `rayon::join` incurs very little cost if both tasks are executed on the same worker. Hence, we regularly define more than 2^{11} parallel subtasks – allowing fine-grained work-balancing. In case of a compatible pseudo-random number generator (requires `rand::SeedableRNG` trait), we use a deterministic sequence to derive the subtasks' generators from the provided generator. Then, two runs from the same state yield the same permutation despite non-deterministic scheduling; this optional reproducibility can be crucial (e.g., for debugging of the embedding code).

Based on Figure 11 (Appendix), we empirically optimized the number of buckets k as $k = 64$. The vast majority of code is implemented in the *safe* subset of Rust. In Section 6, we use a highly optimized implementation of *RoughScatter* that requires pointer arithmetic and memory accesses without explicit boundary checks which is considered *unsafe* in Rust. While the memory safety guarantees of these sections are "only" comparable with C/C++, as an implementor it is easier to reason about these small code segments (as opposed to the whole program) and to check assumptions during runtime.

[8] `https://github.com/rayon-rs/rayon`

On the `x86` platform, the code executes $2\lfloor 64/\lceil\log_2(k)\rceil\rfloor$ (i.e. 32 for $k = 16$ and 20 for $k = 64$) random assignments without any branching instructions. This allows a high utilization of the CPU's pipeline which is further increased by explicitly prefetching the memory locations. Further, instead of using a standard `swap`(x, y) with three move operations (namely $t \leftarrow x$, $x \leftarrow y$, $y \leftarrow t$ where t is a temporary storage), we use two temporary variables resulting in $2 + \epsilon$ data movements per assignment. The resulting assignment process is at least five times faster than any weighted sampling strategy we experimented with (including fast implementations of [21] and various rejection schemes in spirit of [6]).

IpScShuf and *PIpScShuf* use a *FY* implementation for instances below 2^{18} items that resorts to 32 bit arithmetic and often produces two indices from one random 64 bit word.

The repository includes highly optimized sequential and parallel reimplementations of *MergeShuffle* which include similar techniques as above. Their performance is incomparable with the original implementation[9] which, on the one hand, includes handcrafted assembly code for merging, but, on the other hand, uses the `rdrand` instruction [17] to acquire random bits; depending on the specific processor, `rdrand` one to two orders of magnitude slower than `Pcg64Mcg`. [12, 28] Further, both choices are highly non-portable. Overall our portable implementation using `Pcg64Mcg` is faster than the original. Observe that *MergeShuffle* has linear parallel depth since only independent subproblems are executed in parallel while the merging of the first recursion layer is purely sequential.

6 Empirical evaluation

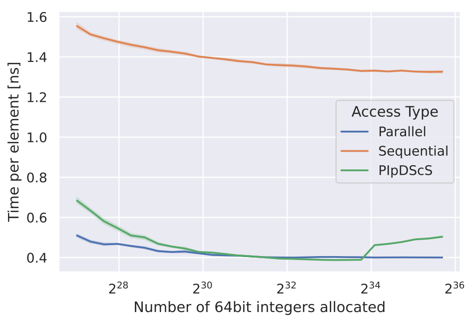
In this section, we investigate the performance of multiple shuffling algorithms for diverse settings. If not stated differently, we use the following standard parameters:

- Measurements are collected on a machine with an *AMD EPYC 7702P* CPU (64 cores/128 hardware threads), 512 GB RAM, running *Ubuntu 20.04*, `rustc 1.71` and `gcc-10`, using release builds (`cargo -release`, `g++ -O3`) without machine-specific optimizations. Further, we consider nine more machines with quite different configurations in Section 6.6.
- Experiments focus on the fast pseudo-random number generator `Pcg64Mcg`, as this choice exposes overheads in the shuffling algorithms rather in the generators itself. In Figure 8 (Appendix), we report the performance for different generators. The relative performance of algorithms remains qualitatively similar for different randomness sources, though *IpScShuf* and *PIpScShuf* are less affected by the generator choice as *FY* variants.
- Experiments focus on 64 bit integers, which seems to be a typical index size in data sets of several 100 GiB. As indicated in Figure 9 (Appendix), the throughput (measured in bytes per second) increases for larger elements since the per-element overhead shrinks. Again, *IpScShuf* and *PIpScShuf* exhibit a smaller spread than *FY* variants.
- All performance measurements reported are the mean of at least five runs. To reduce systematic errors, an individual run is the average of N repetitions where N is chosen such that the measured time exceeds 100 ms. Consecutive repetitions use different locations in a larger memory region to simulate a *cold start* where the input is not already cached.

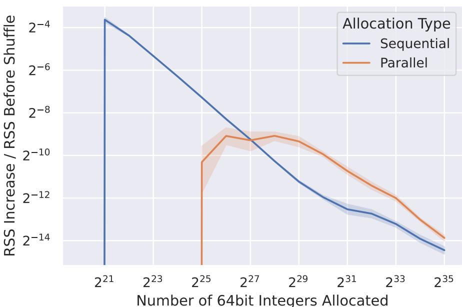
6.1 Memory usage and allocation costs

One important motivation for this work is the runtime cost of allocating large amounts of memory which we quantify in Figure 3a. For each measurement, we obtain a certain amount of data using the low level `libc::malloc` instruction, initialize it, and then return the data

[9] `https://github.com/axel-bacher/mergeshuffle`



(a) Runtime cost of memory allocation.



(b) Memory usage of *PIpScShuf* implementation.

Figure 3 Measurements of memory runtime costs and memory usage as described in Section 6.1.

using `libc::free`. For large volumes, `malloc` requests the operating system to map a certain memory size into virtual memory. Critically, the memory will only be backed by physical memory if it is actually accessed. For this reason, we initialize the data twice, and subtract the second round from the first one. The difference between both runs is the time it takes the system to provide the physical memory (without initializing it). Our measurements also suggest that writing the values in parallel does not scale well – despite an investment of 128 threads, we observe only a speedup of 4.9 for the initialization which is reduced to 3.1 for the whole process since `malloc` and `free` are sequential. For reference, we included the runtime of *PIpScShuf* and find that shuffling the data in parallel takes roughly as long as to acquire the memory needed to store a copy – without copying it.

In Figure 3b, we report the effective memory usage of *PIpScShuf*. We start a dedicated process for each run and measure the *maximal resident set size* (RSS) of the process (i.e. the maximal amount of memory that was physically backed at any time during the execution). We measure the RSS before and after the invocation of *PIpScShuf* and report the relative growth. As already discussed in Section 5, `rayon`'s worker pool needs to be initialized once. If we allocated the data sequentially, *PIpScShuf* implicitly sets up a pool resulting in 1631 heap operations to reserve a total of 923 KiB. After a parallel allocation, on the other hand, no heap operations are carried out. Even then, we observe a small increase of the RSS for large inputs – this seems to be caused by growing stack memory of the 128 active threads. In this case, the largest observed growth is 0.2 % and diminishes for very large inputs.

6.2 Performance overview

In Figure 4, we report the performance of several shuffling implementations. For each run, we set a timeout of 30 s and stop a graph after its algorithm hit said budget. The only exception is our *PIpScShuf* implementation with a runtime of 20.8 s for the largest data point. From a pool of various *Fisher-Yates Shuffle* implementations (see Figure 10, Appendix), we only include our own variant which strictly outperforms all competitors in the relevant regime.

The two fastest algorithms are `parallel::shuffle` (a C++ implementation of *ScatterShuffle* included in `stdlibc++`) and our *PIpScShuf*. For relatively small inputs *ScatterShuffle* is faster than *PIpScShuf* which takes the lead for inputs larger than 256 MiB.

All algorithms exhibit deteriorating throughput for larger inputs. This is remarkable for *FY* derivatives which have a predicted linear runtime. Their slowdown can be attributed to cache misses and related effects of the memory hierarchy. In Figure 10 (Appendix), we

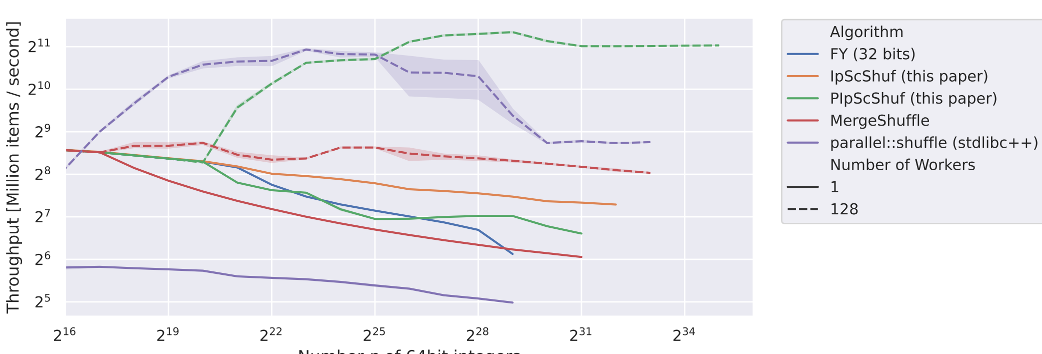


Figure 4 Performance of several shuffling algorithms with a time budget of 30 s per run. *FY*, *IpScShuf*, and *PIpScShuf* are our own implementations. `std::shuffle` and `parallel:shuffle` are implemented in C++. For parallel algorithms, we indicate the number of cores available as p.

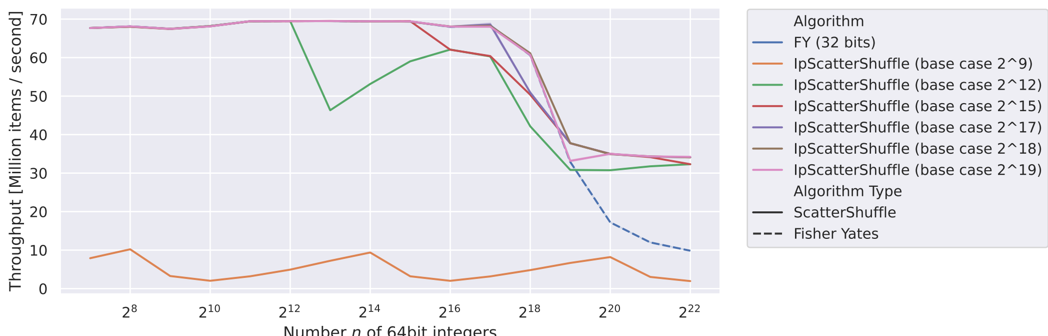


Figure 5 Performance of several sequential shuffling algorithms run in parallel on different data.

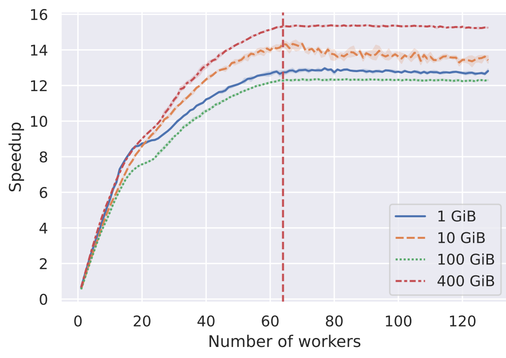
demonstrate that memory latency can be hidden by explicitly prefetching memory locations ahead of time.[10] However, prefetching only helps to simulate a slightly larger cache and we still observe a significant performance drop around 2 GiB. As predicted, all implementations based on *ScatterShuffle* exhibit a $\log_k(n)$ dependency in their runtime; this is especially visible for *PIpScShuf* when executed on a single core.

Due to incompatible libraries, we were unable to include fair performance measurements of [15] in our campaign. However, Figure 7 reports a higher throughput for *PIpScShuf* on a quad-core laptop (i7-8550U) than [15, Table 6] for a quad-socket server (E7-8867 v4) with 72 cores (while *PIpScShuf* uses less memory and includes the computation of random bits).

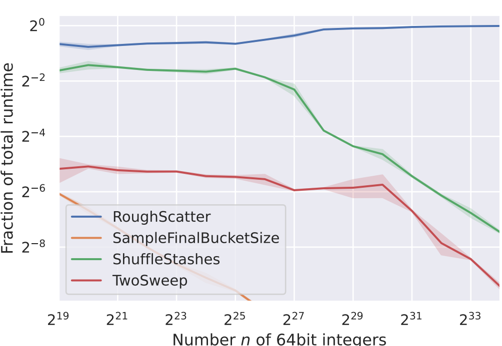
6.3 Parallel execution of sequential algorithms

When selecting an appropriate base case algorithm for *PIpScShuf*, Figures 4 and 10 can be misleading as they report the performance of sequential algorithms executed in isolation. In this setting, the studied algorithm has more resources at its disposal compared to the case where several instances are executed in parallel on independent data. This might also be relevant in different scenarios, e.g., if computational resources are shared by different users.

[10] We use a ring buffer to generate and prefetch random indices 16 swaps prior to the actual swap.



(a) Strong scaling of *PIpScShuf*. The vertical line indicates the number of physical cores.



(b) Fraction of runtime of the first recursion layer of *PIpScShuf* with 128 cores.

Figure 6 Parallel performance of *Parallel In-Place ScatterShuffle*.

Figure 5 is recorded similarly to Figure 4 with the difference that we execute 128 independent tasks in parallel and report the mean of their individual runtime as a single run. To avoid scheduling artifacts, we discard and repeat any run in which the wall-time of the experiment (i.e. from the start of the first thread to the termination of the last) is 20 % larger than the mean runtime of the individual tasks.

In this setting, we observe that memory becomes the dominating bottleneck; with minor exceptions (such as too small base case sizes), all algorithms exhibit roughly the same performance for instances below 1 MiB (the CPU has 256 MiB L3 cache that is now shared among 128 threads). For larger instances, *IpScShuf* variants are more than 3 times faster than the *FY* variants. Also observe that the contributions of implementation details, such as prefetching or base case size, pale in comparison the importance of memory locality.

6.4 Parallel scaling

To quantify the parallel speedup of *PIpScShuf*, we carry out a strong scaling experiment as follows. For fixed input sizes, we execute *IpScShuf* and the fastest *FY* implementation as base lines and then profile *PIpScShuf* for an increasing number of workers. In Figure 6a, we report the parallel speedup over the fast sequential implementation (*IpScShuf* in all cases).

For data set sizes of 10 GiB and larger, the speedup over the fastest sequential solution approaches up to 16. The self speedup is larger (e.g., 22.9 for 400 GiB), indicating a good scalability that is somewhat offset by the additional overhead of the parallel implementation. For the same instance, *PIpScShuf* is 141 times faster than *Fisher-Yates Shuffle*.

We see a substantial increase in speed until the number of workers matches the number of physical cores; using virtual cores (simultaneous multi-threading) has little impact. This is to be expected since shuffling is memory-bound and virtual cores primarily help to saturate arithmetic units of super-scalar processors, but do not affect memory performance.

6.5 Relative performance of subproblems

When designing, implementing, and analyzing *IpScShuf* and *PIpScShuf* we focused on the fast opportunistic *RoughScatter* which then requires the additional *FineScatter* post-processing to deal with the few remaining items. While we heavily optimized *RoughScatter*, we opted for simple and easy to implement solutions in *FineScatter*.

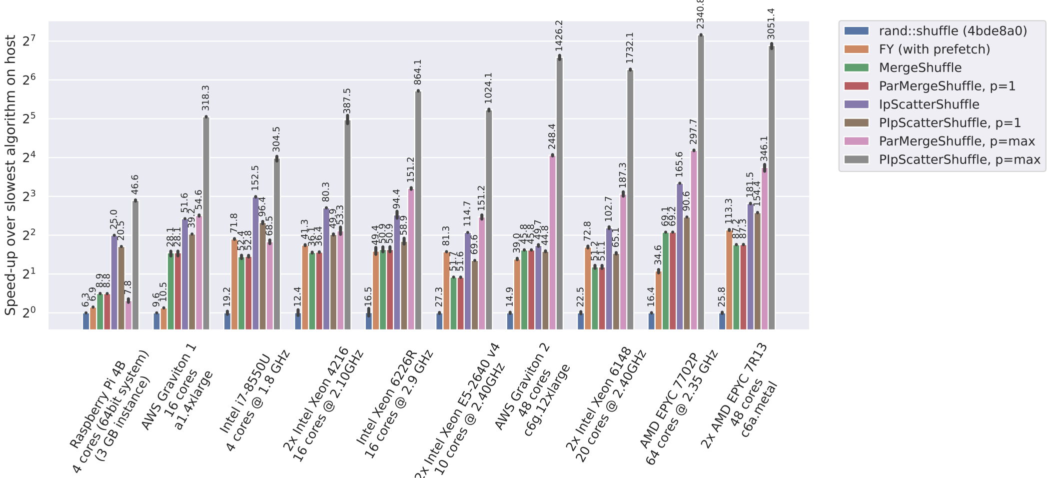


Figure 7 Performance of selected algorithms on different computers shuffling 64 bit integers. The length of a bar corresponds to the speedup compared to the slowest algorithm on that system. The numbers above a bar indicate the absolute through-put in million elements per second.

To empirically support this design decision, we measure the runtime of the first recursion layer of *PIpScShuf* for a wide range of input sizes. In Figure 6b, we then report the relative wall time of the four sub-algorithms that constitute said layer. In our implementation only *RoughScatter* is executed in parallel with 128 workers available while the remaining parts are sequential algorithms. Despite the asymmetry in available workers, we observe that for $n \geq 2^{27}$ *ParRoughScatter* accounts for more than 90 % (99 % for $n \geq 2^{33}$) of the runtime. This supports our design decisions since optimizing *FineScatter* leads to diminishing results.

6.6 Performance on different machines

To verify that our empirical findings are representative for modern computers, we quantify the performance of shuffling on different machines in Figure 7. The machines range from a single-board computer, over a laptop, to dual-socket servers (covering more than two orders in magnitude in purchase price). They use different instruction sets, micro-architectures, processor manufactures, and core counts. Their configurations are specified in the figure. We reiterate that no machine-specific optimizations are used; in fact, exactly two binaries were used (for ARM and x86, respectively).

To accommodate most systems, we selected an instance size of 10 GiB with the exception of the Raspberry PI 4B which features only 4 GiB of main memory. The measurements consist of runs of sequential algorithms, sequential runs of parallel algorithms (indicated by $p = 1$), and parallel runs with one worker per hardware thread (indicated by $p = \max$). The maximal throughput of the fastest system is 50 times higher than that of the slowest system. In all cases `rand::shuffle` is the slowest contender, *IpScShuf* the fastest sequential implementation, and *PIpScShuf* the overall fastest solution.

References

1 Daniel Allendorf, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Nick Wormald. Engineering uniform sampling of graphs with a prescribed power-law degree sequence. In *ALENEX*, pages 27–40. SIAM, 2022.

2 Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms. *ACM Trans. Parallel Comput.*, 9(1):2:1–2:62, 2022.

3 Axel Bacher, Olivier Bodini, Alexandros Hollender, and Jérémie O. Lumbroso. Mergeshuffle: a very fast, parallel random permutation algorithm. In *GASCom*, volume 2113 of *CEUR Workshop Proceedings*, pages 43–52. CEUR-WS.org, 2018.

4 Edward A. Bender and E. Rodney Canfield. The asymptotic number of labeled graphs with given degree sequences. *J. Comb. Theory, Ser. A*, 24(3):296–307, 1978.

5 Jon Louis Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, 1980.

6 Petra Berenbrink, David Hammer, Dominik Kaaser, Ulrich Meyer, Manuel Penschuck, and Hung Tran. Simulating population protocols in sub-constant time per interaction. In *ESA*, volume 173 of *LIPIcs*, pages 16:1–16:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

7 Guojing Cong and David A. Bader. An empirical analysis of parallel random permutation algorithms ON smps. In *PDCS*, pages 27–34. ISCA, 2005.

8 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

9 Luc Devroye. *Non-Uniform Random Variate Generation.* Springer, 1986.

10 Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

11 Thomas S. Ferguson. Who solved the secretary problem? *Stat. Sci.*, 4(3):282–289, 1989.

12 Agner Fog. Instruction tables. URL: `https://www.agner.org/optimize/instruction_tables.pdf`.

13 Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019.

14 Hermann Gruber, Markus Holzer, and Oliver Ruepp. Sorting the slow way: An analysis of perversely awful randomized sorting algorithms. In *FUN*, volume 4475 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2007.

15 Yan Gu, Omar Obeya, and Julian Shun. Parallel in-place algorithms: Theory and practice. In *APOCS*, pages 114–128. SIAM, 2021.

16 Chris Hinrichs, Vamsi K Ithapu, Qinyuan Sun, Sterling C Johnson, and Vikas Singh. Speeding up permutation testing in neuroimaging. In C. J. C. Burges et al., editor, *Advances in Neural Information Processing Systems*, volume 26, pages 890–898. Curran Associates, Inc., 2013.

17 Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, 2022.

18 Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition.* Addison-Wesley, 1981.

19 Charles E. Leiserson. Programming irregular parallel applications in cilk. In *IRREGULAR*, volume 1253 of *Lecture Notes in Computer Science*, pages 61–71. Springer, 1997.

20 Daniel Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1):3:1–3:12, 2019.

21 Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. Dynamic generation of discrete random variates. *Theory Comput. Syst.*, 36(4):329–358, 2003.

22 Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. Engineering radix sort. *Comput. Syst.*, 6(1):5–27, 1993.

23 SJ Meyer. A failure of structured programming, Zilog Corp. Technical report, Software Dept. Technical Report, 1979.

24 Richard Nixon. Executive order 11497 — amending the selective service regulations to prescribe random selection, 1969.

25 Melissa E. O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, September 2014.

26 Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in scalable network generation. *CoRR*, abs/2003.00736, 2020.

27 Martin Raab and Angelika Steger. “Balls into bins” – A simple and tight analysis. In *RANDOM*, volume 1518 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 1998.

28 Matthew Route. Radio-flaring ultracool dwarf population synthesis. *The Astrophysical Journal*, 845(1):66, August 2017. `doi:10.3847/1538-4357/aa7ede`.

29 Peter Sanders. Random permutations on distributed, external and hierarchical memory. *Inf. Process. Lett.*, 67(6):305–309, 1998.

30 Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *SODA*, pages 431–448. SIAM, 2015.

31 Johannes Singler and Benjamin Konsik. The GNU libstdc++ parallel mode: software engineering considerations. In *IWMSE@ICSE*, pages 15–22. ACM, 2008.

A Additional measurements

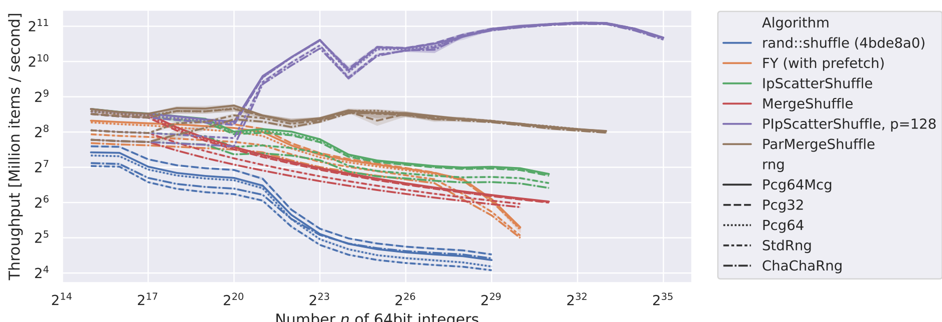


Figure 8 Performance of selected algorithms with different pseudo-random number generators.

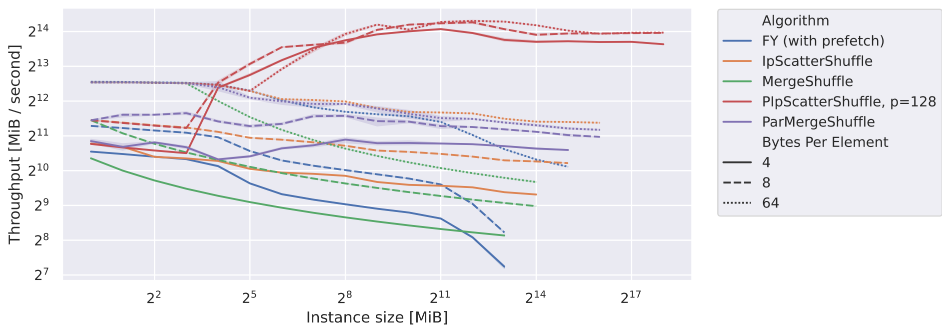


Figure 9 Performance of selected algorithms with different element sizes.

B Quantifying the hidden constants

In Lemma 2, we bound the number $R \leq \sqrt{2nk \log k}$ of items that remain staged after *RoughScatter* whp. Additionally, in Corollary 6, we bounded the complexity of *TwoSweep* by $\mathcal{O}\big(k\sqrt{nk \log k}\big)$. To provide empirical evidence and study the hidden constants, we simulate both processes. In Figure 12, we report the mean over 1000 independent runs divided by $\sqrt{2nk \log k}$ and $k\sqrt{nk \log k}$ respectively. Recall that our implementations use $k = 16$ and $k = 64$; we additionally simulated $k = 2^{16}$ as an accommodating upper bound for the foreseeable future. The small growth in k visible in Figure 12 is due to the small k values. Simulations with k up to 2^{28} agree with Lemma 2 and approach a factor of 0.66 in Corollary 6.

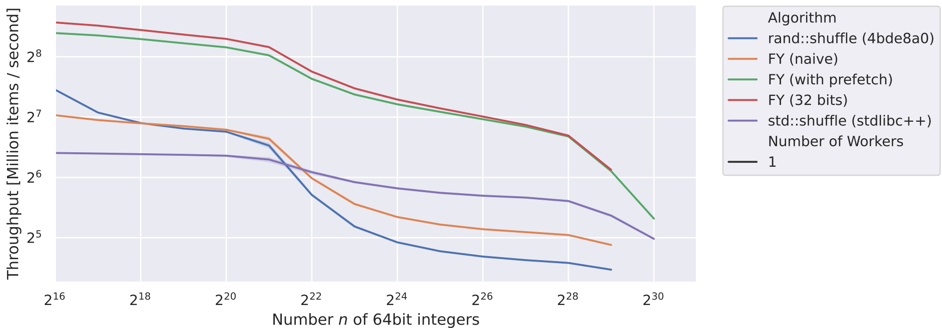


Figure 10 Performance of several Fisher-Yates implementations with a time budget of 30 s.

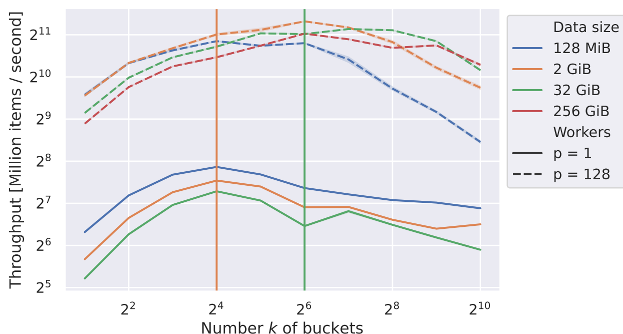


Figure 11 Performance of *In-Place ScatterShuffle* ($p = 1$) and *Parallel In-Place ScatterShuffle* ($p = 128$) for various data sizes as function of the number of buckets k. The two vertical lines correspond to default values of $k = 16$ for *IpScShuf* and $k = 64$ for *PIpScShuf*, respectively.

C Omitted proofs

▶ **Lemma 5.** *Let $N^f = (n_i^f)_i$ be the final bucket sizes, denote their deviation from the mean n/k as $d_i = n_i^f - n/k$, and let $D_i = \sum_{j=1}^{i} d_j$ be the inclusive prefix sum of deviations. Then, TwoSweep executes a total of $M(N^f) = \sum_i |D_i|$ swaps and takes time $\mathcal{O}\big(k + M(N^f)\big)$.*

Proof. Observe that a positive value d_i indicates that bucket B_i needs to receive d_i additional staged items from other buckets. Contrary, a negative value d_i means that bucket B_i has to give away $-d_i$ elements. The prefix sum D_i has an analogous meaning but accumulated over the first i buckets. This leads to the following cases:

1. If D_i is positive, buckets $B_1, \ldots, B_i$ have an excess of D_i items required somewhere in $B_{i+1}, \ldots, B_k$. These D_i items will be pushed to the right during the first sweep.
2. If D_i is negative, buckets $B_1, \ldots, B_i$ have a demand of $|D_i|$ items met by an excess somewhere in the buckets $B_{i+1}, \ldots, B_k$. Thus, B_i receives $|D_i|$ items in the second sweep.

In sum, bucket B_i is involved in $|D_i|$ swaps with its direct neighbors, leading to a total of $M(N^f)$ swaps and $\mathcal{O}\big(M(N^f) + k\big)$ work where the k accounts for per-bucket overheads. ◀

▶ **Corollary 6.** *TwoSweep takes time $\mathcal{O}\big(k\sqrt{nk \log k}\big)$ whp.*

Proof. We prove the claim based on Lemma 5 by establishing $\sum_i |D_i| = \mathcal{O}\big(k\sqrt{nk \log k}\big)$ (whp) where $D_i = \sum_{j=1}^{i} d_i$ is the prefix sum over the bucket size deviations from the mean. Observe that by construction, only elements that remain staged after the execution of

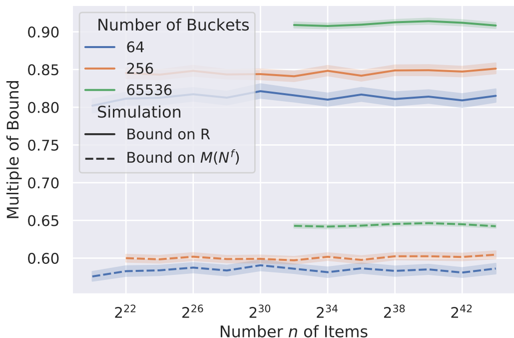


Figure 12 Simulation of Lemma 2 and Corollary 6.

RoughScatter can contribute and therefore $\sum_i |d_i| \leq 2R$ where $R \leq \sqrt{2nk \log k}$ (whp) by Lemma 2. Additionally, since the deviations balance over all buckets we have $\sum_i d_i = 0$. Thus, we trivially have that $\max_i |D_i| \leq R$.

We assume a worst case deviation where the first bucket needs to gain all R elements from the last bucket (or vice versa). While this is rather pessimistic (recall $\max_i \{|d_i|\} = \mathcal{O}\left(\sqrt{n/k \log k}\right)$ whp), it suffices to show the bound. In this instance, we have $|D_i| \leq R$ for all i and $\sum_{i=1}^{k} |D_i| \leq \sum_{i=1}^{k} R \leq kR = \mathcal{O}\left(k\sqrt{nk \log k}\right)$ (whp). ◀

Proxying Betweenness Centrality Rankings in Temporal Networks

Ruben Becker
Ca' Foscari University of Venice, Italy

Pierluigi Crescenzi
Gran Sasso Science Institute, L'Aquila, Italy

Antonio Cruciani
Gran Sasso Science Institute, L'Aquila, Italy

Bojana Kodric
Ca' Foscari University of Venice, Italy

Abstract

Identifying influential nodes in a network is arguably one of the most important tasks in graph mining and network analysis. A large variety of centrality measures, all aiming at correctly quantifying a node's importance in the network, have been formulated in the literature. One of the most cited ones is the *betweenness centrality*, formally introduced by Freeman (Sociometry, 1977). On the other hand, researchers have recently been very interested in capturing the dynamic nature of real-world networks by studying *temporal graphs*, rather than static ones. Clearly, centrality measures, including the betweenness centrality, have also been extended to temporal graphs. Buß et al. (KDD, 2020) gave algorithms to compute various notions of temporal betweenness centrality, including the perhaps most natural one – *shortest temporal betweenness*. Their algorithm computes centrality values of all nodes in time $O(n^3T^2)$, where n is the size of the network and T is the total number of time steps. For real-world networks, which easily contain tens of thousands of nodes, this complexity becomes prohibitive. Thus, it is reasonable to consider proxies for shortest temporal betweenness rankings that are more efficiently computed, and, therefore, allow for measuring the relative importance of nodes in very large temporal graphs. In this paper, we compare several such proxies on a diverse set of real-world networks. These proxies can be divided into *global* and *local* proxies. The considered global proxies include the exact algorithm for static betweenness (computed on the underlying graph), prefix foremost temporal betweenness of Buß et al., which is more efficiently computable than shortest temporal betweenness, and the recently introduced approximation approach of Santoro and Sarpe (WWW, 2022). As all of these global proxies are still expensive to compute on very large networks, we also turn to more efficiently computable local proxies. Here, we consider temporal versions of the ego-betweenness in the sense of Everett and Borgatti (Social Networks, 2005), standard degree notions, and a novel temporal degree notion termed the *pass-through degree*, that we introduce in this paper and which we consider to be one of our main contributions. We show that the pass-through degree, which measures the number of pairs of neighbors of a node that are temporally connected through it, can be computed in nearly linear time for all nodes in the network and we experimentally observe that it is surprisingly competitive as a proxy for shortest temporal betweenness.

2012 ACM Subject Classification Theory of computation → Shortest paths; Networks → Network algorithms; Mathematics of computing → Graph algorithms

Keywords and phrases node centrality, betweenness, temporal graphs, graph mining

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.6

Supplementary Material *Software (Source Code)*: `https://github.com/piluc/TSBProxy` archived at `swh:1:dir:02ece196b54034f669645bc2b220c28d43cbc423`

Funding *Pierluigi Crescenzi*: Partially supported by the French government through the UCAJEDI, while visiting COATI, INRIA d'Université Côte d'Azur, Sophia Antipolis, France.

© Ruben Becker, Pierluigi Crescenzi, Antonio Cruciani, and Bojana Kodric;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Centrality measures are notions for evaluating the importance of nodes in networks, used in network analysis and graph theory. The aim is to assign real values to all the nodes, in such a way that the values are monotonously dependent of the nodes' importance, i.e., more important nodes should have higher centrality values. It is evident that this task is among the most important ones in network analysis. Consequently, there is a vast variety of centrality notions in the existing literature. Popular measures include spectral notions, such as Katz's index [24], Seeley's index [41], and PageRank [10], and combinatorial notions, like the straightforward concept of degree centrality, the closeness centrality [4], the harmonic centrality [32, 6], and the betweenness centrality [18]. This diversity of notions indicates that there is no consensus among researchers on which notion is the "correct one". While Boldi and Vigna [6] provide an axiomatic approach to this question, their work mainly transfers the discord from which centrality notion to use to the question of which axioms to agree upon. In fact, the choice of which centrality notion to employ is mainly dependent on the application which may stem from a diverse set of fields [30, 17, 11]. In many scenarios, the considered networks are characterized by the following challenges: (1) they are very large and (2) they are dynamic or temporal, i.e., they change over time. In the context of these two challenges it is, thus, essential to consider temporal variants of the most important centrality notions, alongside algorithms for computing them, that have a good scaling behaviour. In this work, we focus on the *betweenness centrality*, which is certainly among the most used and most cited centrality notions, and study it in the context of these challenges.

Buß et al. [12] defined the *shortest temporal betweenness* as a temporal counterpart of the *betweenness centrality*, and gave an algorithm to compute all centrality values in time $O(n^3T^2)$, where n is the size of the network and T is the total number of time steps. For nowadays networks, such time complexity easily becomes infeasible. Thus, it is reasonable to consider proxies for shortest temporal betweenness rankings that are more efficiently computed. In this work, we use the following general approach. We employ a set of competitor algorithms that we each use as proxies for temporal betweenness rankings, i.e., for each algorithm, we compute a complete ranking of the nodes and evaluate how this ranking relates to the "correct" ranking. While different scenarios may exist, centrality values are frequently used to rank nodes and our proxy notion is motivated exactly by such applications.

Some of the considered proxies have the property that they still try to capture the global nature inherent in the definition of the shortest temporal betweenness and, as a consequence, still suffer from a comparatively bad running time, meaning that their running times are far from linear in the input size. Note however that, as argued, e.g., by Teng [47], in the age of Big Data, an algorithm should be considered efficient or scalable if its time complexity is nearly-linear. In fact, there is even theoretical evidence, in form of several conditional lower bound results [2, 7], for believing that no such algorithm is achievable, even for *approximately* computing the betweenness values in *static* graphs. We thus shift our focus away from these *global proxies* towards *local proxies* for shortest temporal betweenness rankings. We classify a proxy as local if the centrality values of nodes are completely determined by the induced subgraph of their neighborhood (including themselves).

For measuring proxy quality, we employ several different metrics, most prominently a weighted version of Kendall's τ correlation coefficient and the intersection of the top-k ranked nodes (for different values of k). Note that the latter is directly translatable into the Jaccard similarity of the top ranked nodes. We would like to stress here that it is quite uncomplicated to show that all proxies considered in the present work can perform arbitrarily bad on adversarial examples (in terms of all considered metrics) and no reasonable theoretical guarantees can therefore be given for their ranking quality.

A diverse set of temporal betweenness notions has been defined in the literature (see Section 1.2). Clearly, if the notion of centrality is already vague in static graphs, it becomes even more so in the temporal setting, where in addition the time dimension has to be considered. In this study, we focus on the definition given by Buß et al. [12] as it arguably represents one of the most "natural" and direct temporal analogues of the static betweenness (once the notion of distance has been defined).

1.1 Contribution

We compare a variety of approaches for proxying shortest temporal betweenness rankings in terms of their scalability and output quality. We start our study in Section 3 with a comparison of the following proxies: (1) exact algorithm for the static betweenness computed on the underlying graph, (2) the more efficiently computable prefix foremost temporal betweenness of Buß et al. and (3) the recently introduced (absolute) approximation approach of Santoro and Sarpe [39]. Our evaluation indicates that the static betweenness rankings turn out to be quite competitive, the performance of the prefix foremost temporal betweenness seems somewhat inconsistent, while the quality of the ranking returned by the considered temporal betweenness approximation algorithm very much depends on the provided time.

Next, motivated by the fact that static degree centrality is often compared to other centrality measures, we follow this approach in the temporal setting. In Section 4, we describe our main theoretical contribution: the *pass-through degree*, a new *temporal* degree notion which we believe to be interesting in its own right. Informally the pass-through degree of a node v measures the number of neighbor pairs of v that are temporally connected through v, i.e., that have a temporal path of length two between them that *passes through* v. We proceed by giving an algorithm that computes the pass-through degree of all nodes in a given (directed) temporal graph in $O(M \log m)$ time, where M is the number of temporal arcs and m the number of arcs in the underlying static graph. In other words, the proposed algorithm is scalable in the sense of Teng [47].

In Section 5 we compare the following set of local proxies in terms of their efficiency and quality: (1) temporal versions of the ego-betweenness in the sense of Everett and Borgatti [16], which entails to compute the betweenness centrality values of the nodes in their respective ego-networks (the induced subgraph of a node's neighborhood including himself) (2) the pass-through degree, and (3) the approximation algorithm for temporal betweenness centrality also used as one of the global proxies in Section 3, as it is the only choice from that section that offers scalability in terms of computation time. We note that the pass-through degree falls somewhere between the simple degree notions and the ego-betweenness notion in terms of complexity. Our evaluation here indicates that the ego-networks can be of comparable size as the whole network and, thus, prohibitively large on some data sets, the pass-through degree usually does not perform worse than the ego-variants and is at the same time much faster, while the considered approximation algorithm for temporal betweenness has a more inconsistent performance over different data sets.

Our experimental evaluation is based on a diverse set of real-world networks that includes almost all publicly available networks from the works of Buß et al. [12] and Santoro and Sarpe [39]. We did not include the `Karlsruhe` network [20] (used in [12]) because it does not appear to be available anymore. Moreover, we replaced `Mathoverflow` [29] network (used in [39]) by a bigger temporal network from a different domain to make the set of analyzed temporal graphs more diverse. Finally, we excluded `Ask Ubuntu` and `Super User` [29] (also analyzed in [39]) because of the excessive amount of time needed to compute their exact temporal betweenness rankings.

1.2 Further Related Work

The literature on centrality measures being vast, we restrict our attention to approaches that are closest to ours. We, thus, particularly focus on centrality notions in temporal graphs.

First of all, several works give introductions to temporal graphs that include surveys on temporal centrality measures (see, e.g., [23, 28, 40]). Nicosia et al. [33] introduced different temporal graph notions, such as temporal centralities, temporal motif, temporal clustering, temporal modularity, and temporal communities. Providing top-k algorithms for estimating temporal closeness centrality has also already been treated in the literature [15, 34]. Subsequently, a closeness variant based on bounded random-walks, related to the concept of influence spreading, has been proposed by Haddadan et al. [22]. Furthermore, Tang et al. [46] introduced temporal variants of both closeness and betweenness centrality based on foremost temporal paths, and experimentally showed the effectiveness of such metrics in spotting influential users in real-world temporal graphs. Building upon this direction, Tang et al. [44] used the notion of temporal closeness to provide an empirical analysis of the containment of malware in real-world mobile phone networks. The Katz centrality [24] has been adapted to the temporal setting [21, 5] as well, while Rozenshtein et al. [37] defined the temporal PageRank by replacing random walks with temporal random walks.

Tsalouchidou et al. [48] extended the well-known Brandes algorithm [9] to allow for distributed computation of betweenness in temporal graphs. Specifically, they studied shortest-fastest paths, considering the bi-objective of shortest length and shortest duration. Buß et al. [12] analysed the temporal betweenness centrality considering several temporal path optimality criteria, such as shortest (foremost), foremost, fastest, and prefix-foremost, along with their computational complexities. They showed that, when considering paths with increasing time labels, the foremost and fastest temporal betweenness variants are $\#P$-hard, while the shortest and shortest foremost ones can be computed in $O(n^3T^2)$, and the prefix-foremost one in $O(nM \log M)$. Here n is the number of nodes and M the number of temporal arcs. The complexity analysis of these measures has been further refined since [38].

Santoro and Sarpe [39] provide a sampling-based approximation algorithm for estimating the temporal betweenness centrality of nodes based on shortest path criterion, for situations in which the computational cost of computing exact values is too large.

Ghanem et al. [19] defined a temporal version of ego betweenness based on most recent paths, which are paths that give the most recent information to the destination vertex about the status of the source, i.e., no other path starts from the source at a later point in time. Their definition of temporal ego betweenness is snapshot based, i.e., it gives the ego betweenness of the temporal ego graph at a specific time instant. Simard et al. [42], on the other hand, studied a continuous-time scenario of the shortest paths betweenness.

Finally, Oettershagen et al. [35] defined a random temporal walks based centrality that quantifies the importance of a node by measuring its ability to obtain and distribute information in a temporal network. They provide exact and approximate algorithms for computing their centrality measures and compare it with the state-of-the-art temporal centralities, i.e., with PageRank [37], Katz [5], closeness [15, 34], and betweenness [12].

2 Preliminaries

We proceed by formally introducing the terminology and concepts that we use in what follows. For $k \in \mathbb{N}$, we let $[k] := \{1, \ldots, k\}$.

Static Graphs. We start by introducing standard *static*, i.e., non-temporal, graphs[1]. An *undirected graph* is an ordered pair $G = (V, E)$, where V is a set whose elements are called vertices or nodes, and E is a set of unordered pairs of vertices, whose elements are called edges. We denote by $N(u) = \{v \in V : \{u, v\} \in E\}$ the set of neighbors of a vertex $u \in V$. The *degree* of a vertex $u \in V$ is defined as $\mathrm{d}(u) := |N(u)|$. A *directed graph* is a an ordered pair $G = (V, A)$, where V is a set whose elements are called vertices or nodes, and A is a set of ordered pairs of vertices, whose elements are called arcs. We denote by $N^{\text{in}}(u) = \{v \in V : (v, u) \in A\}$ and $N^{\text{out}}(u) = \{w \in V : (u, w) \in A\}$ the set of *in-neighbors* and of *out-neighbors* of a vertex $u \in V$, respectively. For a subset of nodes $U \subseteq V$, we call $G[U] := (U, A')$, where $A' := \{(u, v) \in A : u, v \in U\}$, the *induced subgraph* of U. We note that an undirected graph can be modeled as a directed graph by introducing, for every edge $e = \{u, v\} \in E$, both arcs (u, v) and (v, u), resulting in the corresponding bidirected graph. We thus focus on directed graphs in what follows.

Temporal Graphs. A directed *temporal graph* is an ordered triple $\mathcal{G} = (V, A, \lambda)$, where (V, A) is a directed graph, called the *underlying graph* of the temporal graph $\mathcal{G}$, and $\lambda : A \to 2^{[T]}$ is a function assigning to every arc in A a finite set of elements from the set of *time labels* $[T]$.[2] We let $\mathcal{A} = \{(u, v, t) : (u, v) \in A, t \in \lambda(u, v)\}$ denote the set of *temporal arcs* of $\mathcal{G}$. Undirected temporal graphs can be modeled via directed graphs resulting in a bidirected underlying graph. Static graphs can be modeled by temporal graphs by defining $\lambda(a) := [T]$ for all arcs $a \in A$.

Temporal Betweenness. A *walk* from node u to node w in a static graph $G = (V, A)$ is a sequence $a_1, \ldots, a_k$ such that $a_i = (v_i, v_{i+1}) \in A$, $v_1 = u$, and $v_{k+1} = w$. We call k the *length* of the walk. A *path* is a walk such that $v_i \neq v_j$ for all $i, j \in [k]$ with $i \neq j$. A *shortest path* from u to w is a path of minimum length among all paths from u to w. We denote by $\sigma_{u,w}$ the total number of shortest paths between u and w in G, while $\sigma_{u,w}(v)$ is the number of shortest paths between u and w that pass through v. The *betweenness* or *betweenness centrality* of a node v in G, formally introduced by Freeman [18] in 1977, is defined as

$$\mathrm{b}_G(v) := \sum_{u,w \in V \setminus \{v\} : \sigma_{u,w} \neq 0} \frac{\sigma_{u,w}(v)}{\sigma_{u,w}}.$$

A *temporal walk* from node u to node w in a temporal graph $\mathcal{G} = (V, A, \lambda)$ is a walk $a_1, \ldots, a_k$ from u to w in the underlying graph $G = (V, A)$ such that there exist time labels $t_1, \ldots, t_k$ with $t_1 < \ldots < t_k$ and $t_i \in \lambda(a_i)$ for every $i \in [k]$. We call k the *length* of the temporal walk and t_k the *arrival time* of the walk at w. A *temporal path* is a temporal walk such that $v_i \neq v_j$ for all $i, j \in [k]$ with $i \neq j$. A *prefix temporal path* of a temporal path is its subpath starting at the same vertex. A *shortest temporal path* from u to w is a temporal path of minimum length among all temporal paths from u to w. Analogously to the static case, we denote by $\sigma^{\text{temp}}_{u,w}$ the total number of shortest temporal paths between u and w in $\mathcal{G}$, while $\sigma^{\text{temp}}_{u,w}(v)$ is the number of shortest temporal paths between u and w that pass through v. The *shortest temporal betweenness (centrality)* of a node v in the temporal graph $\mathcal{G}$ is defined as

[1] We use the terms "graph" and "network" interchangeably.
[2] The value T denotes the *life-time* of the temporal graph, and, without loss of generality for our purposes, we assume that, for any $t \in [T]$, there exists at least one temporal arc a such that $\lambda(a) = t$.

$$\text{stb}_{\mathcal{G}}(v) := \sum_{\substack{u,w \in V \setminus \{v\}: \\ \sigma^{\text{temp}}_{u,w} \neq 0}} \frac{\sigma^{\text{temp}}_{u,w}(v)}{\sigma^{\text{temp}}_{u,w}}.$$

Different notions of temporal betweenness were recently studied by Buß et al. [12]. Their foremost and fastest variants are both $\#P$-hard, making them very impractical. From the remaining variants, the shortest temporal betweenness seems to be the most natural one. We do not consider walk-based betweenness notions as we agree with Buß et al. that "paths are more suitable than walks for defining temporal betweenness centrality" [12]. Buß et al. [12] gave an algorithm to compute the shortest temporal betweenness of all nodes in time $O(n^3T^2)$. We will next introduce the notion of *prefix foremost temporal betweenness* from the work of Buß et al. as we will use it as a proxy for the shortest temporal betweenness. A *prefix foremost shortest path* from u to w is a shortest temporal path from u to w such that no other shortest temporal path has an earlier arrival time at w and such that its every prefix path satisfies the same property. Let $\tau^{\text{temp}}_{u,w}$ now be the total number of prefix foremost shortest paths between u and w in $\mathcal{G}$ and let $\tau^{\text{temp}}_{u,w}(v)$ be the number of those paths that pass through v. The prefix foremost temporal betweenness pftb of v is then defined analogously to the shortest temporal betweenness by replacing σ by τ , i.e.,

$$\text{pftb}_{\mathcal{G}}(v) := \sum_{\substack{u,w \in V \setminus \{v\}: \\ \tau^{\text{temp}}_{u,w} \neq 0}} \frac{\tau^{\text{temp}}_{u,w}(v)}{\tau^{\text{temp}}_{u,w}}.$$

Buß et al. give an algorithm for computing the prefix foremost temporal betweenness of all nodes in time $O(nM \log M)$, where n is the number of vertices and M the total number of temporal arcs.

Temporal Ego-Betweenness The *ego-network* $G^{[v]}$ of a node v in a static graph G is the induced subgraph of its in- and out-neighbors, i.e., $G^{[v]} := G[N^{\text{in}}(v) \cup N^{\text{out}}(v)]$. The *ego-betweenness (centrality)* of v is the betweenness of v in its ego-network, i.e., ego-b$(v) := \text{b}_{G^{[v]}}(v)$. The ego-betweenness (in undirected graphs) was introduced by Everett and Borgatti [16] as a more tractable variant of betweenness. We extend their ego-betweenness to temporal graphs as follows. The *ego-network* $\mathcal{G}^{[v]}$ of a node v in a temporal graph $\mathcal{G}$ is the temporal graph with the underlying graph $G^{[v]} := G[N^{\text{in}}(v) \cup N^{\text{out}}(v)]$ and with the time label function λ' being the restriction of λ to arcs in $G^{[v]}$. The *ego-shortest temporal betweenness* of v is the shortest temporal betweenness of v in its ego-network, i.e., ego-stb$(v) := \text{stb}_{\mathcal{G}^{[v]}}(v)$. Similarly, we define the *ego-prefix foremost temporal betweenness* of v as the prefix foremost temporal betweenness of v in its ego-network, i.e, ego-pftb$(v) := \text{pftb}_{\mathcal{G}^{[v]}}(v)$.

Everett and Borgatti [16] propose an algorithm to compute the ego-betweenness of a single node in an undirected static graph via computation of the square of the incidence matrix of the node's ego-network. We note that in the worst case the ego-network is of the same size as the original graph. For computing the temporal ego-betweennesses of all nodes, this algorithm can thus be implemented in time $O(n^{\omega+1})$, where ω is matrix multiplication exponent, i.e., the smallest real number such that two $n \times n$ matrices can be multiplied within $O(n^{\omega+\varepsilon})$ field operations for all $\varepsilon > 0$. The current best bound on ω is 2.3728596 [3].

3 Global Proxies for Shortest Temporal Betweenness

In this section, we summarize the results of our experimental study on proxying the shortest temporal betweenness values in large real-world networks using global proxies. Recall that a proxy is global, if the centrality value of each node is not purely dependent on its neighborhood. Our general experimental approach here is as follows. We employ a set of competitor algorithms that we each use as a proxy for shortest temporal betweenness centrality rankings. That is, for each algorithm, we compute a complete ranking of the nodes and evaluate (using various metrics) how this ranking relates to the "correct" ranking computed by the algorithm of Buß et al. [12]. In what follows, we will call this benchmark algorithm TempBrandes for "Temporal Brandes algorithm". Recall that TempBrandes computes the shortest temporal betweenness values of all nodes in time $O(n^3T^2)$.

3.1 Experimental Setting

Global Proxies. As global proxies for shortest temporal betweenness, our study includes the following algorithms.

Brandes: The classical algorithm of Brandes, which computes the static betweenness of all nodes in time $O(nm)$ on the underlying graph, i.e., the graph obtained by a union over all the time steps.[3]

Pref: The algorithm of Buß et al. [12] for computing the prefix foremost temporal betweennesses pftb in $O(nM \log M)$.

Onbra: The approximation algorithm of Santoro and Sarpe [39], which is a sampling technique for obtaining an absolute approximation of the shortest temporal betweenness values. The work that introduced this algorithm is rather vague in terms of how to choose the sample size, stating only that they choose it so as to make the algorithm run "within a fraction of the time required by the exact algorithm". In our study, we choose the number of samples such that the running time of Onbra is a tenth, a half and roughly equal to the running time of TempBrandes. We achieve this by first estimating the time taken per sample, and then computing the number of samples by dividing the (fraction of) time needed by TempBrandes with the computed estimate.

Besides Brandes, which is available in the *Graphs.jl* library, we implemented TempBrandes and all competitor algorithms in Julia. We chose to re-implement TempBrandes, Pref and Onbra because the available implementations of TempBrandes and Pref have issues with the number of paths in the tested networks, causing overflow errors (indicated by negative centralities). Since Onbra is based on the TempBrandes code, it results in the same errors. Our implementation uses a sparse matrix representation of the $n \times |T|$ table used in [12, 39], making the implemented algorithms space-efficient and allowing to compute the exact temporal shortest betweenness on big temporal graphs (for which the original version of the code gives out of memory errors). Furthermore, we noticed another error in TempBrandes and Pref, related to time relabeling causing an underestimation of centralities. Our code is available at `https://github.com/piluc/TSBProxy`.

Networks. We evaluate all of the above competitors on real-world temporal graphs of different nature, whose properties are summarized in Table 1. The networks come from two different domains.

[3] We are aware of fast approximation algorithms like Kadabra [8] for the computation of the static betweenness, but for our purpose here the efficiency of the exact algorithm is sufficient.

Table 1 The temporal networks used in our evaluation, where n denotes the number of nodes, m the number of arcs in the underlying static graph, M the number of temporal arcs, T the number of unique time labels, t_{STB} the execution time of TempBrandes, and n_e^{max} the maximum number of nodes in the ego network (type **D** stands for directed and **U** for undirected). The networks are sorted in increasing order with respect to t_{STB}.

Data set	n	m	M	T	t_{STB}	$n_e^{\max}$	Type	Source
`Hypertext 2009`	113	4392	41636	5246	263	99	U	[1]
`High school 2011`	126	3418	57078	5609	446	56	U	[1]
`Hospital ward`	75	2278	64848	9453	659	62	U	[1]
`College msg`	1899	20296	59798	58911	894	256	D	[29]
`Wiki elections`	7115	103680	106985	101012	1192	1066	D	[29]
`High school 2012`	180	4440	90094	11273	1345	57	U	[1]
`Digg reply`	30360	85247	86203	82641	1762	284	D	[36]
`Infectious`	10972	89034	831824	76944	4985	65	U	[1]
`Primary school`	242	16634	251546	3100	5607	135	U	[1]
`Facebook wall`	35817	104942	198028	194904	5751	89	D	[36]
`Slashdot reply`	51083	130370	139789	89862	8653	2916	D	[36]
`High school 2013`	327	11636	377016	7375	20642	88	U	[1]
`Topology`	16564	122140	198038	32823	22453	1401	U	[27]
`SMS`	44090	67190	544607	467838	25178	407	D	[36]
`Email EU`	986	24929	327336	207880	31840	346	D	[29]

Social networks: This domain includes most of the considered networks: `College msg`, `Wiki elections`, `Digg reply`, `Slashdot reply`, a subgraph of `Facebook wall` [50] containing the last $\sim 200k$ temporal arcs (as in the work of Santoro and Sarpe [39]), `SMS` and `Email EU`. These are social networks from different realms, where nodes correspond to users and temporal arcs indicate messages sent between them at specific points in time.

Contact networks: This domain includes the six networks `Hypertext 2009`, `High school 11/12/13`, `Hospital ward`, `Infectious`, `Primary school` and `Topology`. In the first case nodes correspond to individuals, while in the second case they correspond to computers. In both cases temporal arcs indicate a contact between nodes at a specific time.

In Appendix C we briefly discuss another type of temporal networks, that is, public transport networks (see, for example, [14, 15]), which, due to their topology, have quite peculiar properties in terms of both the execution times and of the quality of the analysed proxies.

Evaluation Details. We executed the experiments on a server running Ubuntu 20.04.5 LTS 112 with processors Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz and 112GB RAM. All the correlation coefficients were computed by making use of the corresponding functions available in the Python `scipy.stats` module [13].

3.2 Experimental Results

Experiment 1: Global Proxies' Correlation to TempBrandes

In our first experiment, we run both TempBrandes and all the discussed global proxies on the networks listed in Table 1. We then, for each of these four algorithms (TempBrandes plus three proxies), compute the resulting node ranking and evaluate the correlation of the rankings computed by the proxies with the ranking computed by TempBrandes. Here, we employ two different rank correlation measures, i.e., (1) a weighted version of Kendall's τ coefficient based on the work of Vigna [49], and (2) the number of common highest rank nodes among the first k. We also investigated the Spearman's ρ coefficient [43] and Kendall's τ coefficient [25] of the rankings, but we omit these results here due to space constraints. We, however, note that these measures indicated similar proxy performance as (1), and at the same time we find (1) more relevant, as it gives more importance to approximating the upper part of the ranking. For the weighted Kendall's τ coeffficient, we use a hyperbolic weighting scheme, as proposed by Vigna [49], that gives weights to the positions in the ranking which decay harmonically with the ranks, i.e., the weight of rank r is $1/(r+1)$. We refrain from comparing the proxies with respect to average correlation due to outliers.

Table 2 For each network, we show the execution times of TempBrandes and of all proxies (except for Onbra) in seconds. Dashes indicate that the experiment was interrupted after the time of TempBrandes elapsed. We omit Onbra from the table as its running time is fixed to approximately 1/10, 1/2, or 1 times the running time of TempBrandes due to the choice of the sample size.

Network	Execution Time (seconds)					
	TempBrandes	Brandes	Prefix	EgoPrefix	EgoSTB	PTD
Hypertext 2009	262.58	**0.01**	2.29	25.14	–	**0.01**
High school 2011	445.62	0.02	3.39	15.81	–	**0.01**
Hospital ward	659.13	**0.01**	2.01	37.97	–	**0.01**
College msg	894.44	1.12	21.58	4.83	116.53	**0.02**
Wiki elections	1192.42	6.52	49.84	45.54	586.75	**0.06**
High school 2012	1345.06	0.03	7.77	232.90	–	**0.01**
Digg reply	1762.09	123.37	224.58	1.61	4.43	**0.05**
Infectious	4985.19	3.28	50.26	26.97	820.73	**0.11**
Primary school	5607.17	0.08	39.22	492.73	–	**0.04**
Facebook wall	5750.73	349.01	429.38	2.00	17.86	**0.07**
Slashdot reply	8652.54	442.75	1116.99	7.08	38.78	**0.07**
High school 2013	20641.71	0.11	95.49	200.89	–	**0.09**
Topology	22452.98	124.98	1017.78	905.69	–	**0.08**
SMS	25178.27	129.53	591.98	4.18	476.71	**0.09**
Email EU	31839.72	0.54	180.86	411.07	–	**0.05**

Running Times. The running times of the global proxies can be found in the first three columns of Table 2. We note that Prefix always terminates in at most 15% of the running time of TempBrandes, while Brandes always finishes in at most 7% of the running time of TempBrandes. The efficiency of both proxies is particularly pronounced on contact networks with lots of temporal edges and comparatively few edges in the underlying graph. As a result the underlying graph is comparatively small, which is beneficial for Brandes.

On the other hand, the number of prefix foremost shortest paths is also much smaller than the total number of shortest temporal paths, which is beneficial for Prefix. The running times of the three Onbra versions are fixed to approximately 1/10, 1/2, and 1 times the running times of TempBrandes due to the choice of the sample size.

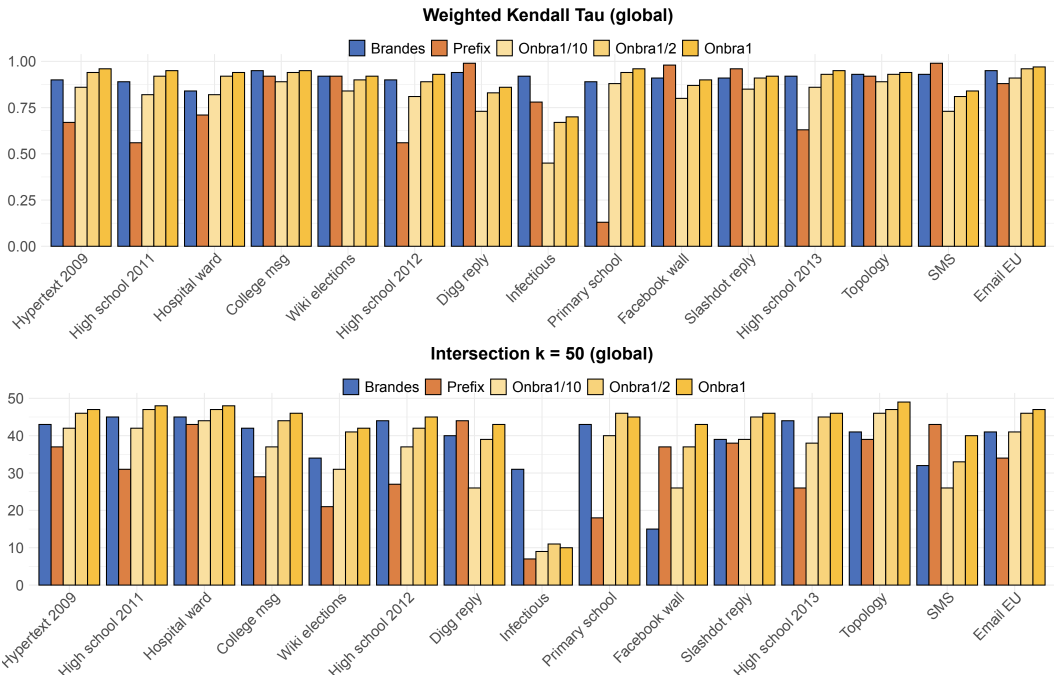


Figure 1 Comparison of the centrality ranking produced by TempBrandes and the rankings produced by the global proxies. The comparison is given in terms of the weighted Kendall's τ coefficient and the intersection of the top 50 nodes.

Ranking Correlation. An illustration of the ranking correlation results of this experiment can be found in Figure 1. On top of the figure, we show the Weighted Kendall's τ correlation of the rankings computed by the respective proxies and the ranking computed by TempBrandes. On the bottom, we show the results in terms of the intersection of the top-k nodes. We choose the value of k to be 50 here, while further results for $k = 1$ and $k = 25$ can be found in Table 5 in the appendix.

In terms of the weighted Kendall's τ correlation (see Table 3), we first observe that there are three (3) networks in which Brandes performs best, five (5) networks in which Prefix performs best, and ten (10) networks in which Onbra with maximal sample size performs best (we count networks with ties multiple times). We, however, also notice that the Onbra's performance heavily relies on the used sample size. Indeed, if the sample size is such that Onbra needs roughly 10% of TempBrandes running time, we observe that the numbers change as follows: there are eleven (11) networks in which Brandes achieves the best correlation and there are five (5) networks in which Prefix performs best, while Onbra never performs best.

As Brandes always terminates in less than 7% of TempBrandes' running time, and in most cases much faster, we can conclude that the static betweenness rankings are actually quite competitive in situations where we are restricted in terms of running time. In other words, it seems really necessary to give Onbra a running time similar to the exact algorithm

Table 3 For each network, we show the weighted Kendall's τ coefficient of the rankings computed by the three global proxies and the ranking computed by TempBrandes. For Onbra we show the results using, respectively, a sample size such that Onbra's execution time is $1/10, 1/2$, and exactly the one of TempBrandes. For each instance, we highlight the best result in bold font.

Network	weighted Kendall's τ coefficient				
	Brandes	Prefix	Onbra$_{\frac{1}{10}}$	Onbra$_{\frac{1}{2}}$	Onbra$_1$
`Hypertext 2009`	0.90	0.67	0.86	0.94	**0.96**
`High school 2011`	0.89	0.56	0.82	0.92	**0.95**
`Hospital ward`	0.84	0.71	0.82	0.92	**0.94**
`College msg`	**0.95**	0.92	0.89	0.94	**0.95**
`Wiki elections`	**0.92**	**0.92**	0.84	0.90	**0.92**
`High school 2012`	0.90	0.56	0.81	0.89	**0.93**
`Digg reply`	0.94	**0.99**	0.73	0.83	0.86
`Infectious`	**0.92**	0.78	0.45	0.67	0.70
`Primary school`	0.89	0.13	0.88	0.94	**0.96**
`Facebook wall`	0.91	**0.98**	0.80	0.87	0.90
`Slashdot reply`	0.91	**0.96**	0.85	0.91	0.92
`High school 2013`	0.92	0.63	0.86	0.93	**0.95**
`Topology`	0.93	0.92	0.89	0.93	**0.94**
`SMS`	0.93	**0.99**	0.73	0.81	0.84
`Email EU`	0.95	0.88	0.91	0.96	**0.97**

in order for it to outperform Brandes. At this point, we would like to emphasize that our choice of sample size for Onbra is inherently impractical as it requires to run the exact algorithm first. We chose this approach in order to be as fair as possible when evaluating its performance in terms of quality. Choosing its sample size based on the time of other proxies, as, e.g., Brandes, makes its performance much worse in comparison. The results based on the intersection measure are somewhat similar, with Onbra performing slightly better.

4 Pass-Through Degree

Motivated by the fact that the running times of the global proxies employed in the previous section all grow much faster than linearly, we now turn to local proxies, i.e., proxies which compute centrality values purely based on nodes' neighborhoods. In the case of static graphs, it is common practice to compare more involved centrality notions to the simple degree centrality. Motivated by this fact, we here introduce a new degree notion for temporal graphs, which we will evaluate as a local proxy for shortest temporal betweenness in what follows. This new degree notion is somewhat related to the ego-betweenness, but it is in fact even simpler. In the end of this section, we will show that it can be computed for all nodes in nearly linear time in the number of temporal arcs.

Static Pass-Through Degree. With the aim of a simpler exposition, we start by giving the definition of our new degree notion for static directed graphs. We first note that the two standard degree notions in directed graphs, the in-degree $\mathrm{d}^{\mathrm{in}}(u) = |N^{\mathrm{in}}(u)|$ and the out-degree $\mathrm{d}^{\mathrm{out}}(u) = |N^{\mathrm{out}}(u)|$, both fail to observe the vertex as a whole, by taking in-going and out-going arcs into account in isolation. In undirected graphs, on the other hand, the

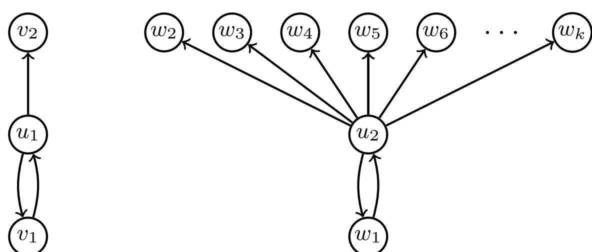


Figure 2 For the first variant, the pass-through degree of vertices u_1 and u_2 in the example graphs depicted above is equal. Namely, $d_1(u_1) = |N^{\text{in}}(u_1)| = 1 = |N^{\text{in}}(u_2)| = d_1(u_2)$. For the second variant this is not the case, as $d_2(u_1) = \sqrt{2}$ and $d_2(u_2) = \sqrt{k} = \Theta(\sqrt{n})$, where n denotes the number of nodes in the graph.

degree of a vertex also measures the number of neighbor pairs that can reach each other by passing through u, albeit normalized by the size of the neighborhood of u. In other words, $\mathrm{d}(u) = \frac{|N(u)|\cdot|N(u)|}{|N(u)|}$. This is, of course, just an overly complicated way of writing down the identity $\mathrm{d}(u) = |N(u)|$, but we use it as motivation for defining the analogous degree notion in directed graphs. We actually give two candidate definitions first, both generalizing the above equality to directed graphs, and then argue which of the two notions is more reasonable. The two variants of a directed degree notion that we propose, for a node $u \in V$, are $d_1(u) := \frac{|N^{\text{in}}(u)|\cdot|N^{\text{out}}(u)|}{|N^{\text{in}}(u)\cup N^{\text{out}}(u)|}$ and $d_2(u) := \sqrt{|N^{\text{in}}(u)| \cdot |N^{\text{out}}(u)|}$. When modeling an undirected graph $G = (V, E)$ as a directed graph $D = (V, A)$, by introducing two arcs (u, v) and (v, u) for every edge $\{u, v\} \in E$, we obtain, for every node u in the undirected graph, $N(u) = N^{\text{in}}(u) = N^{\text{out}}(v)$ and $d_1(u) = d_2(u) = d(u)$. Thus, both notions are proper generalizations of the undirected degree.

While at first sight it is not obvious which vertex degree definition is more suitable, both of them being legitimate generalizations of the undirected degree, one of the two turns out to be better suited for measuring vertex importance. As the examples in Figure 2 illustrate, the first candidate, d_1, has a serious drawback. More formally, when $N^{\text{in}}(u) \cup N^{\text{out}}(u) \in \{N^{\text{in}}(u), N^{\text{out}}(u)\}$, then $d_1(u) \in \{|N^{\text{in}}(u)|, |N^{\text{out}}(u)|\}$. This in particular means that in such a case, contrary to our initial intention, the degree of a node depends only on the in-going or the out-going arcs. Since the second candidate does not suffer from this issue, we find it more suitable for defining our directed degree notion. We now formally define it as the *pass-through degree* of a node.

▶ **Definition 1.** *In a static directed graph $G = (V, A)$, the* pass-through-degree *of $u \in V$ is defined as*

$$\mathrm{ptd}(u) := \sqrt{|N^{\text{in}}(u)| \cdot |N^{\text{out}}(u)|}$$

We point out that the pass-through degree is the geometric mean of in- and out-degree, the two classical notions of directed degree.

Temporal Pass-Through Degree. The introduced pass-through degree notion nicely generalizes to temporal graphs. Recall that the pass-through degree of a node u is equal to the geometric mean of the number of ordered neighbor pairs v, w that are connected through u. We generalize this to temporal nodes via pairs of neighbors that are temporally connected via exactly two hops through u. Formally, we write $v \xrightarrow{u} w$ if and only if there exist $a_1 = (v, u) \in A$ and $a_2 = (u, w) \in A$ such that $\lambda(a_1) < \lambda(a_2)$. We are now ready to define the temporal pass-through degree.

▶ **Definition 2.** *In a temporal graph* $\mathcal{G} = (V, A, \lambda)$*, the* temporal pass-through-degree *of* $u \in V$ *is*

$$\text{t-ptd}(u) := \sqrt{|\{(v,w) \in (V \setminus \{u\})^2 : v \xrightarrow{u} w\}|}$$

Algorithm 1 Temporal Pass-Through Degree.

Data: temporal arc list $\mathcal{A}$
Result: temporal pass-through degree of all vertices t-ptd

```
1  $\overline{G}, \underline{G} = \{\emptyset\}$ // initialize two empty temporal graphs
2  for each $(u,v,t) \in \mathcal{A}$ do
       // check if the edge already exists in $\overline{G}, \underline{G}$
3      if $(u,v) \in E(\overline{G})$ then
           // update max and min encountered label
4          $\overline{G}(u,v) = \max(\overline{G}(u,v), t)$, $\underline{G}(u,v) = \min(\underline{G}(u,v), t)$
5      else
6          add $(u,v)$ to $E(\overline{G}), E(\underline{G})$
7          $\overline{G}(u,v) = t$, $\underline{G}(u,v) = t$
8  sort edges of $\underline{G}$ in ascending order according to time labels
9  $L_{eat} = [[\cdot],[\cdot],\ldots,[\cdot]]$ // list of $n$ empty lists
10 for each $(u,v,\underline{t}) \in \underline{G}$ do
11     $L_{eat}[v].append(\underline{t})$
12 t-ptd $= [0,0,\ldots,0]$// initialize array of $n$ zeros
13 for each $(v,w,\overline{t}) \in \overline{G}$ do
       // compute the pass-through degrees
14     t-ptd$[v]$ = t-ptd$[v] + |\{t \in L_{eat}[v] : t < \overline{t}\}|$
15 return t-ptd
```

Computation of the Temporal Pass-Through Degree. Algorithm 1, given the temporal arc list $\mathcal{A}$ of a temporal graph, computes the pass-through degrees in $O(M \log m) = \tilde{O}(M)$ time and $O(m+n)$ space, where M is the number of temporal arcs and m, n are, respectively, the number of arcs and the number of nodes of the underlying static graph. More precisely, the first **for** loop (lines 2-7) iterates over all the temporal arcs and builds two simple labeled directed graphs, $\overline{G}$ and $\underline{G}$, which respectively keep track of the maximum and the minimum appearance time of each arc from the underlying graph. Building $\overline{G}$ and $\underline{G}$ requires $O(M \log m)$ time, as we can maintain a vertex-sorted list of already added arcs, and $O(m+n)$ space. Subsequently, the algorithm sorts the m arcs of the temporal graph $\underline{G}$ according to their time labels in time $O(m \log m)$. The second **for** loop (lines 10-11) iterates over all the (now sorted) arcs of the temporal graph $\underline{G}$, and appends the appearance time $\underline{t}$ of the arc $(u, v, \underline{t})$ to the minimum incoming times list of node v. Since $\underline{G}$ has exactly m arcs, this loop requires $O(m)$ steps and uses $O(m+n)$ space. Finally, using $O(n)$ space, the last **for** loop (lines 13-14) iterates over all the m temporal arcs $(u, w, \overline{t})$ of $\overline{G}$ and increments the t-ptd variables. More specifically, when encountering the temporal arc $(v, w, \overline{t})$, it increases the previous t-ptd value of v by the number of distinct in-going temporal arcs of v in $\underline{G}$ with $t < \overline{t}$ (line 14). Since $L_{eat}[v]$ is a sorted lists of length at most n (as $\underline{G}$ is a simple graph), for each $v \in V$ we can compute the new $ptd[u]$ in $O(\log n)$ via binary-search. Therefore, the last loop requires $O(m \log n)$ steps. The overall time and space complexities are therefore $O(M \log m) = \tilde{O}(M)$ and $O(m+n)$, respectively.

5 Local Proxies for Shortest Temporal Betweenness

We now turn to an experimental analysis of local proxies for shortest temporal betweenness. Our approach here is the same as in Section 3 and, besides the different choice of proxies, our experimental setting is identical. We first list the set of local proxies for shortest temporal betweenness that our study includes.

EgoSTB: The algorithm for computing the ego-shortest temporal betweenness ego-stb of all nodes by going through them iteratively, computing the ego-network of the respective node, and then calling the algorithm of Buß et al. [12] for computing the shortest temporal betweenness of the node in its ego-network.

EgoPrefix: The algorithm that, analogously to the one above, computes the ego-prefix foremost temporal betweenness ego-pftb of all nodes.

PTD: The algorithm for computing the temporal pass-through degree of all nodes in nearly linear time in the number of temporal arcs, described in Section 4.

We in addition examine the rankings produced by both the static and temporal versions of the in- and out-degree. These results are omitted from the main body of the paper (but can be found in Table 7, Appendix B). The quality of the rankings returned by PTD is usually much better, and only in a single case (on `Infectious`) is the obtained Weighted Kendall's τ value more than 0.01 worse than for another degree notion.

Table 4 For each network, we show the weighted Kendall's τ coefficient of the rankings computed by the three local proxies and the ranking computed by TempBrandes. For Onbra we show the results using a sample size such that Onbra's execution time is 1/10 the one of TempBrandes. For each instance, we highlight the best result in bold font.

Network	weighted Kendall's τ coefficient			
	Onbra$_{\frac{1}{10}}$	EgoPrefix	EgoSTB	PTD
`Hypertext 2009`	0.86	0.73	–	**0.89**
`High school 2011`	**0.82**	0.69	–	0.76
`Hospital ward`	**0.82**	0.77	–	**0.82**
`College msg`	0.89	0.94	0.94	**0.95**
`Wiki elections`	0.84	**0.94**	**0.94**	**0.94**
`High school 2012`	**0.81**	**0.81**	–	**0.81**
`Digg reply`	0.73	**0.96**	**0.96**	**0.96**
`Infectious`	0.45	0.76	**0.81**	0.65
`Primary school`	**0.88**	0.63	–	0.83
`Facebook wall`	0.8	**0.94**	**0.94**	0.93
`Slashdot reply`	0.85	**0.97**	**0.97**	0.96
`High school 2013`	**0.86**	0.83	–	0.83
`Topology`	0.89	**0.92**	–	**0.92**
`SMS`	0.73	0.95	**0.96**	0.94
`Email EU`	**0.91**	**0.91**	–	**0.91**

Experiment 2: Local Proxies' Correlation to TempBrandes

Running Times. The local proxies' running times can be found in the last three columns of Table 2. We note that the running time of EgoSTB easily becomes prohibitively large: in fact, we interrupted its execution once the time of TempBrandes elapsed, which resulted in eight (8) missing values for EgoSTB. We note that this is due to the large size of the ego networks, which can be deduced from the $n_e^{\max}$ column in Table 1. We emphasize that the nearly linear time algorithm from the previous section computes the pass-through degree of all nodes in less than 0.005% of the running time of TempBrandes on all data sets.

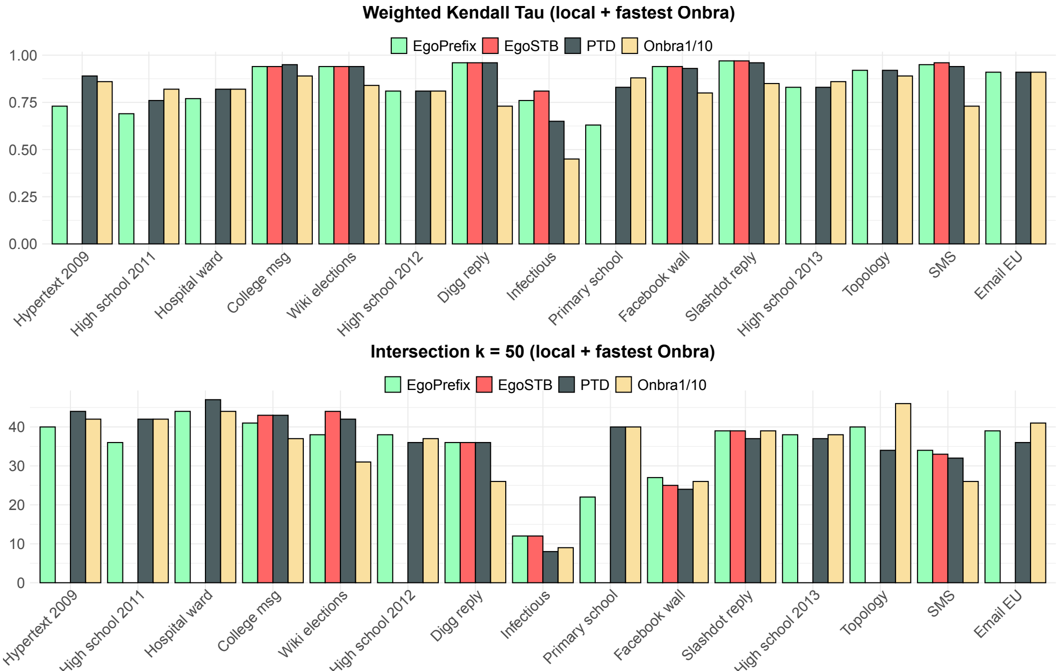


Figure 3 Comparison of the centrality ranking produced by TempBrandes and the rankings produced by the local proxies and Onbra with the smallest considered sample size. The comparison is given in terms of the weighted Kendall's τ coefficient and the intersection of the top 50 nodes.

Ranking Correlation. An illustration of the ranking correlation results of this experiment can be found in Figure 3. On top of the figure, we show the Weighted Kendall's τ correlation coefficient of the rankings computed by the respective proxies and the ranking computed by TempBrandes (see also Table 4). On the bottom, we show the results in terms of the intersection of the top-k nodes (again $k = 50$, see Table 6, Appendix A, for $k = 1$ and $k = 25$). In order to allow for better comparison with the results for global proxies from Section 3, in all the tables and plots that follow, we also include the fastest variant of Onbra, i.e, the variant with roughly 10% of TempBrandes' running time. We observe that the pass-through degree usually does not perform worse than the ego-variants of the shortest temporal betweenness and is at the same time much faster. In terms of both weighted Kendall's τ coefficient and the intersection measure, the pass-through degree performs better or at least as good as the considered version of Onbra on 10 out of 15 instances. At the same time its running time is between 3 and 4 orders of magnitudes smaller on all instances.

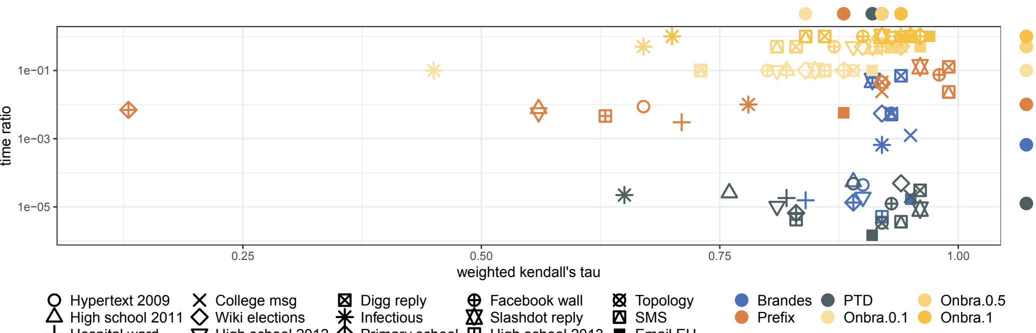


Figure 4 A two-dimensional illustration of ranking quality in terms of weighted Kendall's τ coefficient (on the horizontal linear axis) and the ratio between the proxies execution time and the time of TempBrandes (on the vertical logarithmic axis). The shapes of the points indicate the network, while the color indicates the proxy. On the top and on the right we plot the median value of the weighted Kendall's τ and the time ratio, respectively. We note that the running time ratios of the three Onbra variants are fixed to 1/10, 1/2, and 1, respectively.

6 Conclusion

We experimentally compared three global and three local proxies for shortest temporal betweenness rankings. One of these local proxies is a novel temporal degree notion, called the pass-through degree, which computes the number of neighbor pairs that are temporally connected by a two-hop path passing through the node at hand. Our experimental results are summarized in Figure 4, which depicts the performance of both global and local proxies discussed in previous sections (both in terms of running time and ranking quality). When applied to very large temporal networks, the pass-through degree clearly outperforms all the other competitors in terms of time performance. As indicated by the median time ratios that are depicted on the right of the plot, the pass-through degree achieves a time ratio that is around two (2), three (3), and four (4) orders of magnitude better than Brandes, Prefix, and the fastest considered Onbra variant, respectively. In terms of ranking quality, the medians of the two time-intense Onbra variants are best, followed by Brandes, PTD, Prefix, and the fastest Onbra variant.[4]

One future direction is explaining the correlations between PTD and the shortest temporal betweenness by using temporal graph parameters, such as the ones defined in the works of Tang et al. [45] and Nicosia et al. [33]. It would also be interesting to use PTD as a proxy for both static and temporal centralities in the context of routing schemes [31], as its local character enables an efficient distributed computation. From a theoretical point of view, possible directions of research include finding a conditional lower bound on the time complexity of computing shortest temporal betweenness that is better than the lower bound implied by its non-temporal counterpart. Proving a conditional lower bound on the computation of the ego-network betweenness measures (or designing a better algorithm) is also a very challenging question. Finally, the pass-through degree easily generalises to k-hop paths (instead of 2-hop paths). We believe that designing a quasi-linear time algorithm for computing such a generalisation, and verifying its quality in terms of proxying the shortest temporal betweenness, is the most natural continuation of this work.

[4] We note that we chose to compute medians rather than averages here, as the data seems to include several outliers (see, e.g., Prefix on the primary school network).

References

1 Sociopatterns. https://www.sociopatterns.org/, last checked on February 10, 2023.

2 Amir Abboud, Fabrizio Grandoni, and Virginia Vassilevska Williams. Subcubic equivalences between graph centrality problems, APSP and diameter. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1681–1697. SIAM, 2015. doi:10.1137/1.9781611973730.112.

3 Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 522–539. SIAM, 2021. doi:10.1137/1.9781611976465.32.

4 Alex Bavelas. Communication patterns in task-oriented groups. *The journal of the acoustical society of America*, 22(6):725–730, 1950.

5 Ferenc Béres, Róbert Pálovics, Anna Oláh, and András A Benczúr. Temporal walk based centrality metric for graph streams. *Applied network science*, 2018.

6 Paolo Boldi and Sebastiano Vigna. Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262, 2014.

7 Michele Borassi, Pierluigi Crescenzi, and Michel Habib. Into the square: On the complexity of some quadratic-time solvable problems. *Electron. Notes Theor. Comput. Sci.*, 322:51–67, 2016. doi:10.1016/j.entcs.2016.03.005.

8 Michele Borassi and Emanuele Natale. KADABRA is an adaptive algorithm for betweenness via random approximation. *ACM J. Exp. Algorithmics*, 24(1):1.2:1–1.2:35, 2019. doi:10.1145/3284359.

9 Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 2001.

10 Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

11 Laura F. Bringmann, Timon Elmer, Sacha Epskamp, Robert W. Krause, David Schoch, Marieke Wichers, Johanna Wigman, and Evelien Snippe. What do centrality measures measure in psychological networks? *Journal of Abnormal Psychology*, 128(8):892, 2019.

12 Sebastian Buß, Hendrik Molter, Rolf Niedermeier, and Maciej Rymar. Algorithmic aspects of temporal betweenness. In Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash, editors, *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 2084–2092. ACM, 2020. doi:10.1145/3394486.3403259.

13 The SciPy community. Statistical functions. https://docs.scipy.org/doc/scipy/reference/stats.html, last checked on February 10, 2023.

14 Pierluigi Crescenzi, Clémence Magnien, and Andrea Marino. Approximating the temporal neighbourhood function of large temporal graphs. *Algorithms*, 12(10), 2019.

15 Pierluigi Crescenzi, Clémence Magnien, and Andrea Marino. Finding top-k nodes for temporal closeness in large temporal graphs. *Algorithms*, 13(9), 2020.

16 Martin G. Everett and Stephen P. Borgatti. Ego network betweenness. *Soc. Networks*, 27(1):31–38, 2005. doi:10.1016/j.socnet.2004.11.007.

17 Robert Faris and Diane Felmlee. Status struggles: Network centrality and gender segregation in same-and cross-gender aggression. *American Sociological Review*, 76(1):48–73, 2011.

18 Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, March 1977. doi:10.2307/3033543.

19 Marwan Ghanem, Florent Coriat, and Lionel Tabourier. Ego-betweenness centrality in link streams. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017, Sydney, Australia, July 31 - August 03, 2017*. ACM, 2017.

20 R. Goerke. Email network of KIT informatics. https://i11www.iti.kit.edu/en/projects/spp1307/emaildata, 2011. Online; accessed 10 February 2023.

21 Peter Grindrod, Mark C Parsons, Desmond J Higham, and Ernesto Estrada. Communicability across evolving networks. *Physical Review E*, 2011.

22 Shahrzad Haddadan, Cristina Menghini, Matteo Riondato, and Eli Upfal. Repbublik: Reducing polarized bubble radius with link insertions. In *WSDM '21, The Fourteenth ACM International Conference on Web Search and Data Mining, Virtual Event, Israel, March 8-12, 2021*. ACM, 2021.

23 Petter Holme. Modern temporal network theory: a colloquium. *The European Physical Journal B*, 2015.

24 Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

25 Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

26 R. Kujala, C. Weckström, R. Darst, M. Madlenocić, and J. Saramäki. A collection of public transport network data sets for 25 cities. *Sci. Data*, 5:article number: 180089, 2018.

27 J. Kunegis. The KONECT Project. http://konect.cc, last checked on February 10, 2023.

28 Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. Stream graphs and link streams for the modeling of interactions over time. *Soc. Netw. Anal. Min.*, 2018.

29 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, last checked on February 10, 2023.

30 Carlos Lozares, Pedro López-Roldán, Mireia Bolibar, and Dafne Muntanyola. The structure of global centrality measures. *International Journal of Social Research Methodology*, 18(2):209–226, 2015.

31 Leonardo Maccari, Lorenzo Ghiro, Alessio Guerrieri, Alberto Montresor, and Renato Lo Cigno. On the distributed computation of load centrality and its application to DV routing. In *2018 IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018*, pages 2582–2590. IEEE, 2018. doi:10.1109/INFOCOM.2018.8486345.

32 Massimo Marchiori and Vito Latora. Harmony in the small-world. *Physica A: Statistical Mechanics and its Applications*, 285(3-4):539–546, 2000.

33 Vincenzo Nicosia, John Tang, Cecilia Mascolo, Mirco Musolesi, Giovanni Russo, and Vito Latora. Graph metrics for temporal networks. In *Temporal networks*. Springer, 2013.

34 Lutz Oettershagen and Petra Mutzel. Efficient top-k temporal closeness calculation in temporal networks. In *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2020.

35 Lutz Oettershagen, Petra Mutzel, and Nils M. Kriege. Temporal walk centrality: Ranking nodes in evolving networks. In *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*. ACM, 2022.

36 Ryan A. Rossi and Nesreen K. Ahmed. Network repository. https://networkrepository.com, last checked on February 10, 2023.

37 Polina Rozenshtein and Aristides Gionis. Temporal pagerank. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2016.

38 Maciej Rymar, Hendrik Molter, André Nichterlein, and Rolf Niedermeier. Towards classifying the polynomial-time solvability of temporal betweenness centrality. In *Graph-Theoretic Concepts in Computer Science - 47th International Workshop, WG 2021, Warsaw, Poland, June 23-25, 2021, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2021.

39 Diego Santoro and Ilie Sarpe. ONBRA: rigorous estimation of the temporal betweenness centrality in temporal networks. In *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*. ACM, 2022.

40 Nicola Santoro, Walter Quattrociocchi, Paola Flocchini, Arnaud Casteigts, and Frédéric Amblard. Time-varying graphs and social network analysis: Temporal indicators and metrics. *CoRR*, 2011.

41 John R Seeley. The net of reciprocal influence. a problem in treating sociometric data. *Canadian Journal of Experimental Psychology*, 3:234, 1949.

42 Frédéric Simard, Clémence Magnien, and Matthieu Latapy. Computing betweenness centrality in link streams. *CoRR*, 2021.
43 Charles Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:72–101, 1904.
44 John Kit Tang, Cecilia Mascolo, Mirco Musolesi, and Vito Latora. Exploiting temporal complex network metrics in mobile malware containment. In *12th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks, WOWMOM 2011, Lucca, Italy, 20-24 June, 2011*. IEEE Computer Society, 2011.
45 John Kit Tang, Mirco Musolesi, Cecilia Mascolo, and Vito Latora. Temporal distance metrics for social network analysis. In *Proceedings of the 2nd ACM Workshop on Online Social Networks, WOSN 2009, Barcelona, Spain, August 17, 2009*. ACM, 2009.
46 John Kit Tang, Mirco Musolesi, Cecilia Mascolo, Vito Latora, and Vincenzo Nicosia. Analysing information flows and key mediators through temporal centrality metrics. In *Proceedings of the 3rd Workshop on Social Network Systems, Paris, France, April 13, 2010*. ACM, 2010.
47 Shang-Hua Teng. Scalable algorithms for data and network analysis. *Found. Trends Theor. Comput. Sci.*, 12(1-2):1–274, 2016. doi:10.1561/0400000051.
48 Ioanna Tsalouchidou, Ricardo Baeza-Yates, Francesco Bonchi, Kewen Liao, and Timos Sellis. Temporal betweenness centrality in dynamic graphs. *Int. J. Data Sci. Anal.*, 2020.
49 Sebastiano Vigna. A weighted correlation index for rankings with ties. In Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi, editors, *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 1166–1176. ACM, 2015. doi:10.1145/2736277.2741088.
50 Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*, 2009.

A Top-k intersection value tables

Table 5 For each network, we show the intersections between the top 1, 25 and 50 nodes in the rankings computed by the three global proxies and the ranking computed by TempBrandes. For Onbra we show the results when its running time is, respectively, one tenth, half and exactly TempBrandes' execution time.

Network	Intersection														
	Brandes			Prefix			Onbra$_{\frac{1}{10}}$			Onbra$_{\frac{1}{2}}$			Onbra$_1$		
k	1	25	50	1	25	50	1	25	50	1	25	50	1	25	50
`Hypertext 2009`	1	22	43	1	15	37	1	20	42	1	22	46	1	22	47
`High school 2011`	0	20	45	0	14	31	0	19	42	0	20	47	1	22	48
`Hospital ward`	0	22	45	1	18	43	0	20	44	0	22	47	0	24	48
`College msg`	1	22	42	1	12	29	1	19	37	0	23	44	1	23	46
`Wiki elections`	1	16	34	0	7	21	1	15	31	1	21	41	1	22	42
`High school 2012`	1	21	44	0	8	27	1	18	37	1	20	42	1	22	45
`Digg reply`	1	20	40	1	22	44	0	14	26	0	20	39	1	20	43
`Infectious`	1	19	31	0	3	7	0	6	9	0	5	11	0	6	10
`Primary school`	0	18	43	0	3	18	1	20	40	1	23	46	1	23	45
`Facebook wall`	0	10	15	1	17	37	1	15	26	1	17	37	1	19	43
`Slashdot reply`	1	18	39	1	20	38	1	19	39	1	22	45	1	23	46
`High school 2013`	1	20	44	0	11	26	1	20	38	0	22	45	1	23	46
`Topology`	1	21	41	1	20	39	1	23	46	1	23	47	1	24	49
`SMS`	0	13	32	1	20	43	0	15	26	0	18	33	1	19	40
`Email EU`	1	20	41	0	14	34	1	19	41	1	23	46	1	23	47

Table 6 For each network, we show the intersections between the top 1, 25 and 50 nodes in the rankings computed by the three global proxies and the ranking computed by TempBrandes. For Onbra we show the results when its running time is one tenth of TempBrandes execution time.

Network	Intersection											
	ONBRA$_{\frac{1}{10}}$			EgoPrefix			EgoSTB			PTD		
k	1	25	50	1	25	50	1	25	50	1	25	50
Hypertext 2009	1	20	42	1	17	40	–	–	–	1	20	44
High school 2011	0	19	42	0	16	36	–	–	–	1	17	42
Hospital ward	0	20	44	1	18	44	–	–	–	0	21	47
College msg	1	19	37	0	20	41	0	22	43	1	23	43
Wiki elections	1	15	31	1	19	38	1	21	44	1	19	42
High school 2012	1	18	37	1	16	38	–	–	–	1	13	36
Digg reply	0	14	26	1	19	36	1	19	36	1	19	36
Infectious	0	6	9	0	4	12	0	4	12	0	2	8
Primary school	1	20	40	0	11	22	–	–	–	0	17	40
Facebook wall	1	15	26	1	14	27	1	14	25	1	13	24
Slashdot reply	1	19	39	1	20	39	1	20	39	1	19	37
High school 2013	1	20	38	0	18	38	–	–	–	0	17	37
Topology	1	23	46	1	21	40	–	–	–	0	16	34
SMS	0	15	26	0	13	34	0	13	33	0	12	32
Email EU	1	19	41	0	18	39	–	–	–	1	18	36

B Comparison among Degree Notions

Table 7 For each network, we show the weighted Kendall's τ coefficient of the rankings computed by the static/temporal degree notions and the pass-through degree and the ranking computed by TempBrandes.

Network	weighted Kendall's τ coefficient				
	PTD	in-degree	out-degree	t-in-degree	t-out-degree
Hypertext 2009	**0.89**	**0.89**	**0.89**	0.72	0.72
High school 2011	0.76	**0.77**	**0.77**	0.40	0.40
Hospital ward	0.82	0.83	0.83	**0.85**	**0.85**
College msg	**0.95**	0.91	0.92	0.90	0.91
Wiki el's	**0.94**	0.74	0.72	0.72	0.72
High school 2012	0.81	**0.82**	**0.82**	0.50	0.50
Digg reply	**0.96**	0.84	0.83	0.84	0.83
Infectious	0.65	**0.70**	**0.70**	0.42	0.42
Faceb'k w'l	**0.93**	0.86	0.89	0.85	0.88
Primary school	0.83	**0.84**	**0.84**	0.70	0.70
Slashdot reply	**0.96**	0.78	0.94	0.80	0.94
SMS	**0.94**	0.84	0.88	0.69	0.81
High school 2013	0.83	**0.84**	**0.84**	0.50	0.50
Topology	**0.92**	**0.92**	**0.92**	**0.92**	**0.92**
Email EU	**0.91**	0.87	0.90	0.77	0.83

C Public transport networks

A special class of temporal networks are public transport networks, in which the existence of a temporal arc (u, v, t) indicates that it is possible to reach location v from u by taking a mean of public transport at time t. Indeed, these networks are characterized by a sort of "regularity" (that is, nodes are very similar in terms of in- and out-degree), which makes the local proxies quite bad in proxying the temporal shortest betweenness. At the same time, they contain a huge amount of shortest temporal paths, which forced us to use big integer data structures for TempBrandes and Onbra, thus significantly slowing down their execution time. As an example, we considered the two networks `Venice` and `Bordeaux` that stem from the work of Kujala et al. [26], and are chosen to be analysed because of their different sizes and geographies. The main characteristics of these two networks are summarised in the following table.

Data set	**n**	**m**	**M**	**T**	$\mathbf{t_{STB}}$	$\mathbf{n_e^{max}}$	**Type**	**Source**
`Venice`	1874	3465	113670	1691	7758	20	D	[26]
`Bordeaux`	3435	4040	236075	60582	50937	13	D	[26]

The execution times TempBrandes and of all proxies (except for Onbra) in seconds are shown in the following table (once again dashes indicate that the experiment was interrupted after the time of TempBrandes elapsed and we omit Onbra from the table as its running time is fixed to approximately 1/10, 1/2, or 1 times the running time of TempBrandes due to the choice of the sample size).

Network	Execution Time (seconds)					
	TempBrandes	Brandes	Prefix	EgoPrefix	EgoSTB	PTD
`Venice`	7758	0.7374	72.9	31	48	0.0168
`Bordeaux`	50937	2.8722	443.0	93	107	0.0161

The weighted Kendall's τ coefficient of the rankings computed by the three global proxies and the ranking computed by TempBrandes are shown in the following table (once again, for Onbra we show the results using, respectively, a sample size such that Onbra's execution time is $1/10, 1/2$, and exactly the one of TempBrandes, and, for each instance, we highlight the best result in bold font).

Network	weighted Kendall's τ coefficient				
	Brandes	Prefix	Onbra$_{\frac{1}{10}}$	Onbra$_{\frac{1}{2}}$	Onbra$_1$
`Venice`	0.90	0.80	0.93	0.96	**0.98**
`Bordeaux`	0.96	0.82	0.94	0.97	**0.98**

We can observe that, in this case, Onbra even in the case of the smallest sample size is better than Brandes and Prefix. However, it is also worth observing that Brandes performs quite well in the case of both networks, suggesting that, in these cases, the temporality of the network does not influence so much the ranking of the nodes. This might be intuitively justified by the fact that a "temporally" central node in this kind of networks is also central in the underlying graphs. This is confirmed by the following table, which shows the intersection values for all global proxies for values of $k = 1, 25, 50$ (once again, a value of x in the table means that the top-k nodes with respect to the ranking computed by a given proxy contain x of the top-k nodes of the ranking computed by TempBrandes).

Network		Intersection														
		Brandes			Prefix			$\text{Onbra}_{\frac{1}{10}}$			$\text{Onbra}_{\frac{1}{2}}$			Onbra_{1}		
	k	1	25	50	1	25	50	1	25	50	1	25	50	1	25	50
`Venice`		0	16	36	0	12	23	1	23	41	0	24	47	0	24	48
`Bordeaux`		1	23	46	1	17	29	1	19	48	1	23	48	1	24	49

The next table shows the weighted Kendall's τ coefficient of the rankings computed by the three local proxies and the ranking computed by TempBrandes (once again, for Onbra we show the results using a sample size such that Onbra's execution time is 1/10 the one of TempBrandes, and, for each instance, we highlight the best result in bold font).

Network	weighted Kendall's τ coefficient			
	$\text{Onbra}_{\frac{1}{10}}$	EgoPrefix	EgoSTB	PTD
`Venice`	**0.93**	0.64	0.62	0.61
`Bordeaux`	**0.94**	0.63	0.55	0.61

As expected, the local proxies perform quite bad and Onbra is by far better than all of them. This is confirmed by the following table, which shows the intersections between the top 1, 25 and 50 nodes in the rankings computed by the three global proxies and the ranking computed by TempBrandes (once again, for Onbra we show the results when its running time is one tenth of TempBrandes execution time).

Network		Intersection											
		$\text{ONBRA}_{\frac{1}{10}}$			EgoPrefix			EgoSTB			PTD		
	k	1	25	50	1	25	50	1	25	50	1	25	50
`Venice`		1	23	41	0	5	13	0	4	13	0	5	11
`Bordeaux`		1	19	48	0	4	5	0	4	5	0	4	5

Simple Runs-Bounded FM-Index Designs Are Fast

Diego Díaz-Domínguez
Department of Computer Science, University of Helsinki, Finland

Saska Dönges
Department of Computer Science, University of Helsinki, Finland

Simon J. Puglisi
Helsinki Institute for Information Technology (HIIT), Finland
Department of Computer Science, University of Helsinki, Finland

Leena Salmela
Department of Computer Science, University of Helsinki, Finland

Abstract

Given a string X of length n on alphabet σ, the FM-index data structure allows counting all occurrences of a pattern P of length m in $O(m)$ time via an algorithm called *backward search*. An important difficulty when searching with an FM-index is to support queries on L, the Burrows-Wheeler transform of X, while L is in compressed form. This problem has been the subject of intense research for 25 years now. Run-length encoding of L is an effective way to reduce index size, in particular when the data being indexed is highly-repetitive, which is the case in many types of modern data, including those arising from versioned document collections and in pangenomics. This paper takes a back-to-basics look at supporting backward search in FM-indexes, exploring and engineering two simple designs. The first divides the BWT string into blocks containing b symbols each and then run-length compresses each block separately, possibly introducing new runs (compared to applying run-length encoding once, to the whole string). Each block stores counts of each symbol that occurs before the block. This method supports the operation $\mathsf{rank}_c(L, i)$ (i.e., count the number of times c occurs in the prefix $L[1..i]$) by first determining the block i/b in which i falls and scanning the block to the appropriate position counting occurrences of c along the way. This partial answer to $\mathsf{rank}_c(L, i)$ is then added to the stored count of c symbols before the block to determine the final answer. Our second design has a similar structure, but instead divides the run-length-encoded version of L into blocks containing an equal number of runs. The trick then is to determine the block in which a query falls, which is achieved via a predecessor query over the block starting positions. We show via extensive experiments on a wide range of repetitive text collections that these FM-indexes are not only easy to implement, but also fast and space efficient in practice.

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

Keywords and phrases data structures, efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.7

Supplementary Material *Software (Source Code)*: `https://github.com/saskeli/block_RLBWT` archived at `swh:1:dir:244fd876cd15bd291d91e444c8c04bb289aed05f`

Funding This work was supported in part by the Academy of Finland via grants 339070, 351150, and 323233.

Acknowledgements The authors wish to thank the Finnish Computing Competence Infrastructure (FCCI) for supporting this project with computational and data storage resources.

1 Introduction

Given a string $X[0, n-1]$, the suffix array [21] of X, denoted SA_X (or just SA when clear from context) is a permutation of the integers $[0, n-1]$ that tells the lexicographical order of the suffixes of X, i.e., SA is the permutation such that $X[SA[0]..n] < X[SA[1]..n] < \ldots <$

© Diego Díaz-Domínguez, Saska Dönges, Simon J. Puglisi, and Leena Salmela;
licensed under Creative Commons License CC-BY 4.0

$X[SA[n-1]..n]$. Because of the lexicographical ordering, all the positions of occurrence of any pattern P that is a substring of X lie in a contiguous interval of SA, and thus pattern matching over X reduces to determining the appropriate interval of SA.

An FM-index is a data structure that supports finding such SA intervals. The main common feature of the FM-index and its variants is support for the function $\mathsf{extendLeft}([i,j],c)$, which, given a suffix array interval $SA[i,j]$ containing all the occurrences of some pattern P, and a symbol c, returns the interval $[i',j']$ such that $SA[i',j']$ contains all occurrences of pattern $P' = cP$. Clearly, having support for extendLeft allows one to perform pattern matching: given a pattern P of length m, we can find all the occurrences of P in X via a sequence of m applications of extendLeft that return intervals corresponding to increasingly longer suffixes of the pattern.

FM-index implementations (see [19, 23] and references therein) differ primarily on how they support extendLeft. For many years, this was done via rank queries on the Burrows-Wheeler transform (BWT) of the input string. The BWT of string X, which we denote with L_X, or just L when clear from context, is a permutation of the symbols of X defined by SA_X. In particular, $L[i] = X[SA[i]-1]$ except when $SA[i] = 0$, in which case $L[i] = X[n]$.

In their groundbreaking article [10], Ferragina and Manzini showed that, somewhat remarkably, $\mathsf{extendLeft}([i,j],c) = [C[c] + \mathsf{rank}_c(L,i), C[c] + \mathsf{rank}_c(L,i) - 1]$, where $C[c]$ is the total number of symbols in X less than symbol c, and the query $\mathsf{rank}_c(L,i)$ returns the number of occurrences of symbol c in $L[0..i]$. Most known FM-index variants follow this scheme. The fastest rank-based FM-index we know of, due to Gog et al. [13], divides L into blocks and represents each block with a Huffman-shaped wavelet tree, the bitvectors of which are compressed with different schemes depending on characteristics of each block's entropy.

Another more recent (and less populated) class of FM-index variants avoids rank queries altogether, instead essentially storing space-efficient mappings from (interval, symbol) pairs to intervals in order to implement extendLeft. The first of these methods is due to Belazzougui and Navarro [3], who describe an index that uses minimal monotone perfect hash functions and uses $nH_k + O(n)$ bits of space, where H_k is the kth-order empirical entropy of the input text [22]. More recently, Nishimoto and Tabei [25], describe a structure using $O(r)$ words space, where r is the number of runs[1] in L.

On highly repetitive strings, which are now produced by many sources, and are notably central to the field of computational genomics [28, 8, 9, 23, 20], the BWT string L is composed of relatively few runs – i.e., r is significantly smaller than n, perhaps by 2-3 orders of magnitude depending on the input – and so L can be compressed well with run-length encoding. The problem then becomes supporting extendLeft (via rank queries or otherwise) in close to $O(r)$ space.

Contribution. This paper is a back-to-basics examination of run-length compressed FM indexes that explores two simple ways to achieve space usage close to $O(r)$ words while maintaining fast query times in practice.

Both designs are rank-based and divide L into blocks. The first approach divides L into n/b blocks of b symbols each (with the last block possibly having less), before applying run-length encoding to each block. Rank can be implemented on this structure in $O(b)$ time and the index takes $O(r + \sigma n/b)$ words of space. The second approach divides L into blocks containing an equal number b' of *runs*. Rank now takes $O(b' + \mathsf{pred}(r/b'))$ time, where

[1] A run $L[i..j] = c^\ell$ has $\ell > 1$ consecutive copies of c such that $i = 1$, or $L[i-1] \neq c$, and $L[j] = n$ or $L[j+1] \neq c$.

$\mathsf{pred}(m)$ is the time for a predecessor query over m integers, and space is $O(r + \sigma r/b')$ words. We find that both these simple schemes afford very fast practical implementations, and dominate other approaches in practice on many data sets. Along the way we explore practical structures to support predecessor queries, which may be of independent interest.

Our rationale in this study is twofold. Firstly, simple code is easier to maintain and easier to adapt to different application requirements, which is important given the relevance of the FM-index in modern genomics analyses [18, 8]. Secondly, simpler data structures are often easier to optimize, both for the programmer and for the compiler. For the particular problem we consider in this paper, our results suggest that simpler designs may have an inherent speed advantage over more complex ones. In their simplicity and strong performance, the indexes we describe herein represent non-trivial baselines against which the performance of future, possibly more sophisticated, FM indexes can be measured.

Roadmap. The remainder of this paper is organized as follows. In the next section we review related work. Our new FM-indexes are then described in Section 3. In that section we also present a microbenchmark of different predecessor structures, which are essential for our second index. Then, in Section 4, we present results of benchmarks comparing our indexes to the state of the art. Conclusions and avenues for future work are then offered.

2 Background and Related Work

We now briefly describe four FM index designs [19, 13, 2, 25] that represent the state of the art, either in theory or practice. Implementations of all methods described here are included in our experiments. For a more thorough treatment, including earlier and more obscure indexes, we refer the reader to [24, 23].

Mäkinen and Navarro [19] proposed the FM-index variant to encode L in $O(r \log n)$ bits. It supports $\mathsf{rank}_c(L, i)$ (and therefore $\mathsf{extendLeft}$) in $O(\log \log_w(\sigma + n/r))$ time. Our description here follows that given in Gagie et al. [11].

Let $R = (c_1, \ell_1), \ldots, (c_r, \ell_r)$ be the run-length encoding of L, with pair (c_i, ℓ_i), $i \in [1..r]$, representing the *ith* run in L, where $c_i \in \Sigma$ is the run symbol and ℓ_i is the run length. We maintain a vector $L' = c_1, \ldots, c_r$ with the run heads in the same order as they appear in L. L' is represented with a data structure of $O(r \log n)$ bits that supports $O(1)$-time access and $\mathsf{rank}_c(L', i)$ in $O(\log \log_w \sigma)$ time. We also maintain an array $C'[1, \sigma]$ storing in $C'[c]$ the number of runs in L whose symbol is smaller than $c \in \Sigma$.

A predecessor data structure of $O(r \log n)$ bits encodes the set $E = \{1\} \cup \{1 < i \leq n,\ L[i-1] \neq L[i]\}$ with the positions in L for the run heads (i.e., the leftmost symbol in every run). The query $\mathsf{pred}(E, i)$ returns pair (i', b), where i' is the predecessor of i in E, and b is the rank of i' in E. $\mathsf{pred}(E, i)$ takes $O(\log \log_w(n/r))$ time using [4].

Let R' be a permutation of R in which the runs are stably sorted by their symbols and let (c_i, ℓ_i) be the *ith* run in the permutation R'. We store an array $D[1..r]$ of $r \log n$ bits storing in $D[i]$ the cumulative length of the runs associated with c_i in $R'[1..i]$.

Answering $\mathsf{rank}_c(L, i)$ in the RLFM requires us to call $(i', b) = \mathsf{pred}(E, i)$ to get the head position i' and rank b for the run where i lies. We can then obtain the symbol $c' = L'[b]$ associated with i's run. Subsequently, we obtain the number $k = \mathsf{rank}_c(L', b-1)$ of runs for c in the prefix $L'[1..b-1]$ and finally compute the number $x = D[C'[c] + k]$ of cs in the prefix $L[1..j'-1]$, returning $x + i - i' + 1$ if $c = c'$, or return x otherwise.

Overall query time is dominated by $\mathsf{pred}(E, i)$ and $\mathsf{rank}_c(L', b-1)$, which combined take $O(\log \log_w(\sigma + (n/r)))$ time.

Gog et al. [13] describe what is currently the fastest general-purpose FM index we know of. Their approach shares some similarity to one of our schemes in that it divides L into blocks of fixed size b. Each block stores, for each symbol of the alphabet, precomputed ranks up to the beginning of the block. Each block is then encoded using a Huffman-shaped wavelet tree [19], which can answer rank queries up to the beginning of each block, and combined with the precomputed ranks, enables general rank queries to be answered in $O(\log \sigma)$ time. Each wavelet tree take space proportional to the entropy of its block, leading to a bound of $nH_k + o(n \log \sigma)$ bits for the whole index.

Although the index size is not directly relatable to r, the number of runs in L, experiments in Gog et al. [13] show the index performs well on inputs with small r. Intuitively, on a L having many long runs, the blocks tend to have a small alphabet with a skewed distribution of symbols and so will have a small representation when Huffman encoded (which is essentially what the Huffman-shaped wavelet tree does). However, a further optimization has been made, that favors runs even further. In particular, the bit vectors of the wavelet trees are represented with the hybrid encoding of Kärkkäinen et al. [16] which run-length encodes bitvectors having long runs. If the block being encoded has long runs then the bitvector at the root of its wavelet tree will have at least as many, and bitvectors at other nodes will tend to preserve runs too. This makes the space usage much closer to r than it was designed to be, even if it does not explicitly encode runs in L.

Prezza et al. [2] store one character per run in a string $H \in \Sigma^r$ and mark with a 1 the beginning of each run in a bitvector $V_{all}[0..n-1]$. For every $c \in \Sigma$ they store the lengths of all runs of character c consecutively in a bitvector V_c. More precisely, every run of symbol c of length k is represented in V_c as 10^{k-1}. This representation allows them to map rank and access queries on L to rank, select and access queries on H, V_{all}, and V_c. By gap-encoding the bitvectors, this representation takes $r(2\log(n/r) + \log \sigma)(1 + o(1))$ bits of space. The multiplicative term $\log(n/r)$ can be reduced by storing in V_{all} just one out of $1/\epsilon$ ones, where $0 < \epsilon \leq 1$ is a constant. One is still able to answer all queries on L, by using the V_c vectors to reconstruct the positions of the missing ones in V_{all}, though this multiplies query time by $1/\epsilon$. In their implementation of this scheme, Prezza et al. [2] represent H as a Huffman-shaped wavelet tree and store the bitvectors in an Elias-Fano structure [29].

Nishimoto and Tabei [25] proposed the first encoding that represents L in $O(r)$ bits and supports $\mathsf{extendLeft}(i, L[i])$[2] in constant time without resorting to rank operations. Let $I = \{(s_1, e_1), \ldots, (s_r, e_r)\}$ be a sequence of r consecutive ranges over $[1, n]$ such that $L[s_j, e_j]$, with $j \in [1, r]$, is the jth run of L (from left to right). Each range $(s_j, e_j) \in I$ has an associated mapping pair $(s'_j = \mathsf{extendLeft}(s_j, L[s_j]), e'_j = \mathsf{extendLeft}(e_j, L[e_j])$ that represents the contiguous range $L[s'_j, e'_j]$ one obtains by performing $\mathsf{extendLeft}$ operations over the symbols in $L[s_j..e_j]$. Note that (s'_j, e'_j) does not necessarily match a range in I, but can be fully contained in one or cover several of them. The key idea in Nishimoto and Tabei's method to obtain constant time is to further break the ranges to produce a new sequence I' of length r', $r \leq r' < 2r$ where every $(s_j, e_j) \in I'$ has a mapping pair (s'_j, e'_j) that covers a constant number c of other ranges in I'. They maintain an array $D[1..r']$ that stores in $D[j]$ the pair (s_j, s'_j) and a vector $D' = [1..r']$ that stores in $D'[j] = y \in [1, r']$ the index y of the pair $(s_y, e_y) \in I'$ where s'_j lies (i.e., $s_y \leq s'_j \leq e_y$). Answering $\mathsf{extendLeft}(i, L[i])$ in this scheme requires first knowing the pair $(s_j, e_j) \in I'$ enclosing i, then scanning the area $D[D'[j]..D'[j]{+}c{-}1]$ until the range $(s_y, e_y) \in I'$ that contains $\mathsf{extendLeft}(i, L[i]) = s'_j + (i - s_j)$ is found. This approach was recently implemented by Brown et al. [5].

[2] This constrained version of $\mathsf{extendLeft}$ is also known as $\mathsf{LF}(i)$ in the literature.

3 Simple Runs-Bounded FM-index Designs

Both of our indexing schemes are based on splitting the BWT into run-length encoded blocks along with preceding symbol counts. To answer $\mathsf{rank}_c(L, i)$ queries, it suffices to find the block containing the *ith* character, scanning the block and adding the result of the scan to the number of preceding occurrences of c.

The number of preceding symbols are stored in at most $7 + \sum_{c\in\Sigma} \lceil \log_2 \mathsf{rank}_c(L, n)\rceil$ bits per block[3]. In our implementations with a constant number of symbols per block, there is an additional "super block" level that stores precomputed answers for rank queries for blocks of size 2^{32}. The super block approach was taken to allow storing large chunks of memory for the blocks instead of just one very large block of memory, and to guarantee that four bytes would be sufficient to store any symbol counts on the block level. Neither of these turned out to be relevant in our experiments, and the additional layer of complexity likely slows down query performance very slightly and has a negligible impact on data structure size.

Blocks with b symbols. If each block contains b symbols. We can store pointers to the $\lceil n/b\rceil$ blocks in an array BP. With this, $\mathsf{rank}_c(L, i)$ reduces to $\mathsf{rank}_c(BP[\lfloor i/b\rfloor], i \bmod b)$ on the block.

The upper bound space complexity, given logarithmic encoding of integers comprises of:

- $\mathcal{O}\left(\left(\frac{n}{b} + r\right)(\log b + \log \sigma)\right)$ bits for run encoding,
- $\mathcal{O}\left(\frac{n}{b}\sigma \log n\right)$ bits for encoding the block headers containing preceding character counts and
- $\mathcal{O}\left(\frac{n}{b}w\right)$ bits for storing block pointers, w being the machine word length,

making

$$\mathcal{O}\left(\left(\frac{n}{b} + r\right)(\log b + \log \sigma) + \frac{n}{b}(w + \sigma \log n)\right)$$

bits an asymptotic upper bound for these SB (for *symbol block*) structures.

Query time is simply $\mathcal{O}(b)$ under the word RAM model. We expect at most two memory transfers from uncached memory locations per query, meaning most of the work would be done on cached data.

The blocks themselves consist of a run-length-encoded sequence either using two bytes per run with runs split as necessary, or with each run being encoded with a variable number of bytes depending on the length of the run. In both approaches the first $\lceil \log_2 \sigma\rceil$ bits encode the run head.

- In two byte encoded blocks, $16 - \lceil \log_2 \sigma\rceil$ bits are used to store the run length. Thus the maximum length for one run is $2^{16-\lceil \log_2 \sigma\rceil}$, meaning long runs will need to be split into multiple runs within the same block. In the worst case, for a large alphabet ($> 2^7$) and large block sizes ($\geq 2^{13}$) a single run can be split into more than 2^4 blocks, thus increasing the space usage considerably. However, for genomics data and small ($< 2^{13}$) block sizes, the vast majority of runs fit into two bytes. Scanning these two byte encoded blocks is fast since there is no branching or data dependencies in decoding runs and only a single branch miss-prediction to end scanning. We note that for this variant, the bounds for space complexity apply as long as b is considered a bound constant as opposed to being linear in n for example.

[3] The +7 bits is the possible overhead of keeping data byte-aligned.

- With runs encoded into a variable number of bytes, arbitrarily long runs can be efficiently encoded in $\mathcal{O}(\log\sigma + \log m)$ bits where m is the length of the run. With long runs this is more space efficient than the two byte approach, but decoding, especially of short runs is slower as decoding requires data-dependent branching.

In addition to these encodings we experimented with hybrid compression schemes where the encoding is chosen on a per-block basis. However, we found that the overhead of decoding the more rare block type is high enough that our dynamic encodings remain uncompetetive with more simple block encodings. We suspect that in addition to a branch missprediction, the cold code path causes inefficiencies in instruction caching or pipeline unrolling, that dominate the efficiency gains of a superior encoding for some minority of blocks. An improvement in performance of querying the rarer blocks with a separate encoding compared to the majority blocks of ≈ 100 nanoseconds ($\approx 20\%$ improvement) would be sufficient to warrant reconsidering a hybrid approach. This idea may be feasible with some as yet untested block encodings.

Blocks with b' runs. For finding the block containing the ith symbol when blocks contain a variable number of symbols, we use a predecessor search to find m and $\mathsf{arg\ max}_j(j \leq i))$ s.t. block m starts with the jth symbol. Now given m and j, $\mathsf{rank}_c(L, i)$ becomes $\mathsf{rank}_c(BP[m], i - j)$.

For running the predecessor search we tested four different approaches:

- A simple **binary search** over an array of tuples with the number of symbols in L preceding each block and a pointer to the block itself. This approach is very simple to implement and has minimal memory overhead. However, the memory access pattern is not well supported on modern microprocessors, unless the entire array can fit in cache.
- A **heap ordered binary search tree** is a balanced search tree stored in an array A, such that for every internal node $A[i]$, the left child will be found at $A[2i + 1]$ and the right child at $A[2i + 2]$. Internal nodes contain the number of symbols represented by their left sub tree, while leaves contain pointers to the actual blocks. The computation done is exactly the same as for binary search, but the heap ordering makes the memory pattern an increasing stride in the same direction, allowing for more efficient predictive caching by modern microprocessors. For tree traversal to work properly, all but the final internal level in the tree need to be full (containing 2^ℓ elements where ℓ is the level) even if some sub trees are empty, this potentially doubles the memory footprint of the heap ordered binary search tree compared to a simple binary search.
- A **B$^+$-tree** [6] is a B-tree with pointers to blocks only stored in the leaves, while internal nodes only store the number of symbols represented by the sub trees. When $B = 2^k$ for some $k \in \mathbb{N}$, a fast templated branchless [27] binary search can be implemented for branch selection in the B-tree. While the internal nodes close to the root of the tree are likely to be present in cache, accessing nodes closer to the leaves are likely to trigger cache misses. In addition, the B-tree is comparatively space inefficient due to the need to store pointers internally.
- A **heap ordered B$^+$-tree** aims to be the best of both worlds, by storing the B$^+$-tree in contiguous memory without the need for child pointers. The children of node $A[i]$ will be at $A[Bi + 1..Bi + B]$. Internal branch selection can be done with templated binary searches and the memory access pattern follows a somewhat predictable increasing forward stride. While the need to keep internal levels of the tree full implies significant memory overhead, the low number of levels necessary to do block selection even for large data sets, keeps the memory overhead low in practice.

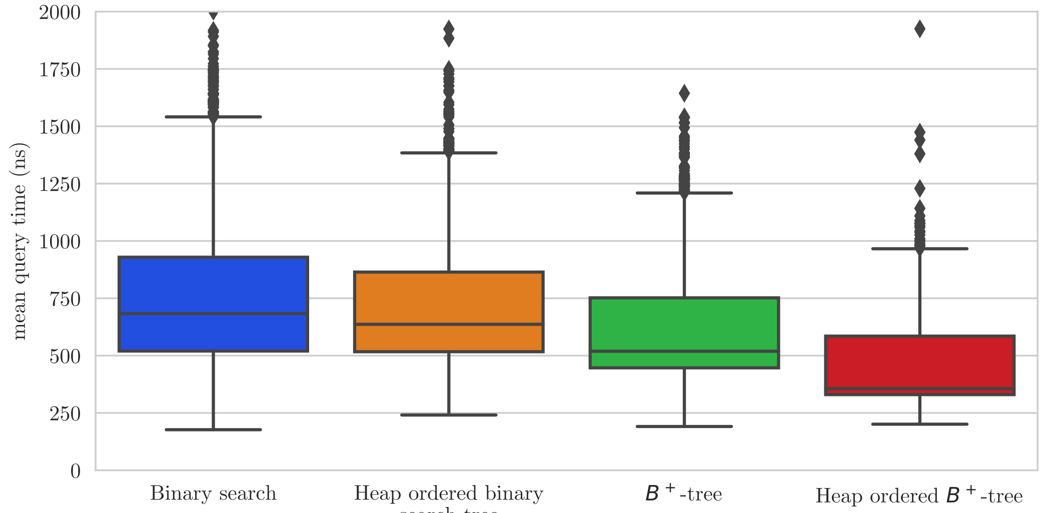


Figure 1 Predecessor search performance. Based on 10^5 pred(x) queries over 10^7 elements, with x chosen at random from $[0..m + 10]$ where m is the largest element in the collection. Both B$^+$-tree and the heap ordered binary search tree are faster than a simple binary search, while the heap ordered B$^+$-tree is significantly faster still. The performance here and in Table 1 differ due to the overhead of outputting results of every query as opposed to just calculating summary statistics.

After benchmarking all of these potential approaches, we found that the heap ordered B$^+$-tree was the fastest and had reasonable memory overhead (Figure 1 and Table 1). The space complexity of our RB (for *run block*) implementation comprises of:

- $\mathcal{O}(r(\log\sigma + \log n))$ bits for encoding runs,
- $\mathcal{O}\left(\frac{r}{b'}\sigma\log n\right)$ bits for encoding the block headers containing preceding character counts,
- $\mathcal{O}\left(\frac{r}{b'}w\right)$ bits for storing block pointers, and
- $\mathcal{O}\left(\frac{r}{b'}w\right)$ bits for storing the heap ordered B$^+$-tree,

making

$$\mathcal{O}\left(r(\log\sigma + \log n) + \frac{r}{b'}(\sigma\log n + w)\right)$$

bits an asymptotic upper bound for these RB structures.

Time complexity is $\mathcal{O}\left(\log\frac{r}{b'} + b'\right)$. We expect fewer than $\log r/b'$ memory accesses to uncached memory locations due to the apparent efficiency of pre-caching with increasing stride.

See Figure 2 for an example, illustrating the memory layout and query logic of our index structures.

Table 1 Performance of benchmarked predecessor search structures. 10^5 random predecessor queries on a set of ten million random (unique) elements. Heap ordered binary search trees and B$^+$-trees are faster than a traditional binary search, but have significant space overhead. Heap ordered B$^+$-tree is faster still with only minor overhead in space efficiency compared to binary search.

	binary search	heap ordered BST	B$^+$-tree	heap ordered B$^+$-tree
Mean query time	524.348ns	331.548ns	363.409ns	313ns
Space usage	76.294MB	128MB	157.511MB	78.3257MB

SB with block size 10:

$\mathsf{rank}_C(L, 35)$

$\lfloor 35/10 \rfloor = 3$

0 1 2 3

$\mathsf{rank}_C(BP[3], 35 \bmod 10)$

A: 0 C: 0 G: 0 T: 0 A^{10} A: 10 C: 0 G: 0 T: 0 A^5C^3GT A: 15 C: 3 G: 1 T: 1 $T^5G^3A^2$ A: 17 C: 3 G: 4 T: 6 A^2TCG T

RB with 3 runs per block and heap-ordered B^+-tree with b = 2:

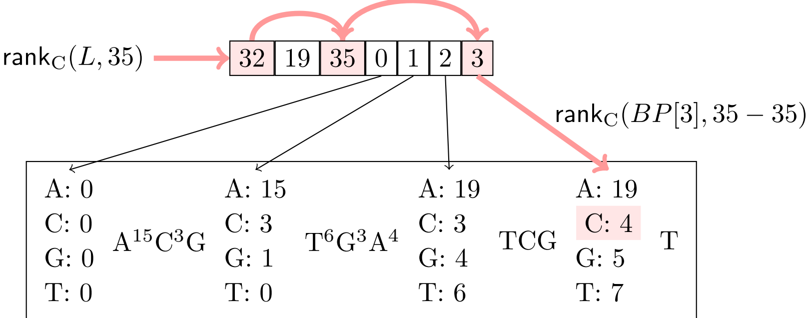


Figure 2 Illustrative example of our index structures. Indexes built on input sequence $L = A^{15}C^3GT^6G^3A^4TCGT$. Memory access pattern for $\mathsf{rank}_C(L, 35)$ highlighted. For SB, five symbols, in four runs, containing one "C" character get read from block $BP[3]$, this one "C" count gets added to the number of preceding "C" characters in the block header (3), these get added together giving $\mathsf{rank}_C(L, 35) = 4$. For RB the pred query tells us that $BP[3]$ is the target block, and that a total of 35 symbols precede the block, no scanning is needed and the result is the total count of "C" characters in the block header (4).

4 Performance

We implemented the designs described in Section 3 to verify their efficacy in practice, and to explore time-space trade-offs in performance by varying b' for our RB variant and varying b for SB variants. The source code is in C++, and it is available at `https://github.com/saskeli/block_RLBWT`.

4.1 Experimental Setup

Machine and Compilation. Tests were run on a machine with an AMD EPYC 7452 processor and 496GB of DDR4 RAM. The operating system was AlmaLinux 8.4 (with Linux 4.18.0-372.9.1.el8.x86_64 kernel). Code was compiled with GCC 12.2.0 and the performance-relevant flags: `-std=c++2a`, `-march=native`, `-Ofast` and `-DNDEBUG`. Experiments were repeated on a machine on the same HPC cluster with an Intel Xeon E7-8890 v4 processor (see Appendix B). We found results to be relatively stable across these two systems.

Indexes Measured. We use sbt_rlbwt (for *symbol blocks, two-byte*) to refer to the SB FM index variant from Section 3 that first divides the BWT into blocks contain equal numbers of symbols and then applies two byte run length encoding to each block; sbv_rlbwt (for *symbol blocks, variable width*) to refer to the SB index variant that divides the BWT into blocks containing equal numbers of symbols before using a variable width encoding to compress the blocks; and use rb_rlbwt for our RB index that divides the run-length encoded BWT into blocks of fixed runs.

We explored the space efficiency / query time trade-offs of varying block sizes for our indexes. We used block sizes of $2^8 \leq b \leq 2^{17}$ for SB variants, and $2^0 \leq b' \leq 2^9$ for RB, as these ranges cover what we consider practical values in most cases.

- sbt_rlbwt. Uses a default block size of $b = 2^{11}$, which yields reliably fast indexes and occasionally suffer in terms of compression performance.
- sbv_rlbwt. Uses a default block size of $b = 2^{14}$. This value enables good compression ratios most of the time while not completely sacrificing query performance.
- rb_rlbwt. With each block containing $b' = 32$ runs by default and using a heap ordered B^+-tree predecessor structure, with $B = 64$, for answering pred queries. We selected these parameters as good defaults for genomics data sets.

We further compared the performance of our variants using default parameters against other state-of-the-art FM-index implementations. These were:

- FBB. The fixed-block boosting with wavelet trees index of Gog et al. [13]. We obtained the code from the SDSL-lite library [12]. We realised that substituting the hybrid bit vector implementation used in the wavelet trees with other bit vector variants available in the SDSL-lite library led to interesting time / space trade-offs for the FBB implementations. We include four of these variants as fbb_xx in our results
  - fbb_hyb was built using the hybrid bit vector of Kärkkäinen et al. [17]. The SDSL-lite library implements the hybrid bit vector in the hyb_vector class.
  - fbb_bv was built using uncompressed bit vectors from the SDSL-lite library (class bit_vector) with separate rank support data structures.
  - fbb_il was built using the SDSL-lite implementation (bit_vector_il) of the bit vector from Gog et al. [14] that interleaves the partial rank queries with uncompressed bit vector blocks.
  - fbb_rrr uses the rrr_vector class of SDSL-lite, which implements the H_0-compressed bit vector representation of Raman et al. [26].
- RLBWT is the rlmn class of the SDSL-lite library that implements the run-length-encoded BWT representation with rank support of Mäkinen and Navarro [19].
- SRLBWT. The sparse RLBWT scheme of Belazzougui et al. [2], which is the component used for counting in the the r-index implementation [11]. We obtained its implementation from the r-index's official repository[4].
- R-INDEX-F represents the scheme of Nishimoto and Tabei [25] as implemented by Brown et al. [5][5].

We remark that in preliminary experiments (not shown here) we also measured the performance of indexes based on balanced and Huffman-shaped wavelet trees applied to the entire BWT (with various internal bitvector representations), but found, as other authors

[4] https://github.com/nicolaprezza/r-index
[5] https://github.com/drnatebrown/r-index-f

Table 2 Details of the datasets used in performance benchmarks. The symbol σ denotes the alphabet size, n is the number of symbols in the dataset, and r is the number of runs in the BCR BWT of that dataset.

Dataset	σ	n	r	n/r
`hum50`	16	1.54×10^{11}	3.89×10^{9}	39.53
`ecoli3.6K`	16	1.90×10^{10}	1.57×10^{8}	120.55
`cov400K`	6	1.19×10^{10}	9.05×10^{6}	1317.79
`einstein`	140	4.68×10^{8}	2.90×10^{5}	1611.18
`worldleaders`	90	4.70×10^{7}	5.73×10^{5}	81.90
`coreutils`	235	2.05×10^{8}	4.68×10^{6}	43.82

have (e.g., [15, 13]), that these approaches were always inferior to the indexes listed above on our data sets. We therefore exclude these from further mention in the experiments for the sake of clarity.

Datasets. We used six repetitive collections for the experiments. See Table 2. They vary in length, alphabet size and level of repetitiveness to reflect different application scenarios. The dataset `hum50` consists of 50 different human assemblies. The dataset `ecoli3.6k` is the concatenation of 3600 E. coli genomes. The dataset `covid400k` contains 400,000 variants of the covid genome. `einstein` and `worldleaders` are each concatenations of different versions of a Wikipedia entry (for "Albert Einstein" and "World Leaders", respectively). `coreutils` contains different versions of the GNU coreutils' source code[6].

Benchmarks. For every dataset, we proceeded as follows. We built its BCR BWT string L [1] using the `grlBWT` tool [7]. Then, with L as input, we built an instance of each index listed above. We sampled sets of patterns of lengths 10, 30 and 50. For each set we sampled substrings of the desired length from ten thousand random positions in the input data set, such that the substrings contained only printable characters for the general data sets and no "N" symbols for genomic data sets. We measured the elapsed times for each index to count all patterns in each pattern set and then took the average, further dividing by pattern length to get a per symbol time for each pattern set.

4.2 Results

Time-space trade-offs. Increasing the block sizes for SB variants had the expected effect of improving compression while sacrificing query performance, suggesting that when optimizing for query speed, block size should be reduced as much as possible within memory constraints. For RB increasing block size improves compression to the detriment of query performance, but reducing the number of runs per block beyond a certain point decreases query performance. This performance decrease is expected if we observe that as the number of runs per block approaches one, the index becomes a heap ordered B$^+$-tree with single runs at the leaves, and the $\Theta(\log r/b)$ tree traversal becomes $\Theta(\log r)$. These performance trade-offs are shown in Figure 3.

[6] More details of the last three data sets can be found at corpus: `http://pizzachili.dcc.uchile.cl/repcorpus.html`.

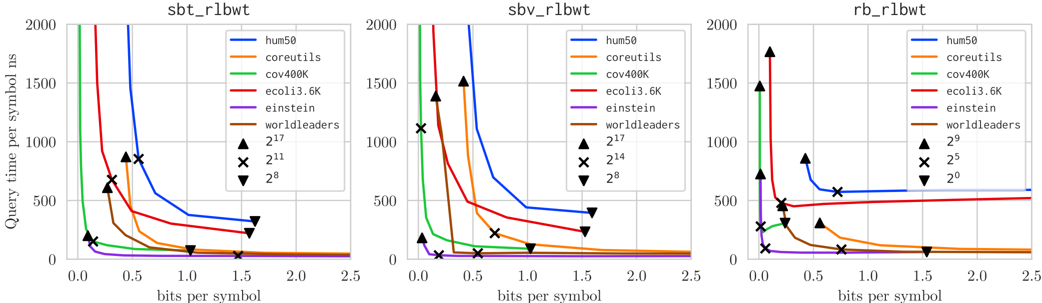


Figure 3 Performance trade-offs for varying block sizes, with maximum, default and minimum tested block sizes annotated as shapes. SB variants have a clear trade-off where increasing block size improves compression and slows down queries, while decreasing block size degrades compression performance and speeds up queries. The same is mostly true for RB as well but the increased overhead of the predecessor search when block sizes decrease limits how much speed up is possible. The sweet spot for minimizing both compressed size and query time differs between data sets. Default parameters for `sbt_rlbwt` and `rb_rlbwt` seem mostly reasonable but parameter tuning is recommended for best performance in specific applications.

Comparison against other FM-index implementations. Our index implementations are very competitive with other FM-index implementations, as can be observed in Figure 4. Our experiments show that as pattern length increases, the performance of our indexes improves in comparison to the competition (See Appendix A for figures with additional pattern lengths). We posit that this gain is due to the likelihood of both of the rank queries associated with a step of extendLeft targeting the same block increasing as the number matching suffixes becomes lower. This feature allows the second query to act on fully cached memory making the simple scanning approaches very fast in practice.

5 Concluding Remarks

We have described, engineered, and experimentally analysed two strikingly simple FM indexes, which often outperform more complex prior art. We close with three avenues for future work.

Firstly, our indexes were designed with genomics applications in mind and so, therefore, the approach we selected for encoding and decoding run heads is likely to be sub-optimal for data sets on larger alphabets. We believe that improving the run-head encoding with, e.g., Huffman codes, and allowing the character encoding, in some cases, to exceed eight bits, would improve compression with a negligible impact on query performance.

For larger alphabets, the space overhead incurred by the precomputed ranks stored at each block quickly becomes substantial. To mitigate this, it may be fruitful to treat rare symbols differently, possibly with an entirely different rank structure, avoiding the need to store a full σ precomputed ranks at each block.

Finally, as mentioned in Section 3, selecting encoding on a per-block basis was explored but discarded as the overhead for decoding was too high to be practical with out currently available block encodings. Block encodings designed specifically for fast decoding of blocks with specific attributes may prove fast enough in practice to be worth the overhead in encoding detection when querying.

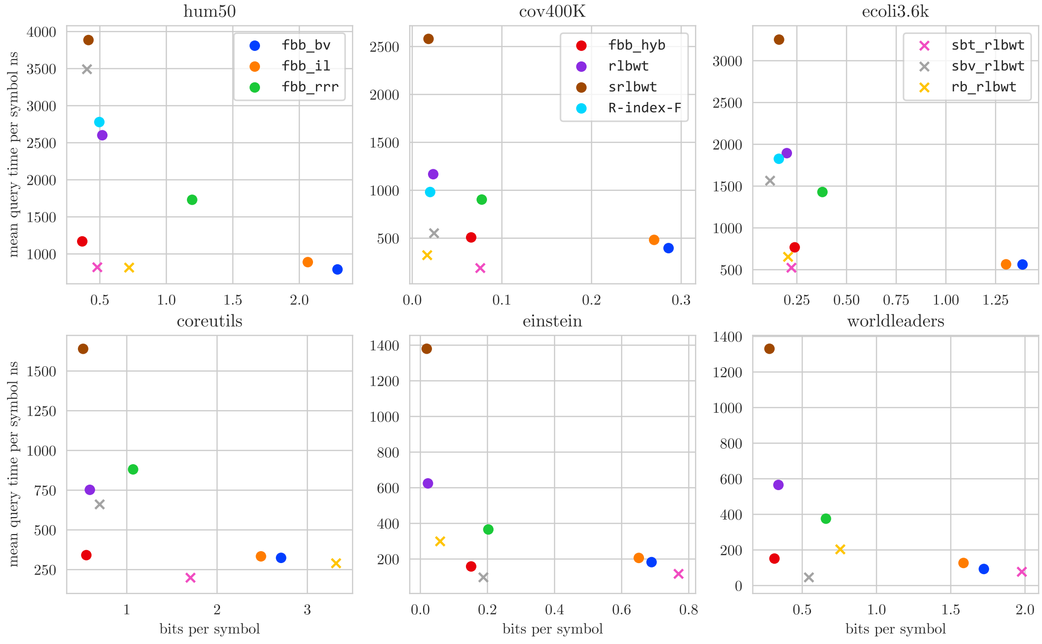


Figure 4 Mean match counting per symbol query times and bits per symbol for patterns of length 30. Our `sbt_rlbwt` and `rb_rlbwt` variants with default parameters are as fast or faster than the competition for genomics data, and compress significantly better than the closest competitors in query time. For the more general data sets we remain competitive but lose out in compression, possibly due to our implementation not doing any entropy-based compression, and thus encoding run heads inefficiently. However, the simple run head encoding does translate to good query performance.

References

1. Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483:134–148, 2013.
2. Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Flexible indexing of repetitive collections. In *Proc. 13th Conference on Computability in Europe (CiE)*, pages 162–174, 2017.
3. Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23:1–23:19, 2014.
4. Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4), 2015.
5. Nathaniel K. Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. In *Proc. 20th International Symposium on Experimental Algorithms (SEA)*, page article 16, 2022.
6. Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
7. Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the BWT for repetitive text using string compression. In *Proc. 33rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 223, page article 29, 2022.
8. Dirk D. Dolle, Zhicheng Liu, Matthew Cotten, Jared T. Simpson, Zamin Iqbal, Richard Durbin, Shane A. McCarthy, and Thomas M. Keane. Using reference-free compressed data structures to analyze sequencing reads from thousands of human genomes. *Genome Research*, 27(2):300–309, 2017.
9. Jana Ebler, Peter Ebert, Wayne E. Clarke, Tobias Rausch, Peter A. Audano, Torsten Houwaart, Yafei Mao, Jan O. Korbel, Evan E. Eichler, Michael C. Zody, et al. Pangenome-based genome inference allows efficient and accurate genotyping across a wide spectrum of variant classes. *Nature Genetics*, 54(4):518–525, 2022.

10 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

11 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):article 2, 2020.

12 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms, (SEA)*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.

13 Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Fixed block compression boosting in FM-indexes: Theory and practice. *Algorithmica*, 81(4):1370–1391, 2019.

14 Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.

15 Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Softw. Pract. Exp.*, 44(11):1287–1314, 2014.

16 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Proc. 14th Data Compression Conference (DCC)*, pages 302–311, 2014.

17 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Proc. 24th Data Compression Conference (DCC)*, pages 153–162, 2014.

18 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Methods*, 9(4):357–359, 2012.

19 V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

20 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

21 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

22 Giovanni Manzini. An analysis of the burrows-wheeler transform. *J. ACM*, 48(3):407–430, 2001.

23 G. Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.

24 G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

25 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Proc. 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, page article 101, 2021.

26 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

27 Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *Proc. 12th European Symposium on Algorithms (ESA)*, pages 784–796, 2004.

28 Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. Big data: astronomical or genomical? *PLoS Biology*, 13(7):e1002195, 2015.

29 Sebastiano Vigna. Quasi-succinct indices. In *Proc. Sixth ACM International Conference on Web Search and Data Mining (WSDM)*, pages 83–92. ACM, 2013.

A Additional match counting results on AMD EPYC 7452

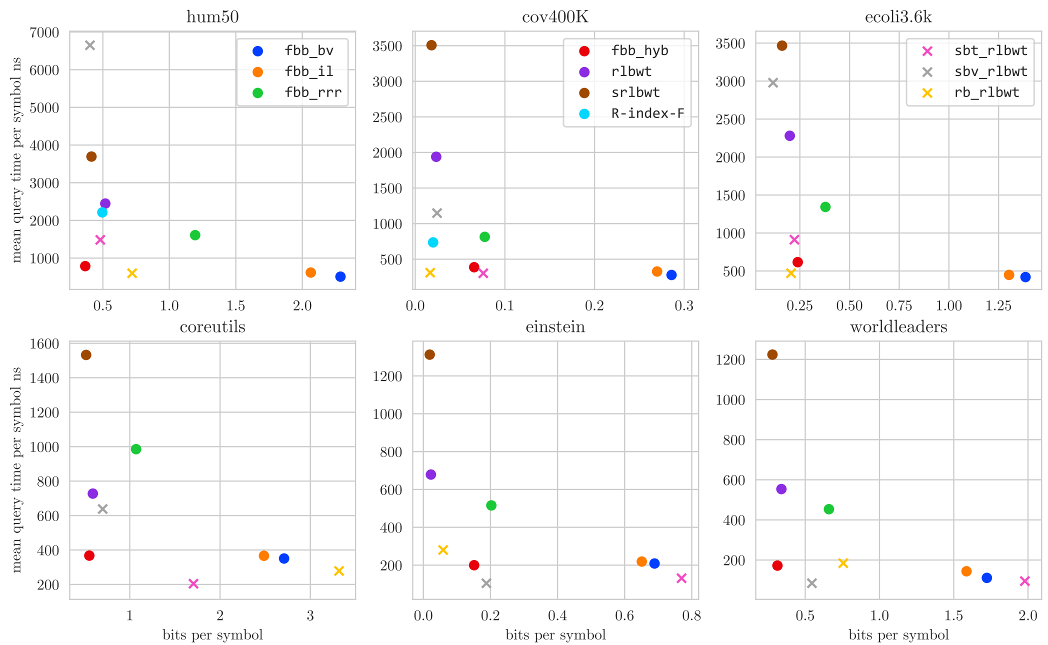


Figure 5 Mean match counting per symbol query times and bits per symbol for patterns of length 10.

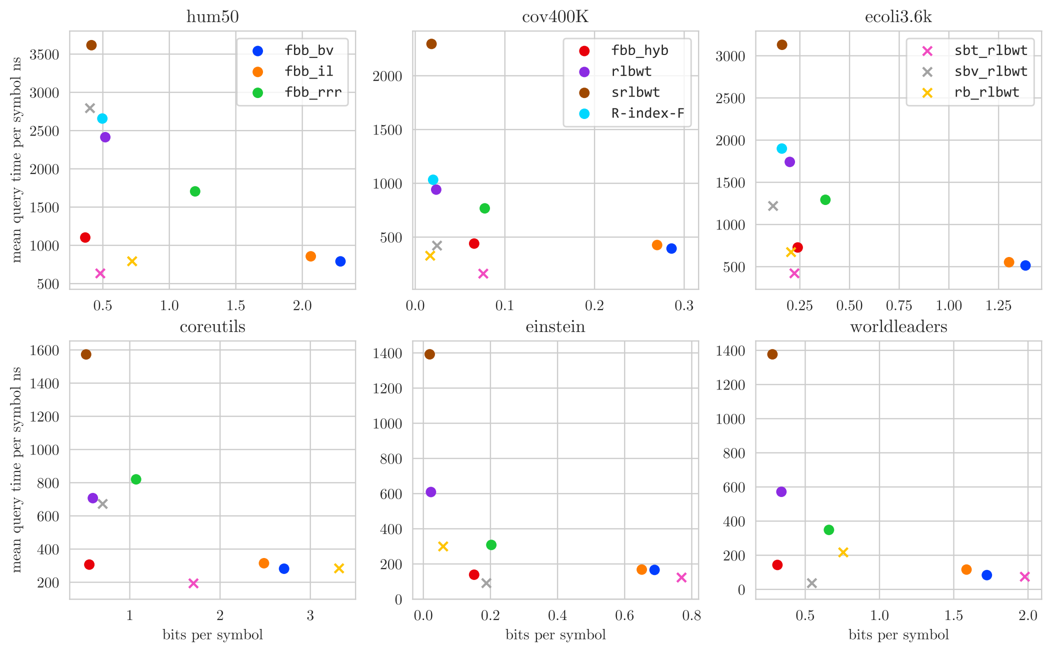


Figure 6 Mean match counting per symbol query times and bits per symbol for patterns of length 50.

B Intel Xeon E7-8890 v4 experiment results

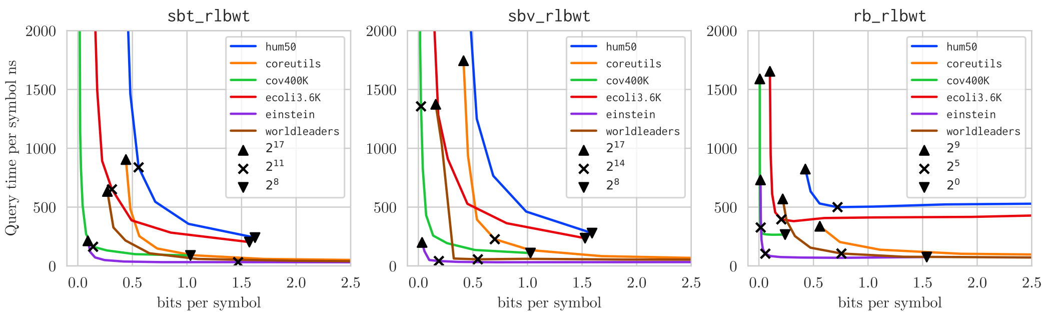


Figure 7 Performance trade-offs for varying block sizes. Run on Intel Xeon E7-8890 v4.

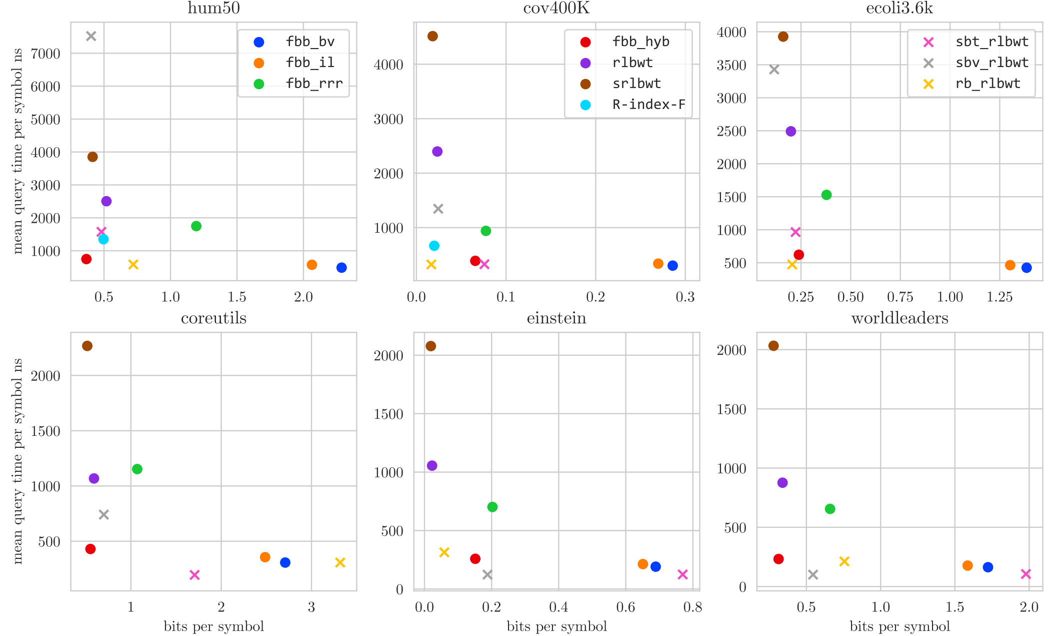


Figure 8 Mean match counting per symbol query times and bits per symbol for patterns of length 10. Run on Intel Xeon E7-8890 v4.

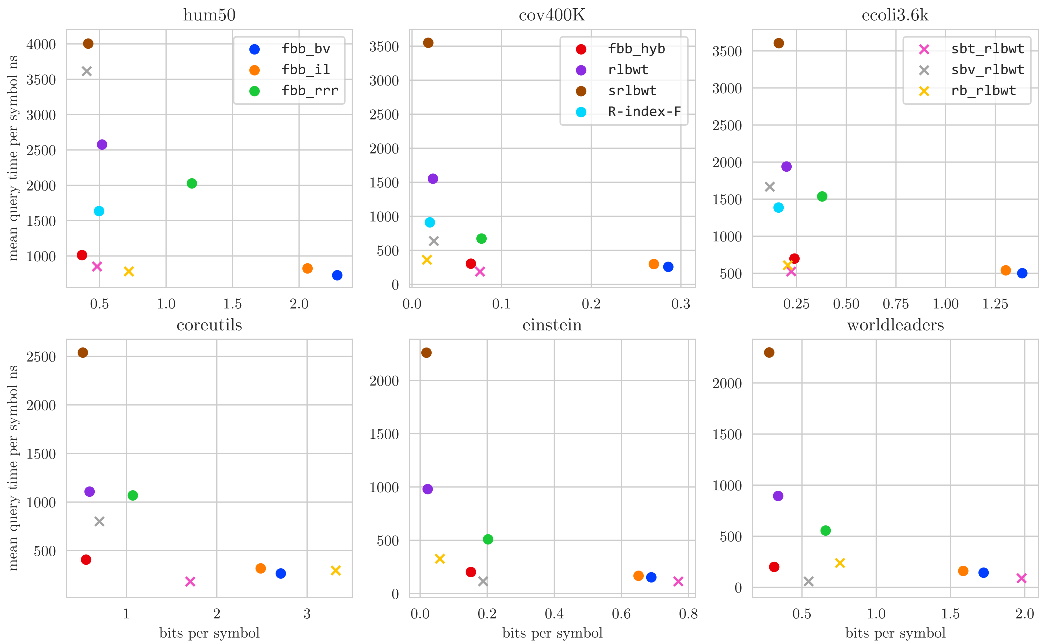


Figure 9 Mean match counting per symbol query times and bits per symbol for patterns of length 30. Run on Intel Xeon E7-8890 v4.

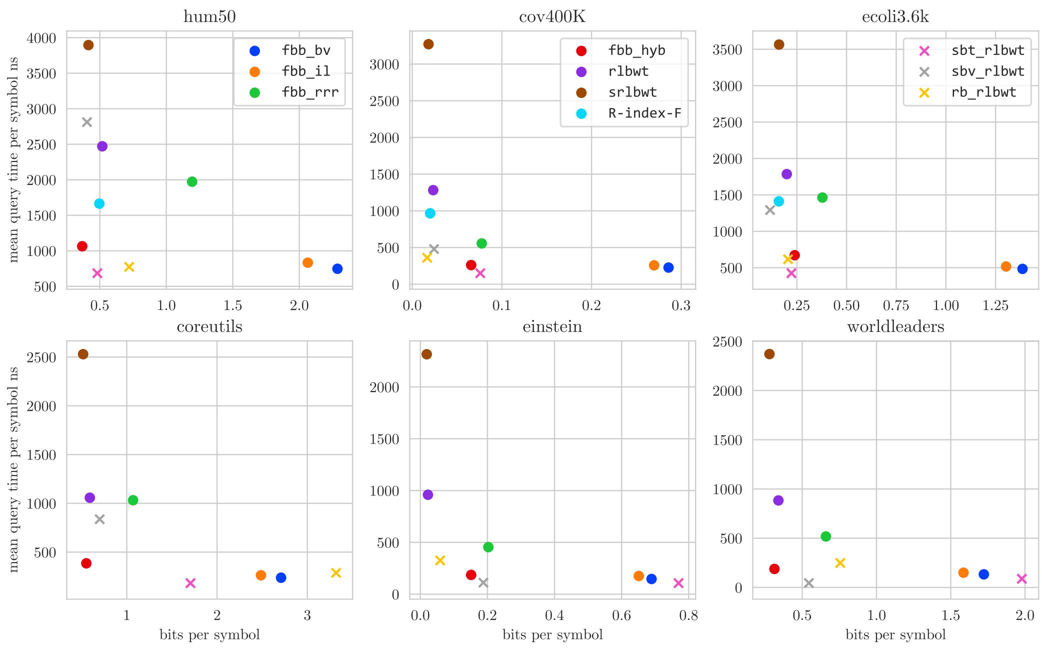


Figure 10 Mean match counting per symbol query times and bits per symbol for patterns of length 50. Run on Intel Xeon E7-8890 v4.

Noisy Sorting Without Searching: Data Oblivious Sorting with Comparison Errors

Ramtin Afshar ✉
University of California, Irvine, CA, USA

Michael Dillencourt ✉
University of California, Irvine, CA, USA

Michael T. Goodrich ✉
University of California, Irvine, CA, USA

Evrim Ozel ✉
University of California, Irvine, CA, USA

Abstract

We provide and study several algorithms for sorting an array of n comparable distinct elements subject to probabilistic comparison errors. In this model, the comparison of two elements returns the wrong answer according to a fixed probability, $p_e < 1/2$, and otherwise returns the correct answer. The ***dislocation*** of an element is the distance between its position in a given (current or output) array and its position in a sorted array. There are various algorithms that can be utilized for sorting or near-sorting elements subject to probabilistic comparison errors, but these algorithms are not data oblivious because they all make heavy use of noisy binary searching. In this paper, we provide new methods for sorting with comparison errors that are data oblivious while avoiding the use of noisy binary search methods. In addition, we experimentally compare our algorithms and other sorting algorithms.

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

Keywords and phrases sorting, algorithms, randomization, experimentation

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.8

Supplementary Material
Software (Source Code): `https://github.com/UC-Irvine-Theory/NoisyObliviousSorting`
archived at `swh:1:dir:d49510784ab64d4ac0f4f9d2879587b83e8d91a8`

1 Introduction

Given an array, A, of n distinct comparable elements, we study the problem of efficiently sorting A subject to noisy probabilistic comparisons. In this framework, which has been extensively studied [2, 5, 7–9, 14, 16, 17, 19, 21, 23, 31], the comparison of two elements, x and y, results in a true result independently according to a fixed probability, and otherwise returns the opposite (false) result. In the case of ***persistent*** errors [2, 7–9, 17], the result of a comparison of two given elements, x and y, always returns the same result. In the case of ***non-persistent*** errors [5, 14, 16, 19, 23, 31], however, the probabilistic determination of correctness is determined independently for each comparison, even if it is for a pair of elements, (x, y), that were previously compared.

Motivation for sorting with comparison errors comes from multiple sources, including applied cryptography scenarios where cryptographic comparison protocols can fail with known probabilities (see, e.g., [6, 20, 33]). In such cases, reducing the noise from comparison errors can be computationally expensive, and the framework advanced in our paper offers an alternative, possibly more efficient approach, where a higher error rate is tolerated while still achieving the ultimate goal of sorting or near-sorting with high probability. Further, other applications of sorting with comparison errors include ranking objects in online forums via group A/B testing [32].

© Ramtin Afshar, Michael Dillencourt, Michael T. Goodrich, and Evrim Ozel;
licensed under Creative Commons License CC-BY 4.0
21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis; Article No. 8; pp. 8:1–8:18
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Since it is not possible to always correctly sort an array, A, subject to persistent comparison errors, we follow the formulation of Geissmann *et al.* [7–9], and define the ***dislocation*** of an element, x, in an arary, A, as the absolute value of the difference between x's index in A and its index in the correctly sorted permutation of A. Further, define the ***maximum dislocation*** of A as the maximum dislocation for the elements in A, and let the ***total dislocation*** of A be the sum of the dislocations of the elements in A. By known lower bounds [7–9], the best a sorting algorithm can achieve under persistent comparison errors is a maximum dislocation of $O(\log n)$ and a total dislocation of $O(n)$. Thus, coming close to such asymptotic maximum and total dislocation guarantees should be the goal for a sorting algorithm in the presence of persistent comparison errors.

Given the cryptographic applications of noisy comparisons, we desire sorting algorithms that are ***data oblivious***, which support privacy-preserving cryptographic protocols. A sorting algorithm is data oblivious if its memory access pattern does not reveal any information about the data values being sorted. Unfortunately, existing efficient algorithms for sorting with noisy comparisons are not data oblivious. Indeed, they all make use of noisy binary search [8], which is a data-sensitive random walk in a binary search tree, e.g., see Geissmann, Leucci, Liu, and Penna [8], Feige, Raghavan, Peleg, and Upfal [5], and Leighton, Ma, and Plaxton [19]. Instead, we desire efficient sorting algorithms that tolerate noisy comparisons and avoid the use of noisy binary search, so as to be data oblivious (i.e., privacy preserving if comparisons are done according to a data-hiding protocol).

Related Prior Results. Problems involving probabilistic comparison errors can trace their roots back to a classic problem by Rényi [27] of playing a two-person game where player A poses yes/no questions to a player B who lies with a given probability; see a survey by Pelc [24]. Notable prior results include a paper by Pippenger [25] on computing Boolean functions with probabilistically noisy gates and work by Yao and Yao [34] on sorting networks built from noisy comparators. There is also considerable work on searching when the total number of faulty comparisons is bounded rather than considering probabilistic noisy comparisons, including the work by Kenyon-Mathieu and Yao [15] and Rivest, Meyer, Kleitman, Winklmann, and Spencer [28]. Also of note is work by Karp and Kleinberg [14], who study binary searching for a value $x \in [0, 1]$ in an array of biased coins ordered by their biases.

Braverman and Mossel [2] introduce a persistent-error model, where comparison errors are persistently wrong with a fixed probability, $p < 1/2 - \varepsilon$, and they achieve a sorting algorithm that in our framework runs in $O(n^{3+f(p)})$ time, where $f(p)$ is some function of p, with maximum expected dislocation $O(\log n)$ and total dislocation $O(n)$. Klein, Penninger, Sohler, and Woodruff [17] improve the running time to $O(n^2)$, but with $O(n \log n)$ total dislocation w.h.p. The running time for sorting in the persistent-error model optimally with respect to maximum and total dislocation was subsequentially improved to $O(n^2)$, $O(n^{3/2})$, and ultimately to $O(n \log n)$, in a sequence of papers by Geissmann, Leucci, Liu, and Penna [7–9], all of which are not data oblivious because they make extensive use of noisy binary searching, which amounts to a random walk in a binary search tree.

Our Results. In this paper, we provide data-oblivious sorting algorithms that tolerate persistent noisy comparisons. In addition, we empirically compare our algorithms to other sorting algorithms, including the worst-case optimal algorithm, Riffle sort, by Geissmann, Leucci, Liu, and Penna [8], which is not data oblivious, but it achieves an optimal maximum and total dislocation under noisy comparisons. It runs in $O(n \log n)$ time, but it makes use of noisy binary search.

In addition to providing theoretical analysis for some of our algorithms, we empirically study all of our algorithms by measuring the effect of changing the amount of noise and the input size on the amount of dislocation, inversions, and number of comparisons. Our experiments show that for all of the data-oblivious algorithms we provide in this paper, the maximum and total dislocations are comparable to the optimal bounds of $O(\log n)$ and $O(n)$ respectively for the best algorithms that are not data oblivious. Moreover, we include experimental results for some standard sorting algorithms such as insertion sort, quick sort, and shell sort, for which we provide emipirical evidence that all of our algorithms significantly outperform these other algorithms in terms of the maximum and total dislocation metrics. These results indicate that our algorithms are able to combine the properties of both having a good tolerance to noisy comparisons while also being data-oblivious.

2 Window-Sort

Our first sorting algorithm is a version of window-sort [7], which will be useful as a subroutine in our other algorithms. We describe the pseudo-code at a high level in Algorithm 1, for approximately sorting an array of size n that has maximum dislocation at most $d_1 \leq n$ so that it will have maximum dislocation at most $d_2 = d_1/2^k$, for some integer $k \geq 1$, with high probability as a function of d_2.

Algorithm 1 Window-Sort($A = \{a_0, a_1, \ldots, a_{n-1}\}, d_1, d_2$).

```
1 for w ← 2d_1, d_1, d_1/2, ..., 2d_2 do
2     foreach i ← 0, 1, 2, ..., n − 1 do
3         r_i ← max{0, i − w} + |{a_j < a_i : |j − i| ≤ w}|
4     Sort A (deterministically) by nondecreasing r_i values (i.e., using r_i as the
        comparison key for a_i)
5 return A
```

In addition to implementing window-sort data obliviously, we provide a new analysis of window-sort, which allows us to apply it in new contexts. We begin this new analysis with the following lemma, which establishes the progress made in each iteration of window-sort.

▶ **Lemma 1.** *Suppose the comparison error probability, p_e, is at most $1/16$. If an array, A, has maximum dislocation at most d' prior to an iteration of window-sort for $w = 2d'$ (line 1 of Algorithm 1), then after this iteration, A will have maximum dislocation at most $d'/2$ with probability at least $1 - n2^{-d'/8}$.*

Proof. Let a_i be an element in A. Let W denote the window of elements in A for which we perform comparisons with a_i in this iteration; hence, $2d' \leq |W| \leq 4d'$. Because A has maximum dislocation d', by assumption, there are no elements to the left (resp., right) of W that are greater than a_i (resp., less than a_i). Thus, a_i's dislocation after this iteration depends only on the comparisons between a_i and elements in its window. Let X be a random variable that represents a_i's dislocation after this iteration, and note that $X \leq Y$, where Y is the number of incorrect comparisons with a_i performed in this iteration. Note further that we can write Y as the sum of $|W|$ independent indicator random variables and that $\mu = E[Y] = p_e|W| \leq d'/4$. Thus, if we let $R = d'/2$, then $R \geq 2\mu$; hence, we can use a Chernoff bound as follows:

$$\Pr(X > d'/2) \leq \Pr(Y > d'/2) = \Pr(Y > R) \leq 2^{-R/4} = 2^{-d'/8}.$$

Thus, with the claimed probability, the maximum dislocation for all elements of A will be at most $d'/2$, by a union bound. ◀

This implies the following.

▶ **Theorem 2.** *Suppose the comparison error probability, p_e, is at most $1/16$. If an array, A, of size n has maximum dislocation at most $d_1 \geq \log n$, then executing Window-Sort(A, d_1, d_2) runs in $O(d_1 n)$ time. Further, we can execute Window-Sort(A, d_1, d_2) data-obliviously to result in A having maximum dislocation of $d_2/2$ with probability at least $1 - 2n2^{-d_2/8}$, where $d_2 = d_1/2^k$, for some integer $k \geq 1$.*

Proof. For the running time and data obliviousness, note that we can perform the deterministic sorting step using a data-oblivious sorting algorithm (e.g., see [12]) in $O(n \log n)$ time. The windowed comparison steps (step 3 of Algorithm 1) are already data-oblivious and their running times form a geometric sum adding up to $O(d_1 n)$; hence, the total time for all the deterministic sorting steps (step 4 of Algorithm 1) is $O((\log(d_1/d_2))n \log n)$, which is at most $O(d_1 n)$ for $d_1 \geq \log n$.

For the maximum dislocation bound, note once $w = 2d_2$ and the array A prior to this iteration has maximum dislocation at most d_2, then it will result in having maximum dislocation at most $d_2/2$ with probability at least $1 - n2^{-d_2/8}$, by Lemma 1. Thus, by a union bound, the overall failure probability is at most

$$n\left(2^{-d_2/8} + 2^{-2d_2/8} + 2^{-4d_2/8} + \cdots + 2^{-d_1/8}\right) \;<\; n2^{-d_2/8}\sum_{i=0}^{\infty} 2^{-i} \;=\; 2n2^{-d_2/8}. \quad ◀$$

In terms of efficiency, we note that our data-oblivious implementation of window-sort is only time-efficient for small subarrays; hence, we need to do more work to design an efficient data-oblivious sorting algorithm.

3 Window-Merge-Sort

In this section, we describe a simple algorithm for sorting with noisy comparisons, which achieves a maximum dislocation of $O(\log n)$. Our window-merge-sort method is a windowed version of merge sort; hence, it is deterministic but not data oblivious. Nevertheless, it does avoid using noisy binary search.

Suppose we are given an array, A, of n elements (we use n to denote the original size of A, and N to denote the size of the subproblem we are currently working on recursively). Our method runs in $O(n \log^2 n)$ time and we give the pseudo-code for this method in Algorithm 2, with $d = c \log n$ for a constant $c \geq 1$ set in the analysis.

Our method begins by checking if the current problem size, N, satisfies $N \leq 6d$, in which case we're done. Otherwise, if $N > 6d$, then we divide A into 2 subarrays, A_1 and A_2, of roughly equal size and recursively approximately sort each one. For the merge of the two sublists, A_1 and A_2, we inductively assume that A_1 and A_2 have maximum dislocation at most $3d/2 = (3c/2)\log n$. We then copy the first $3d$ elements of A_1 and the first $3d$ elements of A_2 into a temporary array, S, and we note that, by our induction hypothesis, S contains the smallest $3d/2$ elements currently in A_1 and the smallest $3d/2$ elements currently in A_2. We then call Window-Sort$(S, 4d, d)$, and copy the first d elements from the output of this window-sort to the output of the merge, removing these same elements from A_1 and A_2. Then we repeat this merging process until we have at most $6d$ elements left in $A_1 \cup A_2$, in which case we call window-sort on the remaining elements and copy the result to the output of the merge. The following lemma establishes the correctness of this algorithm.

▶ **Lemma 3.** *If A_1 and A_2 each have maximum dislocation at most $3d/2$, then the merge of A_1 and A_2 has maximum dislocation at most $3d/2$ with probability at least $1 - N2^{-d/8}$.*

Algorithm 2 Window-Merge-Sort($A = \{a_0, a_1, \ldots, a_{N-1}\}, n, d$).

```
1  if N ≤ 6d then
2  |  return Window-Sort(A, 4d, d)
3  Divide A into two subarrays, A_1 and A_2, of roughly equal size
4  Window-Merge-Sort(A_1, n, d)
5  Window-Merge-Sort(A_2, n, d)
6  Let B be an initially empty output list
7  while |A_1| + |A_2| > 6d do
8  |  Let S_1 be the first min{3d, |A_1|} elements of A_1
9  |  Let S_2 be the first min{3d, |A_2|} elements of A_2
10 |  Let S ← S_1 ∪ S_2
11 |  Window-Sort(S, 4d, d)
12 |  Let B' be the first d elements of (the near-sorted) S
13 |  Add B' to the end of B and remove the elements of B' from A_1 and A_2
14 Call Window-Sort(A_1 ∪ A_2, 4d, d) and add the output to the end of B
15 return B
```

Proof. By Lemma 1 and a union bound, each of the calls to window-sort performed during the merge of A_1 and A_2 will result in an output with maximum dislocation at most $d/2$, with at least the claimed probability. So, let us assume each of the calls to window-sort performed during the merge of A_1 and A_2 will result in an output with maximum dislocation at most $d/2$. Consider, then, merge step i, involving the i-th call to Window-Sort$(S, 4d, d)$, where S consists of the current first $3d$ elements in A_1 and the current first $3d$ elements in A_2, which, by assumption, contain the current smallest $3d/2$ elements in A_1 and current smallest $3d/2$ elements in A_2. Thus, since this call to window-sort results in an array with maximum dislocation at most $d/2$, the subarray, B_i, of the d elements moved to the output in step i includes the $d/2$ current smallest elements in $A_1 \cup A_2$. Moreover, the first $d/2$ elements in B_i have no smaller elements that remain in S. In addition, for the $d/2$ elements in the second half of B_i, let S' denote the set of elements that remain in S that are smaller than at least one of these $d/2$ elements. Since the output of Window-Sort$(S, 4d, d)$ has maximum dislocation at most $d/2$, we know that $|S'| \leq d/2$ Moreover, the elements in S' are a subset of the smallest $d/2$ elements that remain in S and there are no elements in $(A_1 \cup A_2) - S$ smaller than the elements in S' (since S includes the $3d/2$ smallest elements in A_1 and A_2, respectively. Thus, all the elements in S' will be included in the subarray, B_{i+1}, of d elements output in merge step $i + 1$. In addition, a symmetric argument applies to the first $d/2$ elements with respect to the d elements in B_{i-1}. Therefore, the output of the merge of A_1 and A_2 will have maximum dislocation at most $3d/2$ with the claimed probability. ◀

Window-merge-sort clearly runs in $O(n \log^2 n)$ time. This gives us the following.

▶ **Theorem 4.** *Given an array, A, of n distinct comparable elements, one can deterministically sort A in $O(n \log^2 n)$ time subject to comparison errors with probability $p_e \leq 1/16$, so as to have maximum dislocation of $O(\log n)$ w.h.p., assuming that the block size B is at least $\log n$.*

This method is not data oblivious, however. For example, in a merge of two subarrays, A_1 and A_2, if each element in A_1 is less than all the elements in A_2, then with high probability the merge will take almost all the elements from A_1 before taking any elements from A_2.

4 Window-Oblivious-Merge-Sort

In this section, we describe a deterministic data-oblivious sorting algorithm that can tolerate noisy comparisons, which uses our data-oblivious window-sort only for small subarrays. Our method is an adaptation of the classic odd-even merge-sort algorithm [1] to the noisy comparison model, and it runs in $O(n \log^3 n)$ time, and achieves a maximum dislocation of $O(\log n)$, set in the analysis. We give our algorithm in Algorithm 3, with $d = c \log n$, where c is a constant set in the analysis.

Algorithm 3 Window-Odd-Even-Sort($A = \{a_0, a_1, \ldots, a_{N-1}\}, n, d$).

```
1  if N ≤ 6d then
2  |  return Window-Sort(A, 4d, d)
3  Divide A into two subarrays, A_1 and A_2, of roughly equal size
4  Window-Odd-Even-Sort(A_1, n, d)
5  Window-Odd-Even-Sort(A_2, n, d)
6  B ← Window-Merge(A_1, A_2, d)
7  return B
8
9  Window-Merge(A_1, A_2, d):
10 if |A_1| + |A_2| ≤ 6d then
11 |  return Window-Sort(A_1 ∪ A_2, 4d, d)
12 Let A_1^o (resp., A_1^e) be the subarray of A_1 of elements at odd (resp., even) indices
13 Let A_2^o (resp., A_2^e) be the subarray of A_2 of elements at odd (resp., even) indices
14 B_1 ← Window-Merge(A_1^e, A_2^e, d)
15 B_2 ← Window-Merge(A_1^o, A_2^o, d)
16 Let B be the shuffle of B_1 and B_2, so its even (resp., odd) indices are B_1 (resp., B_2)
17 for i = 0, 1, 2, ..., |B|/d do
18 |  Window-Sort(B[id : id + 6d], 4d, d)
19 return B
```

Note that, assuming d is $O(\log n)$, the running time for window-merge is characterized by the recurrence, $T(n) = 2T(n/2) + n \log n$, which is $O(n \log^2 n)$; hence, the running time for window-odd-even-sort is characterized by the recurrence, $T(n) = 2T(n/2) + n \log^2 n$, which is $O(n \log^3 n)$.

The correctness of window-merge is proved using induction and the 0-1 principle, which is that if a data-oblivious algorithm can sort an array of 0's and 1's, then it can sort any array[1] [18]. Let n be a power of 2, and consider the elements of each of A_1 and A_2 arranged in two columns with even indices in the left column and odd indices in the right column. (See Figure 1.) By the 0-1 principle, if A_1 and A_2 each have maximum dislocation at most d, then, for each arrangement of A_1 and A_2, the difference between the number of 1's in the left column and the number of 1's in the right column is at most $d + 1$.

Next stack the two-column arrangement of A_1 on top of that for A_2 and note that our window-merge algorithm recursively sorts each column, which, by induction will each have maximum dislocation d. That is, by the 0-1 principle, each column will consist of a contiguous

[1] It is straightforward to show that the 0-1 principle holds for our noisy sorting setting as well.

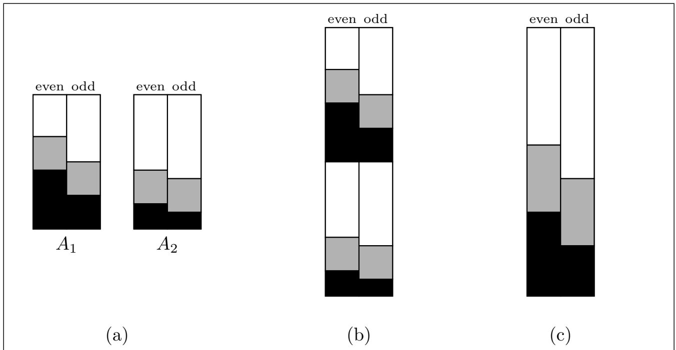


Figure 1 `Window-Merge` (a) Subarrays A_1 and A_2. (b) A before the merge. (c) A after the merge.

sequence of 0's, followed by a sequence of length at most $2d$ comprising a mixture of 0's and 1's, followed by a contiguous sequence of 1's. Further, by how we began our arrangement, the difference between the number of 1's in the left column and the number of 1's in the right column in the full arrangement of A_1 and A_2 is at most $2d + 2$. Thus, all the unsortedness is confined to a region of at most $4d + 2$ consecutively-indexed elements in the merged sequence, which are then completely contained in a region of $5d$ consecutively-indexed elements that begin at a multiple of d. Our window-merge method is guaranteed to call window-sort for this region of elements, bringing its maximum dislocation to be at most d. We observe that there are other calls to window-sort as well, but these will not degrade the sortedness of this region. Thus, the result is that the maximum dislocation of the merged list is at most d.

This gives us the following.

▶ **Theorem 5.** *Given an array, A, of n distinct comparable elements, one can deterministically and data-obliviously sort A in $O(n \log^3 n)$ time, subject to comparison errors with probability $p_e \leq 1/16$, so as to have maximum dislocation of $O(\log n)$ w.h.p.*

We note that the only randomization here is in the comparison model. The algorithm for Theorem 5 is deterministic. If we are willing to use a randomized algorithm, however, we can achieve a faster running time.

5 Randomized Shellsort

In this section, we describe a randomized data-oblivious sorting method that runs in $O(n \log n)$ time. The method is the simple randomized Shellsort algorithm of Goodrich [10], which we review in an appendix in Algorithm 4. It is based on peforming region compare-exhanges between subarrays of equal size, which, for a constant $c \geq 1$ set in the analysis, consists of constructing c random matchings between the elements of the two subarrays and performing compare-exchange operations between the matched elements. We study the dislocation reduction properties of randomized Shellsort empirically.

6 Annealing Sort

We briefly review here the annealing sort algorithm (see Algorithm 5 in an appendix), first introduced by Goodrich [11], which is a randomized data-oblivious sorting algorithm, and uses the simulated annealing meta-heuristic that involves following an **annealing schedule** defined by a **temperature sequence** $T = (T_1, T_2, \ldots T_t)$ and a **repetition sequence**

$R = (r_1, r_2, \ldots, r_t)$. This algorithm essentially uses a randomized round-robin strategy of scanning the input array A and performing, for each $i = 1, 2, \ldots, n$, a compare-exchange operation between $A[i]$ and $A[s]$ where s is a randomly chosen index not equal to i. At each round j, the temperature T_j is then used to determine how far apart the candidate comparison elements with indices i and s should be at each time step. Following the simulated annealing metaheuristic, the temperatures in the annealing schedule decrease over time, and each random choice is repeated r_j number of times in round j. In our experiments, we follow the same three-phase annealing schedule used in the analysis of this algorithm in [11].

7 Experiments

To empirically test the performance of our algorithms under persistent noisy errors, we implemented each of the algorithms described in Sections 2–6, along with riffleSort, which is a non-data-oblivious noisy sorting algorithm introduced by Geissman, Leucci, Liu, and Penna [8] that we review in an appendix in Algorithm 6. We also compare our algorithms to the standard and well-known insertion sort, randomized quicksort, and Shellsort [29] algorithms, e.g., see [3, 13]. For completeness, we include pseudo-code for these classic algorithms in an appendix in Algorithm 7.

We have also considered a variant of randomized Shellsort, which we denote by randomizedShellSortNo2s3s that does not include the 2 hop and 3 hop passes (lines 7-8 in Algorithm 4), as we do not think that they are necessary for the algorithm to perform well in practice. For standard Shell sort, we used the Pratt sequence [26], which uses a gap sequence consisting of all products of powers of 2 and 3 less than the array size, and we denote this algorithm by shellSortPratt.

Parameter configurations. The riffleSort algorithm uses a parameter c to determine the group sizes during noisy binary search. Geissmann, Leucci, Liu, and Penna [8] assume $c = 10^3$ in their analysis; however, we set $c = 5$ so that the algorithm works with the input sequence sizes we use. We also set a parameter, h, of riffle-sort, which affects the height of the noisy binary search tree, to be $\log(\lfloor \frac{n+1}{5d} \rfloor)$, where d is the maximum dislocation of the input sequence given to the noisy binary search tree. For all other parameters, we follow the values used in [8]. We have also made a significant and potentially risky modification to the noisy binary search algorithm described by Geissmann, Leucci, Liu, and Penna [8] so that it works in a practical setting. In particular, while the original description of this subroutine fixes an upper bound $\tau = \lfloor 240 \log n \rfloor$ on the total number of steps performed in a noisy binary search random walk, we found that this resulted in unreasonably long running times for the input sequence sizes we used, and we instead lowered this upper bound to $\tau = \lfloor 7 \log n \rfloor$ in our implementation. Despite using lower τ, the algorithm surprisingly produces good dislocation bounds, while taking significantly less time. Because of its reliance on noisy binary searching, riffle-sort is not data-oblivious, so we used its performance as the best achievable empirical dislocation bounds, which our data-oblivious methods compare against.

For annealing sort, we follow the annealing schedule and constants used in [4], which defines additional parameters h, g_{scale}, and finds suitable values for them alongside the existing parameters c and q defined by Goodrich [11], all of which affect the temperature and repetition schedules used in the algorithm; hence, we set $h = 1$, $g_{scale} = 0$, $c = 10$, and $q = 1$.

For WINDOWMERGESORT and WINDOWODDEVENMERGESORT, we set $d = \log n$. Though a larger constant multiple of $\log n$ is required for the theoretical proofs of these algorithms, we found that this wasn't necessary in practice; in fact we observed that $d = \log n$ resulted in lower inversions and dislocations in our experiments.

Lastly, for WINDOWSORT, we set $d_1 = n/2$ and $d_2 = \log n$, and for RANDOMIZEDSHELLSORT, we set $c = 4$.

Experimental setup. We implemented each algorithm in C++[2], and compared the performance of each algorithm by measuring the total dislocations, maximum dislocations, and the number of inversions of the output arrays, as well as the total number of pairwise comparisons that were done. Each data point in the following plots correspond to the average of 5 runs of the algorithm with random input sequences of integers.

To implement noisy persistent comparisons, we make use of tabulation hashing [22, 30]. In our tabulation hashing setting, we let f denote the number of bits to be hashed, and $s \leq f$ be a block size, and $t = \lceil f/s \rceil$ be the number of blocks. We initialize a two dimensional $t \times 2^s$ array, A, with random q bit integers. Given a key, c, with f bits, we partition f into t blocks of s bits. For our experiments, we set $f = 64$, $s = 8$, and $q = 14$. If c_i represents the i-th block, the hash value $h(c)$ will be derived using the lookup table as follows:

$$h(c) = A[0][c_0] \oplus A[1][c_1] \oplus \cdots \oplus A[t][c_t]$$

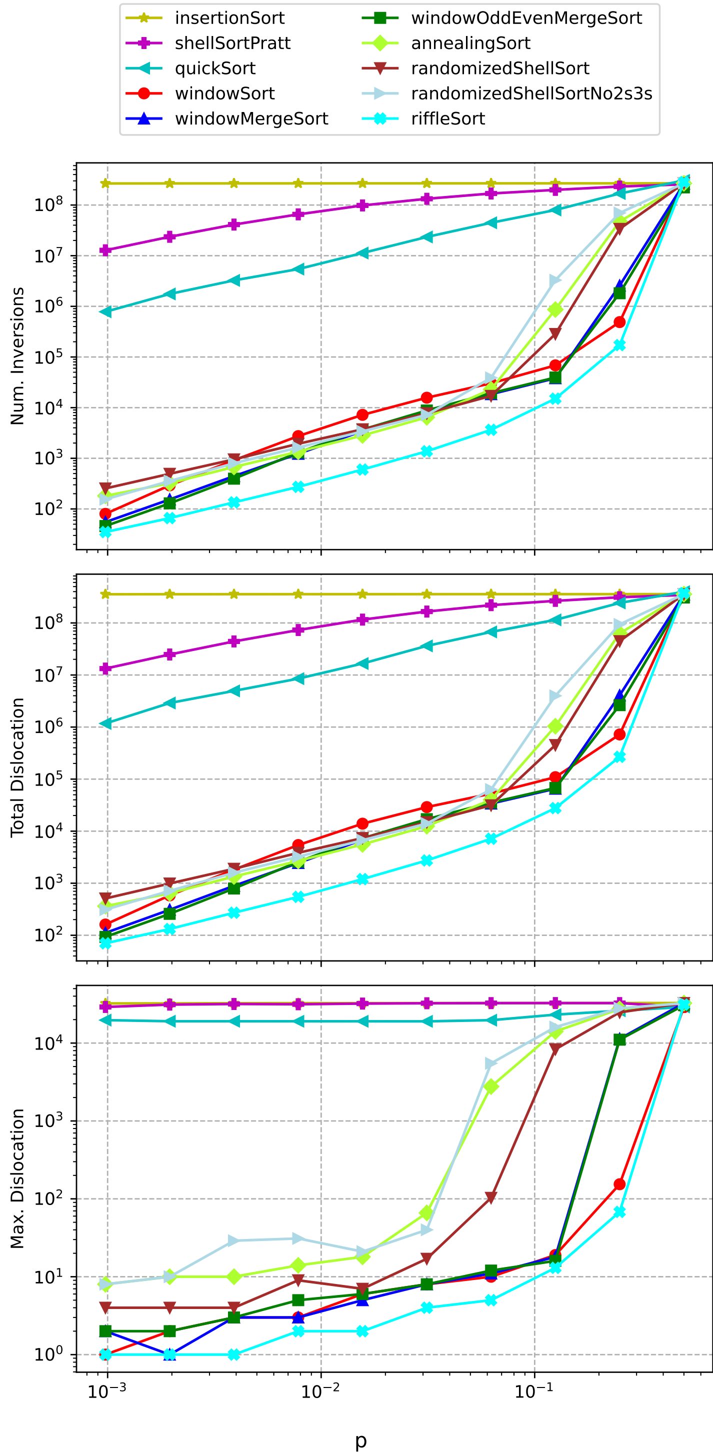
For simulating a noisy comparison, given two 4-Byte Integers, $x < y$, we first concatenate the numbers to get the key, $c = (x \cdot 2^{32}) + y$. Then, we hash c to derive $h(c)$, a random $q = 14$ bit integer. We determine that the comparison of these two numbers is noisy if and only if $h(c) \leq p \cdot 2^q$, where p is the noise probability, and output the result of the comparison accordingly.

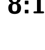
We performed two sets of experiments: one with a varying probability p of comparison error and fixed input size $n = 32768$, and the other with varying input size n and a fixed probability $p = 0.03$ of comparison error. In our experiments, p takes on values $(2^{-1}, 2^{-2} \ldots, 2^{-10})$, and n takes on values $(2^{16}, 2^{15} \ldots, 2^7)$.

Results and analysis. We first consider experiments with varying p, and compare the maximum dislocations, total dislocations and inversions between each algorithm. We see from Figure 2 that all of the data-oblivious algorithms we describe in this paper have maximum and total dislocations that are inline with the theoretical optimal bounds of $O(\log n)$ and $O(n)$ respectively, as well as RIFFLESORT, particularly when $p < 0.1$. For example, we see that WINDOWODDEVENMERGESORT tends to be the best-performing data-oblivious algorithm for different values of p, achieving a total dislocation of at most $\approx 35\,300$, a maximum dislocation of at most 12, and at most $\approx 19\,200$ total inversions for values of $p < 0.1$.

We see that all of the non-standard algorithms tend to form an S-shaped curve, in terms of their dislocation bounds, such that as p starts to increase, the number of dislocations and inversions start to increase slowly, then there is a sharper increase after we reach $p > 0.1$. As expected, we see that the highest dislocation and inversions is when $p = 0.5$, which is the worst-case scenario for p (for any value of $p > 0.5$, reversing the output should result

[2] Our implementations of all algorithms can be found at `https://github.com/UC-Irvine-Theory/NoisyObliviousSorting`.



 **Figure 2** Effect of varying the comparison error probability p on the inversion and dislocation counts, with input sequences of size 32768.

in a sequence with lower dislocation). In particular, as p goes from 1/32 to 1/2, all of the non-standard algorithms go from having up to 100 maximum dislocation and $\approx 28\,900$ total dislocation to having up to $\approx 30\,000$ maximum dislocation and ≈ 345 million total dislocation. These results match our theoretical analyses in this paper, as we assume that $p \leq 1/16$ in order to prove bounds for the dislocation. The proof for the version of riffleSort we use assumes similar bounds for p [8]. On the other hand, we see that our implementations of insertion sort, quick sort, and Shell sort do not have the tendency to form an S-curve, and their inversion and dislocation counts are significantly higher compared to our algorithms.

In Figure 3, we see the effect of varying p and n on the number of comparisons made during the algorithm. We see that the number of comparisons tends to grow smaller as p increases in riffleSort, insertionSort and shellSortPratt. When the input size is varied, we see that riffleSort, insertionSort and quickSort use the fewest number of comparisons. Notably, we see that randomizedShellSort (and its variant without 2 and 3-hop passes), as well as annealingSort, are the best-performing data-oblivious algorithms in terms of the number of comparisons. Overall, we found that riffleSort was the best-performing algorithm in both sets of experiments; however, it uses the noisy binary search subroutine and is thus not a data-oblivious algorithm.

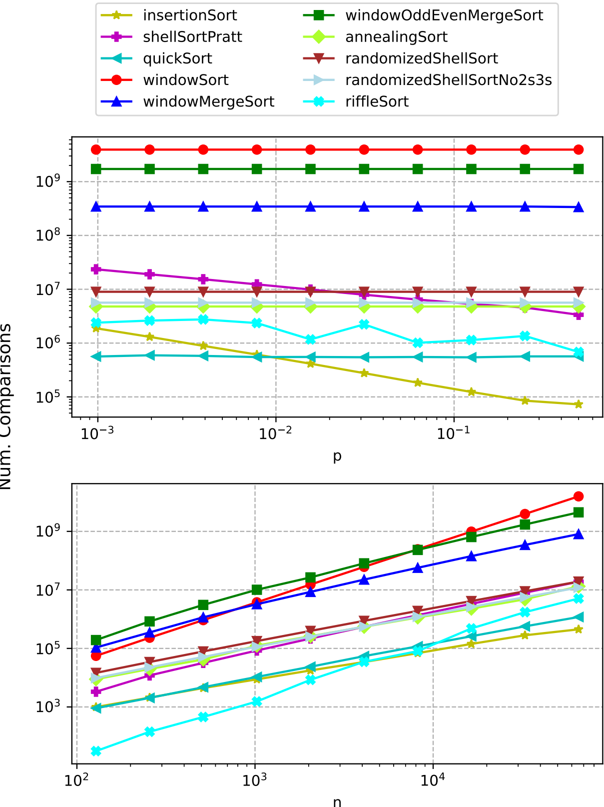


Figure 3 Effect of varying the comparison error probability p and the input size n on the number of comparisons.

We also consider how the dislocation is distributed across the output array for each algorithm. In Figure 4, we see the average dislocation across different array indices for 5 runs of each algorithm with input sequences of size 16384, and $p = 0.03$. For each output array,

we grouped the indices into 128 bins and took the average dislocation inside each bin. From this figure we can see the significant difference in dislocation counts between the standard sorting algorithms insertion sort, Shellsort and quick sort, compared to the other algorithms we implemented. All of the standard sorting algorithms have bins with over 2000 dislocation on average, whereas none of the other algorithms have any bins with over 1.2 dislocations on average, with riffleSort having less than 0.2 dislocations on average accross all of its bins.

While the distribution of dislocation is similar accross most algorithms, we see that insertion sort has most of its dislocation at the two ends of the array, whereas quickSort has a few bins with high dislocation and has lower dislocation for most of the remaining bins.

8 Conclusions and Future Work

We introduced the sorting algorithms Window-Merge-Sort and Window-Odd-Even-Sort, both of which are tolerant to noisy comparisons, with the latter also being data-oblivious, with the key difference from existing algorithms being that we do not require use of a noisy binary search subroutine for either algorithms. We then provided both theoretical and experimental analyses, comparing our algorithms to some standard well-known sorting algorithms, and saw that our algorithms perform well in a practical setting as well. Interestingly, we found that the data-oblivious algorithms Annealing sort and Randomized Shellsort performed quite well under noisy comparisons in our experiments, though we have not provided a theoretical analysis for either of these algorithms. Therefore one possible direction for future work could be to prove similar bounds for these two algorithms.

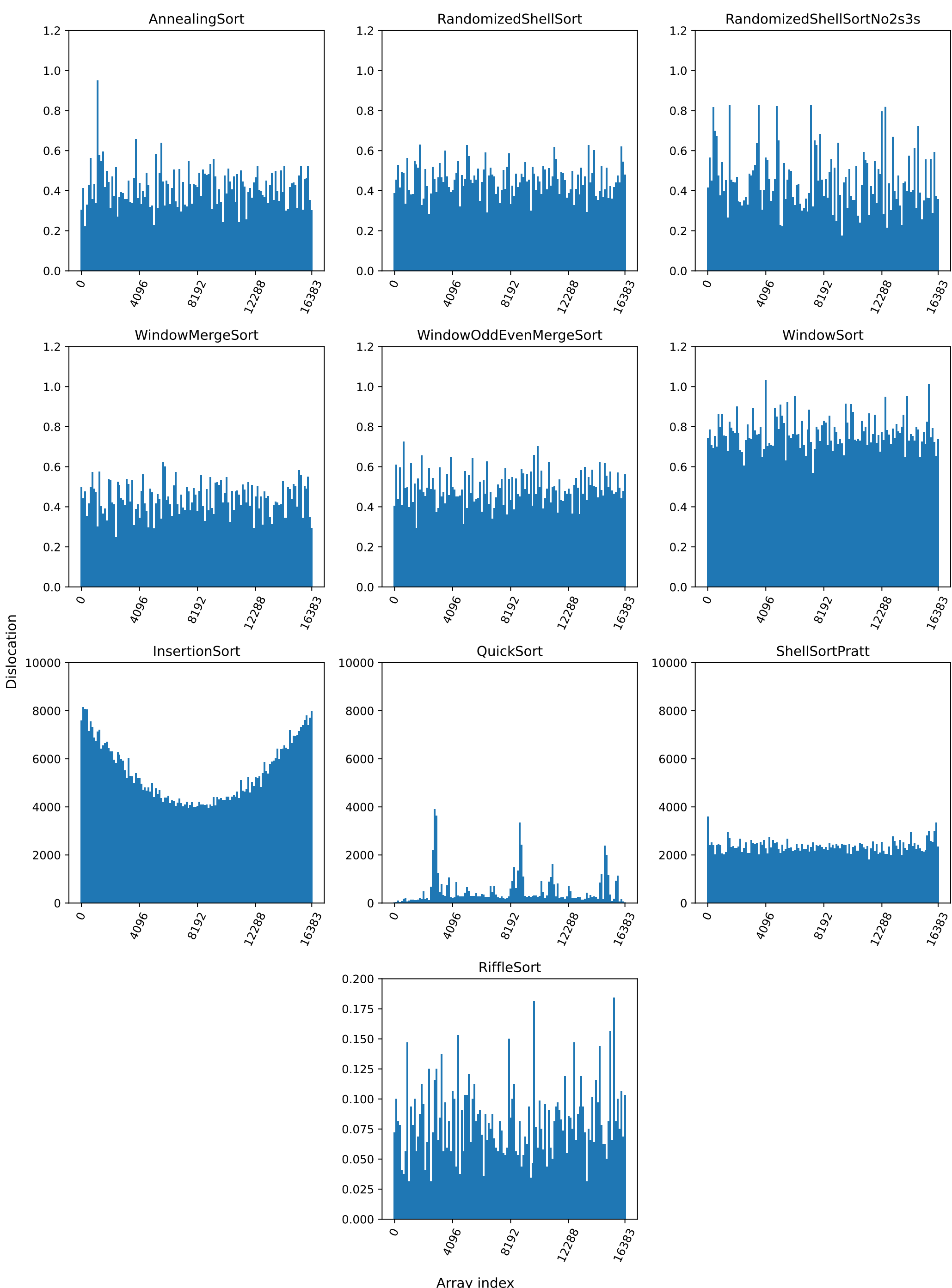


Figure 4 Averaged dislocation counts at different array indices over 5 runs for each algorithm on input sequences of size 16384, and $p = 0.03$. Each bar in the histogram corresponds to a bin of 128 indices.

References

1. Kenneth E Batcher. Sorting networks and their applications. In *Proc. of the Spring Joint Computer Conference (AFIPS)*, pages 307–314. ACM, 1968. doi:10.1145/1468075.1468121.
2. Mark Braverman and Elchanan Mossel. Noisy sorting without resampling. In *19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 268–276, 2008.
3. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 4/e edition, 2022.
4. Kris Vestergaard Ebbesen. On the practicality of data-oblivious sorting. Master's thesis, Aurhus Univ., Denmark, 2015.
5. Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23(5):1001–1018, 1994.
6. Marc Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In David Naccache, editor, *Topics in Cryptology CT-RSA*, pages 457–471. Springer, 2001.
7. Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Sorting with recurrent comparison errors. In Yoshio Okamoto and Takeshi Tokuyama, editors, *28th Int. Symp. on Algorithms and Computation (ISAAC)*, volume 92 of *LIPIcs*, pages 38:1–38:12, 2017. doi:10.4230/LIPIcs.ISAAC.2017.38.
8. Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Optimal sorting with persistent comparison errors. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th European Symposium on Algorithms (ESA)*, volume 144 of *LIPIcs*, pages 49:1–49:14, 2019.
9. Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Optimal dislocation with persistent errors in subquadratic time. *Theory of Computing Systems*, 64(3):508–521, 2020. This work appeared in preliminary form in STACS'18.
10. Michael T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6):27:1–27:26, December 2011. doi:10.1145/2049697.2049701.
11. Michael T. Goodrich. Spin-the-bottle sort and annealing sort: Oblivious sorting via round-robin random comparisons. *Algorithmica*, pages 1–24, 2012. doi:10.1007/s00453-012-9696-5.
12. Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *46th ACM Symposium on Theory of Computing (STOC)*, pages 684–693, 2014. doi:10.1145/2591796.2591830.
13. Michael T Goodrich and Roberto Tamassia. *Algorithm Design and Applications*, volume 363. Wiley, 2015.
14. Richard M Karp and Robert Kleinberg. Noisy binary search and its applications. In *18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 881–890, 2007.
15. Claire Kenyon-Mathieu and Andrew C Yao. On evaluating boolean functions with unreliable tests. *International Journal of Foundations of Computer Science*, 1(01):1–10, 1990.
16. Kamil Khadiev, Artem Ilikaev, and Jevgenijs Vihrovs. Quantum algorithms for some strings problems based on quantum string comparator. *Mathematics*, 10(3):377, 2022.
17. Rolf Klein, Rainer Penninger, Christian Sohler, and David P. Woodruff. Tolerant algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *European Symposium on Algorithms (ESA)*, pages 736–747. Springer, 2011.
18. Donald E Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.
19. Tom Leighton, Yuan Ma, and C. Greg Plaxton. Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. *Journal of Computer and System Sciences*, 54(2):265–304, 1997. doi:10.1006/jcss.1997.1470.
20. Wen Liu, Shou-Shan Luo, and Ping Chen. A study of secure multi-party ranking problem. In *Eighth ACIS Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*, volume 2, pages 727–732, 2007. doi:10.1109/SNPD.2007.367.

21 Cheng Mao, Jonathan Weed, and Philippe Rigollet. Minimax rates and efficient algorithms for noisy sorting. In Firdaus Janoos, Mehryar Mohri, and Karthik Sridharan, editors, *Proceedings of Algorithmic Learning Theory*, volume 83 of *Proceedings of Machine Learning Research*, pages 821–847, 2018.

22 Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3):1–50, June 2012. doi:10.1145/2220357.2220361.

23 Andrzej Pelc. Searching with known error probability. *Theoretical Computer Science*, 63(2):185–202, 1989. doi:10.1016/0304-3975(89)90077-7.

24 Andrzej Pelc. Searching games with errors—fifty years of coping with liars. *Theoretical Computer Science*, 270(1):71–109, 2002. doi:10.1016/S0304-3975(01)00303-6.

25 Nicholas Pippenger. On networks of noisy gates. In *26th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 30–38, 1985. doi:10.1109/SFCS.1985.41.

26 Vaughan Ronald Pratt. *Shellsort and sorting networks*. PhD thesis, Stanford University, Stanford, CA, USA, 1972.

27 Alfréd Rényi. On a problem in information theory. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 6:505–516, 1961. See https://mathscinet.ams.org/mathscinet-getitem?mr=0143666.

28 R.L. Rivest, A.R. Meyer, D.J. Kleitman, K. Winklmann, and J. Spencer. Coping with errors in binary search procedures. *Journal of Computer and System Sciences*, 20(3):396–404, 1980. doi:10.1016/0022-0000(80)90014-8.

29 D. L. Shell. A high-speed sorting procedure. *Comm. ACM*, 2(7):30–32, July 1959. doi:10.1145/368370.368387.

30 Mikkel Thorup. Fast and powerful hashing using tabulation. *Commun. ACM*, 60(7):94–101, June 2017. doi:10.1145/3068772.

31 Ziao Wang, Nadim Ghaddar, and Lele Wang. Noisy sorting capacity. *arXiv*, abs/2202.01446, 2022. arXiv:2202.01446.

32 Ya Xu, Nanyu Chen, Addrian Fernandez, Omar Sinno, and Anmol Bhasin. From infrastructure to culture: A/B testing challenges in large scale social networks. In *21th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 2227–2236, 2015. doi:10.1145/2783258.2788602.

33 Andrew C. Yao. Protocols for secure computations. In *23rd IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 160–164, 1982. doi:10.1109/SFCS.1982.38.

34 Andrew C. Yao and F. Frances Yao. On fault-tolerant networks for sorting. *SIAM Journal on Computing*, 14(1):120–128, 1985. doi:10.1137/0214009.

A Some Existing Sorting Algorithms

In this section, we review some existing sorting algorithms that we included in our tests.

A.1 Randomized Shellsort

The first existing sorting algorithm we review is the randomized Shellsort of Goodrich [10], which we give in Algorithm 4. This algorithm is data oblivious.

Algorithm 4 Random-Shellsort($A = \{a_0, a_1, \dots, a_{n-1}\}$).

```
1  for o = n/2, n/2^2, n/2^3, ..., 1 do
2      Let A_i denote subarray A[io .. io + o − 1], for i = 0, 1, 2, ..., n/o − 1.
3      begin a shaker pass
4          Region compare-exchange A_i and A_{i+1}, for i = 0, 1, 2, ..., n/o − 2.
5          Region compare-exchange A_{i+1} and A_i, for i = n/o − 2, ..., 2, 1, 0.
6      begin an extended brick pass
7          Region compare-exchange A_i and A_{i+3}, for i = 0, 1, 2, ..., n/o − 4.
8          Region compare-exchange A_i and A_{i+2}, for i = 0, 1, 2, ..., n/o − 3.
9          Region compare-exchange A_i and A_{i+1}, for even i = 0, 1, 2, ..., n/o − 2.
10         Region compare-exchange A_i and A_{i+1}, for odd i = 0, 1, 2, ..., n/o − 2.
```

A.2 Annealing Sort

The next existing sorting algorithm we review is the annealing-sort method of Goodrich [11], which we review in Algorithm 5. This algorithm is also data oblivious.

Algorithm 5 Annealing-Sort($A = \{a_0, a_1, \dots, a_{N-1}\}, n, T, R$).

```
1  for j = 1, 2, ..., t do
2      for i = 1, ..., n − 1 do
3          for k = 1, 2, ..., r_j do
4              Let s be a random integer in the range [i + 1, min(n, i + T_j)]
5              if A[i] > A[s] then
6                  Swap A[i] and A[s]
7      for i = n, n − 1, ... 2 do
8          for k = 1, 2, ..., r_j do
9              Let s be a random integer in the range [max(1, i − T_j), i − 1]
10             if A[s] > A[i] then
11                 Swap A[i] and A[s]
```

A.3 Riffle Sort

We include pseudo-code for the riffle-sort method of Geissman, Leucci, Liu, and Penna [8], which we review in Algorithm 6, for $k = (\log n)/2$ and $\gamma = 2020$. The pseudo-code uses a subroutine TEST(x, v) (see [8], Definition 1), which checks whether some element x approximately belongs to the interval pointed to by some node v in the noisy binary search tree, which is the main place where this algorithm is not data oblivious.

Algorithm 6 Riffle-Sort$(A = \{a_0, a_1, \ldots, a_{n-1}\})$.

```
1  T_0, T_1, ..., T_k ← Partition(A)
2  S_0 ← WindowSort(T_0, √n, 1)
3  for j = 1, ..., k+1 do
4      S_i ← Merge(S_i, T_{i-1})
5      S_i ← WindowSort(S_i, 9γ log n, 1)
6  return S_{k+1}
7
8  Partition(A) :
9  for i = k, ..., 1 do
10     T_i ← 2^{i-1}√n elements chosen u.a.r. from A \ {T_{i+1}, ..., T_k}
11 T_0 ← remaining √n elements in A
12 return T_0, ..., T_k
13
14 Merge(A, B) :
15 foreach x ∈ B do
16     rank_x ← NoisyBinarySearch(A, x)
17 Insert simultaneously all elements x ∈ B according to rank_x into A
18 return A
19
20 NoisyBinarySearch(A, x) :
21 Construct noisy binary search trees T_0, T_1 as described in [8], section 3.1.
22 for j = 0, 1 do
23     t ← 7⌊log |A|⌋
24     curr ← T_j.root
25     while t > 0 do
26         if curr is a leaf of T_j then
27             return curr
28         Call test(x, c) for each child c of node curr.
29         if exactly one of the calls pass for some child node c then
30             curr ← c
31         else
               // all tests have failed
32             curr ← curr.parent
33         t ← t − 1
34 return an arbitrary index                    // both walks have timed out
```

A.4 Well-known Sorting Algorithms

For the sake of completeness, we also include pseudo-code for the well-known insertion-sort, quick-sort, and Shellsort algorithms, in Algorithm 7. None of these three algorithms are data oblivious. One can modify insertion-sort to be data oblivious, however, by continuing the compare-and-swap inner loop process to the beginning of the array in every iteration. Likewise, the Shellsort algorithm can also be modified to be data oblivious in the same manner, since its inner loop is essentially an insertion-sort carried out across elements separated by the gap distance in each iteration.

Algorithm 7 Well-known sorting algorithms, assuming the input array, A, is of size n and indexed starting at 0. We sort A by calling Insertion-Sort(A, n), Quick-sort$(A, 0, n-1)$, or Shell-sort(A, n, G), where G is a non-increasing ***gap sequence*** of positive integers less than n, such as the Pratt sequence [26], which consists of all products of powers of 2 and 3 less than n.

```
Insertion-sort(A, n):
for i ← 1, ..., n − 1 do
    j ← i
    while j > 0 and A[j − 1] > A[j] do
        Swap A[j] and A[j − 1]
        j ← j − 1

Quick-sort(A, l, h):
if l < h then
    Choose x uniformly at random from the subarray A[l..h]
    Partition A into A[l..p − 1], A[p], and A[p + 1..h], where A[i] < x for i ∈ [l, p − 1],
      A[p] = x, and A[i] ≥ x for i ∈ [p + 1, h] (if these subarrays exist)
    Quick-sort(A, l, p − 1)
    Quick-sort(A, p + 1, h)

Shell-sort(A, n, G):
foreach g ∈ G do
    for i ← g, ..., n − 1 do
        j ← i
        while j ≥ g and A[j − g] > A[j] do
            Swap A[j] and A[j − g]
            j ← j − g
```

Optimizing over the Efficient Set of a Multi-Objective Discrete Optimization Problem

Satya Tamby
Université Paris Dauphine, PSL Research University, LAMSADE, France

Daniel Vanderpooten[1]
Université Paris Dauphine, PSL Research University, LAMSADE, France

Abstract

Optimizing over the efficient set of a discrete multi-objective problem is a challenging issue. The main reason is that, unlike when optimizing over the feasible set, the efficient set is implicitly characterized. Therefore, methods designed for this purpose iteratively generate efficient solutions by solving appropriate single-objective problems. However, the number of efficient solutions can be quite large and the problems to be solved can be difficult practically. Thus, the challenge is both to minimize the number of iterations and to reduce the difficulty of the problems to be solved at each iteration.

In this paper, a new enumeration scheme is proposed. By introducing some constraints and optimizing over projections of the search region, potentially large parts of the search space can be discarded, drastically reducing the number of iterations. Moreover, the single-objective programs to be solved can be guaranteed to be feasible, and a starting solution can be provided allowing warm start resolutions. This results in a fast algorithm that is simple to implement.

Experimental computations on two standard multi-objective instance families show that our approach seems to perform significantly faster than the state of the art algorithm.

2012 ACM Subject Classification Applied computing → Multi-criterion optimization and decision-making; Theory of computation → Integer programming

Keywords and phrases discrete optimization, multi-objective optimization, non-dominated set, efficient set

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.9

1 Introduction

For problems involving multiple objectives, solutions of interest are *efficient* solutions for which there is no other solution which dominates it, meaning that it is at least as good on all objectives and strictly better on at least one objective. The resulting efficient set is often of large cardinality for multi-objective discrete problems, and in particular multi-objective combinatorial optimization (MOCO) problems. In order to discriminate among efficient solutions, a natural approach is to optimize, over the efficient set, a value function Φ which represents a major objective or the preferences of a specific decision maker. A special case of interest is the determination of the nadir point which, when considering objectives to be minimized, corresponds to the worst values achieved by efficient solutions for each objective. This valuable information allowing a decision maker to better appreciate the values that he/she could expect, can indeed be seen as maximizing independently each objective function over the efficient set.

When the function Φ to be optimized guarantees to return an efficient solution (e.g. when Φ is a positively weighted sum of the objective functions), optimizing Φ over the efficient set can be performed by optimizing Φ over the feasible set. In general, however, optimizing

[1] corresponding author

© Satya Tamby and Daniel Vanderpooten;
licensed under Creative Commons License CC-BY 4.0
21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis; Article No. 9; pp. 9:1–9:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Φ directly over the feasible set will return a dominated solution. The difficulty stems from the fact that, unlike the feasible set, the efficient set is not explicitly defined by a set of constraints.

A trivial approach consists of enumerating the entire efficient set, computing the image of each solution through function Φ before finding the optimal one. However, as mentioned before, this is not a convenient approach since computing the efficient set can be intractable due to its large cardinality. For this reason, most approaches, including ours, try to find an optimal solution by enumerating the smallest possible subset of efficient solutions.

After stating the problem formally in Section 2, we briefly review the literature indicating the positioning of our approach among existing approaches (Section 3). We then state some preliminary results (Section 4) before presenting our algorithm in Section 5. Experimental results on two standard MOCO problems are then reported in Section 6. Some conclusions and perspectives are finally presented.

2 Problem statement

In the following, vectors are written in bold contrarily to scalars. Components of vectors are specified as indices.

2.1 Basic definitions and notations

Given a discrete set $\mathcal{X}$ of feasible solutions, defined by constraints on n decision variables, and p objective functions or criteria $\boldsymbol{f} = (f_1, \dots, f_p)$,

we consider the following multi-objective problem:

$$\text{(MOP)} \qquad \begin{cases} \min & \boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), \dots, f_p(\boldsymbol{x})) \\ \text{s.t.} & \boldsymbol{x} \in \mathcal{X} \end{cases}$$

For any feasible solution $\boldsymbol{x} \in \mathcal{X}$, its image $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$ is referred to as a *feasible point* and $\mathcal{Y} = \boldsymbol{f}(\mathcal{X})$ denotes the set of feasible points. In this setting $\mathbb{R}^n$ and $\mathbb{R}^p$, will be referred to as the *decision space* and the *objective space*, respectively.

Given p dimensional points in $\mathbb{R}^p$, we consider the following binary relations; they are respectively referred to as *(Pareto) dominance*, *strong dominance* and *weak dominance*:

$$\begin{aligned} \boldsymbol{y} \leq \boldsymbol{y}' &\iff \begin{cases} \boldsymbol{y}_i \leq \boldsymbol{y}'_i & \forall i \in \{1, \dots, p\} \\ \boldsymbol{y} \neq \boldsymbol{y}' \end{cases} \\ \boldsymbol{y} \prec \boldsymbol{y}' &\iff \boldsymbol{y}_i < \boldsymbol{y}'_i \qquad \forall i \in \{1, \dots, p\} \\ \boldsymbol{y} \leqq \boldsymbol{y}' &\iff \boldsymbol{y}_i \leq \boldsymbol{y}'_i \qquad \forall i \in \{1, \dots, p\} \end{aligned}$$

The set $\mathcal{Y}_N$, which contains the points that are non-dominated, is defined by: $\mathcal{Y}_N = \{\boldsymbol{y} \in \mathcal{Y}, \nexists \boldsymbol{y}' \in \mathcal{Y}, \boldsymbol{y}' \leq \boldsymbol{y}\}$. The subset of feasible solutions that lead to a non-dominated point is referred to as *the efficient set* and is denoted by $\mathcal{X}_E = \boldsymbol{f}^{-1}(\mathcal{Y}_N)$. It should be observed that several efficient solutions may correspond to the same non-dominated point. Solving problem (MOP) is then usually understood as determining $\mathcal{Y}_N$ and providing one efficient solution associated with each non-dominated point in $\mathcal{Y}_N$. Many algorithms have been proposed for solving problem (MOP) in the discrete case including [9, 13, 12, 6, 2, 14]. As will be seen in Section 3, algorithms for optimizing over the efficient set have been strongly influenced by these algorithms.

Finally, $\boldsymbol{y}_{-k} \in \mathbb{R}^{p-1}$ denotes the *projection of* $\boldsymbol{y}$ *in the direction* k *i.e.* the point $\boldsymbol{y}$ where component k has been omitted, that is $\boldsymbol{y}_{-k} = (\boldsymbol{y}_1, \dots, \boldsymbol{y}_{k-1}, \boldsymbol{y}_{k+1}, \dots, \boldsymbol{y}_p)$.

2.2 Problem statement

Given a function $\Phi : \mathcal{X} \to \mathbb{R}$ to be minimized, the *problem of optimizing* Φ *over the efficient set of* X can be stated as follows:

$$(\text{MOP}_E) \qquad \begin{cases} \min & \Phi(\boldsymbol{x}) \\ \text{s.t.} & \boldsymbol{x} \in \mathcal{X}_E \end{cases}$$

The difficulty of this problem stems from the fact that $\mathcal{X}_E$ is not characterized explicitly, *i.e.* as a set of constraints. Note that under certain assumptions on Φ, this problem amounts to optimizing over the *feasible set*, making the problem much simpler. In particular, this is the case when the optima of Φ are guaranteed to be non-dominated, as stated later in Theorem 2. In general, however, optimizing over the feasible set returns a solution which is not efficient and provides a lower bound on Φ which may be very far from the optimal value.

3 Related works and contribution of this paper

Approaches optimizing over the efficient set usually rely on the concept of *search region* that has been formalized in [8, 4], which is described in Section 4.1. Informally, the search region associated to a set of points N corresponds to the subset of the objective space containing points not dominated by any point in N.

Most methods dealing with the discrete case for problem (MOP_E) follow the same general scheme. They iteratively minimize Φ over the current search region to obtain a candidate point $\boldsymbol{y}$. Since $\boldsymbol{y}$ is potentially dominated, an additional effort is required to check the Pareto-optimality of the candidate. This step is usually performed by solving a program leading to a point $\boldsymbol{y}' \in \mathcal{Y}_N$ that dominates $\boldsymbol{y}$. Among all efficient solutions $\boldsymbol{x}'$ corresponding to $\boldsymbol{y}'$, *i.e.* such that $\boldsymbol{x}' \in \boldsymbol{f}^{-1}(\boldsymbol{y}')$, one optimizing function Φ is selected and retained if it improves the current best value of Φ. Convergence is reached when there is no feasible point in the current search region or when the solution of the first phase is non-dominated.

The evolution of the proposed methods for optimizing over the efficient set of a discrete multi-objective problem (problem (MOP_E)) follows the evolution of the proposed methods for generating the non-dominated set of a discrete multi-objective problem (problem (MOP)) and is actually related to the evolution of the way of representing the search region.

The oldest methods for solving problem (MOP) , such as [9, 13], used a complete and implicit representation of the search region. This involves imposing constraints stating that the new non-dominated point to be generated should improve on at least one objective with respect to *all* non-dominated points previously generated. This may be achieved by adding disjunctive constraints as shown in [13]. While quite easy to implement, the main drawback of this approach is the growth of the number of constraints which makes it impossible to solve other than small size instances. Similarly, the oldest methods for solving problem (MOP_E) , such as [5, 3] use a complete and implicit representation of the search region, with the same drawbacks as for problem (MOP) .

The current methods for solving problem (MOP) resort to a decomposition of the search region into a union of search zones which allows solving at each iteration problems of constant size testing the existence of new non-dominated points in a search zone. The clear advantage is that the required optimization is relatively fast. The corresponding algorithms, such as [12, 6, 2, 14], mostly differ on how *search zones are defined* (note that sometimes a superset of the search region is stored), how *search zones are explored* (*i.e.* which search zone should be explored first and how) and how the *search region is updated* (*i.e.* how to modify the search zones so as to remove the part dominated by a new point) - see [14] for more details.

Following this evolution, the most recent algorithms for solving problem (MOP_E) resort to a decomposition of the search region, and were often proposed by the same authors who adapted their decomposition approach to problem (MOP_E) [1, 11] or to its special case of determining the nadir point [10, 7]. Similarly the algorithm proposed in this paper can be seen as an adaptation of our previous algorithm for problem (MOP) presented in [14]. While preserving the positive features of our previous approach (the definition of rules allowing many search zones to be discarded without exploring them, the guarantee that the required optimization problems are feasible and the existence of an initial feasible solution provided to the solver, which considerably speeds up the solution times,...), our adaptation also includes new positive features specific to problem (MOP_E). In particular, by focusing on the iterative improvement of function Φ, we define new rules to discard additional search zones which cannot contain efficient solutions improving Φ.

4 Preliminary results

4.1 Search region, search zones

Given a set of N points, the corresponding search region denoted by $S(N)$ corresponds to the set of points that are not dominated by a point of N, *i.e.*

$$S(N) = \{\boldsymbol{y} \in \mathbb{R}^p : \nexists \bar{\boldsymbol{y}} \in N, \bar{\boldsymbol{y}} \leqq \boldsymbol{y}\}$$

The search region, which describes the part of the objective space where undiscovered non-dominated points may lie, can be defined as a union of *search zones* delimited by *local upper bounds* (see [8] for more details). Denoting $U(N)$ as the set of these local upper bounds, we have then:

$$\boldsymbol{y} \in S(N) \iff \exists \mathbf{u} \in U(N) : \boldsymbol{y} \prec \mathbf{u}$$

When a new point $\boldsymbol{y}$ is found, the search region must be updated by removing the part dominated by $\boldsymbol{y}$. This is done by splitting each zone strictly dominated by $\boldsymbol{y}$ into p new zones, referred to as *children*. Child i of $\mathbf{u}$ is $\mathbf{u}^i = (\mathbf{u}_1, \ldots, \mathbf{u}_{i-1}, \boldsymbol{y}_i, \mathbf{u}_{i+1}, \ldots, \mathbf{u}_p)$.

Updating the search region can lead to redundancies, *i.e.* zones that are included in others. Since only maximal zones are required to represent the search region, [8, 4] have proposed some methods to avoid generating redundant zones. One of these relies on the identification of the *defining points*, that are the points in N which define the components of the local upper bounds.

▶ **Definition 1.** *A point $\boldsymbol{y} \in N$ is a* defining point *for the component i of $\mathbf{u}$ if and only if $\boldsymbol{y}_i = \mathbf{u}_i$ and $\boldsymbol{y}_{-i} \prec \mathbf{u}_{-i}$.*

The following result allows the efficient identification of maximal local upper bounds.

▶ **Theorem 1** ([8]). *$\mathbf{u}$ is maximal if there exists at least one defining point for every bounded component of $\mathbf{u}$.*

In the following, $\mathcal{D}_i(\mathbf{u})$ denotes the set of defining points of $\mathbf{u}_i$.

4.2 Finding a non-dominated point

A well known theorem in multi-objective optimization states that some functions are guaranteed to lead to a non-dominated point when being optimized. Such functions are called *strongly monotone* and preserve the Pareto-dominance. More formally:

▶ **Definition 2.** *A function* $g : \mathcal{Y} \mapsto \mathbb{R}$ *is said to be* strongly monotone *if and only if*

$$\forall (\boldsymbol{y}, \boldsymbol{y}') \in \mathcal{Y}^2, \boldsymbol{y} \preceq \boldsymbol{y}' \implies g(\boldsymbol{y}) < g(\boldsymbol{y}')$$

▶ **Theorem 2.** *Let* g *be a strongly monotone function and* $\mathbf{u} \in \mathbb{R}^p$. *Then, if problem* $\{\min g(\boldsymbol{y}) : \boldsymbol{y} \in \mathcal{Y}, \boldsymbol{y} \leqq \mathbf{u}\}$ *admits* $\boldsymbol{y}^*$ *as an optimal solution, then* $\boldsymbol{y}^* \in \mathcal{Y}_N$.

Proof. Due to the strong monotonicity of g, any point $\bar{\boldsymbol{y}} \in Y$ dominating $\boldsymbol{y}^*$ should verify $g(\bar{\boldsymbol{y}}) < g(\boldsymbol{y}^*)$. Moreover, we have $\bar{\boldsymbol{y}} \preceq \boldsymbol{y}^* \leqq \mathbf{u}$, thus $\bar{\boldsymbol{y}}$ is feasible, contradicting the optimality of $\boldsymbol{y}^*$. ◀

5 Algorithm statement

The proposed algorithm iteratively explores the search region, trying to improve the current best known value ϕ of function Φ while limiting the number of search zones to be explored, and stops when the search region becomes empty.

5.1 Exploration of the search region

Since the search region is defined as a list of search zones, each zone is investigated independently. The exploration of the zone bounded by $\mathbf{u}$ is performed by solving integer programs *over a projection of* $\mathbf{u}$. All these programs are guaranteed to be feasible, and an initial feasible solution can be provided in each case (*warm start*). These two properties usually lead to faster solution times.

First, a lower bound over the value of Φ is computed by solving the program:

$$(\Pi(\ell, \mathbf{u})) = \{\min \ \Phi(\boldsymbol{x}) : \boldsymbol{x} \in \mathcal{X}, \boldsymbol{f}_{-\ell}(\boldsymbol{x}) \prec \mathbf{u}_{-\ell}\}$$

Note that, by Definition 1 and Theorem 1, if $\mathbf{u}_\ell$ is bounded then any defining point in $\mathcal{D}_\ell(\mathbf{u})$ is feasible for problem $(\Pi(\ell, \mathbf{u}))$, which allows us to optimize this problem using a warm start. Even if the resulting optimal solution $\hat{\boldsymbol{x}}$ is not guaranteed to be efficient, it provides a lower bound on Φ over the zone delimited by $\mathbf{u}$, but also over some similar search zones as stated by the following result.

▶ **Proposition 3.** *Let* $\mathbf{u}' \in \mathbb{R}^p$ *be a local upper bound such that* $\mathbf{u}'_{-\ell} \leqq \mathbf{u}_{-\ell}$ *for some* $\ell \in \{1, \ldots, p\}$. *If* $(\Pi(\ell, \mathbf{u}))$ *admits an optimal solution* $\hat{\boldsymbol{x}}$, *then* $\Phi(\hat{\boldsymbol{x}})$ *is a lower bound for any feasible point in the zone delimited by* $\mathbf{u}'$.

Proof. Since $\mathbf{u}'_{-\ell} \leqq \mathbf{u}_{-\ell}$, every $\boldsymbol{x} \in \mathcal{X}$ whose image by $\boldsymbol{f}$ is in the zone delimited by $\mathbf{u}'$ is feasible for $\Pi(\ell, \mathbf{u})$. ◀

Note that Proposition 3 applies for $\mathbf{u}$ in particular. Therefore, when the optimal value of problem $(\Pi(\ell, \mathbf{u}))$ does not improve ϕ, all search zones delimited by local upper bounds triggering Proposition 3 can be discarded. Otherwise, we proceed to the next step aiming at identifying a candidate while possibly discarding other search zones.

For this purpose, we look for a solution that minimizes $\boldsymbol{f}_\ell$ over the same projection, while improving ϕ. This is performed by solving:

$$(P(\ell, \mathbf{u})) = \{\min \ \boldsymbol{f}_\ell(\boldsymbol{x}) : \boldsymbol{x} \in \mathcal{X}, \boldsymbol{f}_{-\ell}(\boldsymbol{x}) \prec \mathbf{u}_{-\ell}, \ \Phi(\boldsymbol{x}) < \phi\}$$

Observe that, at this stage, the optimal solution returned by problem $(\Pi(\ell, \mathbf{u}))$ can be used as a warm start for problem $(P(\ell, \mathbf{u}))$. This program does not necessarily return an

efficient solution due to the additional constraint over the value of Φ. It provides however *a lower bound on objective $\boldsymbol{f}_\ell$ over all efficient solutions that improve the estimation over the projection.* The following result exploits this property in order to discard some search zones.

▶ **Proposition 4.** *Let $\mathbf{u}' \in \mathbb{R}^p$ be a local upper bound such that $\mathbf{u}'_{-\ell} \leqq \mathbf{u}_{-\ell}$ for some $\ell \in \{1, \dots, p\}$. If $(P(\ell, \mathbf{u}))$ yields a solution $\bar{\boldsymbol{x}}$ such that $\mathbf{u}'_\ell \leq \boldsymbol{f}_\ell(\bar{\boldsymbol{x}})$, then no point in the zone delimited by $\mathbf{u}'$ improves ϕ which can thus be discarded.*

Proof. Any solution $\boldsymbol{x}$ that is feasible for $P(\ell, \mathbf{u}')$ is also feasible for $P(\ell, \mathbf{u})$. Thus, we have $\mathbf{u}'_\ell \leq \boldsymbol{f}_\ell(\bar{\boldsymbol{x}}) \leq \boldsymbol{f}_\ell(\boldsymbol{x})$, meaning that $\boldsymbol{f}(\boldsymbol{x})$ cannot belong to the zone bounded by $\mathbf{u}'$. Due to the constraint $\Phi(\boldsymbol{x}) < \phi$ in program $(P(\ell, \mathbf{u}))$, the image by $\boldsymbol{f}$ of any improving solution cannot belong to this zone. ◀

As before, Proposition 4 applies for $\mathbf{u}$ in particular.

While solution $\bar{\boldsymbol{x}}$ of $(P(\ell, \mathbf{u}))$ improves ϕ, its efficiency is not guaranteed. Therefore, we look for an efficient solution dominating $\bar{\boldsymbol{x}}$, by solving the following program that optimizes a strongly monotone function (guaranteeing efficiency by Theorem 2) and discriminates among the resulting optimal solutions by optimizing Φ.

$$(\text{OptEff}) = \{\text{lexmin } \left\{ \sum_{i=1}^{p} \boldsymbol{f}_i(\boldsymbol{x}), \Phi(\boldsymbol{x}) \right\} : \boldsymbol{x} \in \mathcal{X}, \boldsymbol{f}(\boldsymbol{x}) \leqq \boldsymbol{f}(\bar{\boldsymbol{x}})\}$$

Note that $\bar{\boldsymbol{x}}$ is feasible for this problem and can thus be used as a warm start. The solution $\boldsymbol{x}^*$ of problem (OptEff) while being efficient is no longer guaranteed to improve ϕ.

It is important to observe that the search region is reduced at each iteration. Indeed, at least $\mathbf{u}$ is discarded by application of Proposition 3 or 4, or a new non-dominated point $\boldsymbol{y}^* = \boldsymbol{f}(\boldsymbol{x}^*)$ is found in the zone bounded by $\mathbf{u}$ and the region it dominates is thus removed (by splitting the zones $\boldsymbol{y}^*$ belongs to). The convergence of the resulting algorithm (see Algorithm 1) is therefore guaranteed under standard conditions ensuring that $\mathcal{X}_E$ is finite, trivially satisfied in particular for MOCO problems.

5.2 Updating the search region

Each time a non-dominated point $\boldsymbol{y}^* = \boldsymbol{f}(\boldsymbol{x}^*)$ is found, the search region must be updated by removing the part dominated by $\boldsymbol{y}^*$. The basic operation is described in [8] and is performed in two steps. First, the zones $\boldsymbol{y}^*$ belongs to must be split by replacing the corresponding local upper bounds $\mathbf{u}$ by their p children $\mathbf{u}^{\boldsymbol{i}}$, $i \in \{1, \dots, p\}$. Second, the search zones that are redundant, *i.e.* included in others, must be discarded. Moreover, by application of Propositions 3 and 4, additional zones can be ignored.

We associate to each local upper bound $\mathbf{u}$ a lower bound on Φ denoted by $l_\Phi(\mathbf{u})$. This lower bound is iteratively updated each time Proposition 3 can be applied, *i.e.* each time $\Pi(\ell, \mathbf{u}')$ is solved for a zone $\mathbf{u}'$ such that $\mathbf{u}_{-\ell} \leqq \mathbf{u}'_{-\ell}$. If the lower bound of a search zone is worse than ϕ, the zone is discarded.

It is also important to notice that Propositions 3 and 4 can be triggered at further iterations. For this reason, we store the successive results of problems $(\Pi(\ell, u))$ and $(P(\ell, u))$ in archives denoted by $\mathcal{A}_\Pi$ and $\mathcal{A}_P$, respectively. Entries in $\mathcal{A}_\Pi$ are 3-uples of the form $(\mathbf{u}, \ell, \Phi(\hat{\boldsymbol{x}}))$ and entries in $\mathcal{A}_P$ are 3-uples of the form $(\mathbf{u}, \ell, f_\ell(\bar{\boldsymbol{x}}))$. Before adding a new child $\boldsymbol{v}$, the archives are consulted. If Propositions 3 and 4 can be applied using an entry of $\mathcal{A}_\Pi$ or $\mathcal{A}_P$, the child can be discarded. The use of balanced trees to store the content of each archive allows us to perform efficient lookups since only lower bounds that are greater than ϕ (for $\mathcal{A}_\Pi$) or greater than $\boldsymbol{v}_j$ for some $j \in \{1, \dots, p\}$ (in $\mathcal{A}_P$) are relevant.

This update procedure is described in Algorithm 2.

5.3 Selecting a search zone

To increase the impact of Propositions 3 and 4, we want to prioritize *maximal projections* of local upper bounds, *i.e.* we want to select $\mathbf{u}$ and ℓ such that there is no local upper bound $\mathbf{u}'$ such that $\mathbf{u}_{-\ell} \leqq \mathbf{u}'_{-\ell}$. Moreover, we want to select ℓ such that $\mathbf{u}_\ell$ is bounded in order to exploit Theorem 1 to provide a defining point of $\mathbf{u}_\ell$ as a starting solution for $(\Pi(\ell, \mathbf{u}))$, which is always possible except at the first iteration.

For this reason, we suggest to compute the volume of the projection:

$$h(\mathbf{u}, \ell) = \prod_{\substack{i=1 \\ i \neq \ell}}^{p} \mathbf{u}_i - \boldsymbol{y}^{\boldsymbol{I}}_i$$

where $\boldsymbol{y}^{\boldsymbol{I}}$ denotes the ideal point of (MOP) , which can be obtained by optimizing independently each objective function $\boldsymbol{f}_i$ over $\mathcal{X}$.

Then, we select the projection maximizing this volume, among the current set $\mathcal{U}$ of local upper bounds:

$$(\mathbf{u}^*, \ell) = \operatorname{argmax}_{\substack{\mathbf{u} \in \mathcal{U} \\ i \in \{1,\dots,p\}}} \{h(\mathbf{u}, i)\}$$

Algorithm 1 Optimization over the efficient set.

Input : $\mathcal{X}$, $\boldsymbol{f}$, Φ
Output : ϕ, x^{opt}
/* Initialize the estimation of Φ, the set of non-dominated points, the list of upper bounds and the archives */

1. $\phi \leftarrow \infty$, $N \leftarrow \emptyset$, $\mathcal{U} \leftarrow \{(\infty, \dots, \infty)\}$, $\mathcal{A}_\Pi \leftarrow \emptyset$, $\mathcal{A}_P \leftarrow \emptyset$
2. **while** $\mathcal{U} \neq \emptyset$ **do**
3. $\quad (\mathbf{u}^*, \ell) \leftarrow \operatorname{argmax}_{\substack{\mathbf{u} \in \mathcal{U} \\ i \in \{1,\dots,p\}}} \{h(\mathbf{u}, i)\}$
4. $\quad \hat{\boldsymbol{x}} \leftarrow \operatorname{argmin} \{\Phi(\boldsymbol{x}) : \boldsymbol{x} \in \mathcal{X},\ \boldsymbol{f}(\boldsymbol{x})_{-\ell} \prec \mathbf{u}^*_{-\ell}\}$
5. $\quad \mathcal{A}_\Pi \leftarrow \mathcal{A}_\Pi \cup \{(\mathbf{u}^*, \ell, \Phi(\hat{\boldsymbol{x}}))\}$
6. $\quad$ **if** $\Phi(\hat{\boldsymbol{x}}) \geq \phi$ **then**
7. $\quad\quad \mathcal{U} \leftarrow \mathcal{U} \setminus \{\boldsymbol{v} \in \mathcal{U},\ \boldsymbol{v}_{-\ell} \leqq \mathbf{u}^*_{-\ell}, \boldsymbol{v}_\ell \geq f_\ell(\hat{\boldsymbol{x}})\}$
8. $\quad$ **else**
9. $\quad\quad \bar{\boldsymbol{x}} \leftarrow \operatorname{argmin} \{f_\ell(\boldsymbol{x}) : \boldsymbol{x} \in \mathcal{X},\ \boldsymbol{f}_{-\ell}(\boldsymbol{x}) \prec \mathbf{u}^*_{-\ell},\ \Phi(\boldsymbol{x}) < \phi\}$
10. $\quad\quad \boldsymbol{x}^* \leftarrow \operatorname{arglexmin} \{\sum_{i=1}^{p} f_i(\boldsymbol{x}), \Phi(\boldsymbol{x}) : \boldsymbol{x} \in \mathcal{X}, \boldsymbol{f}(\boldsymbol{x}) \leqq \boldsymbol{f}(\bar{\boldsymbol{x}})\}$
11. $\quad\quad N \leftarrow N \cup \{\boldsymbol{x}^*\}$
12. $\quad\quad$ **if** $\Phi(\boldsymbol{x}^*) < \phi$ // Updating the estimation
13. $\quad\quad$ **then**
14. $\quad\quad\quad \phi \leftarrow \Phi(\boldsymbol{x}^*),\ x^{\text{opt}} \leftarrow \boldsymbol{x}^*$
15. $\quad\quad \text{update}(\mathcal{U}, \phi, \mathbf{u}^*_{-\ell}, \Phi(\hat{\boldsymbol{x}}), f_\ell(\bar{\boldsymbol{x}}), \boldsymbol{x}^*, \boldsymbol{f}(\boldsymbol{x}^*))$
16. $\quad\quad \mathcal{A}_P \leftarrow \mathcal{A}_P \cup \{(\mathbf{u}^*, \ell, f_\ell(\bar{\boldsymbol{x}}))\}$

6 Computational experiments

The evaluation of our algorithm (referred to as *TV* in the following), is performed using instances of standard MOCO problems where Φ is a linear combination of the decision

Algorithm 2 Updating the search region.

Input : $\mathcal{U}$, the search region to be updated
ϕ, the value of the best known solution
$\mathbf{u}^*_{-\ell}$, the explored projection
$\Phi(\hat{\boldsymbol{x}})$, the result of $\Pi(\ell, \mathbf{u}^*)$
$\bar{\boldsymbol{y}}_\ell$, the result of $P(\ell, \mathbf{u}^*)$
$\boldsymbol{x}^*$ and $\boldsymbol{y}^*$, the efficient solution and its associated point

Output : $\mathcal{U}$, the updated search region

1 children $\leftarrow \emptyset$
/* Computing the maximal children */
2 **foreach** $\mathbf{u} \in \mathcal{U}$ **do**
3 **if** $\boldsymbol{y}^* \prec \mathbf{u}$ **then**
4 $\mathcal{U} \leftarrow \mathcal{U} \setminus \{\mathbf{u}\}$
5 **foreach** $i \in \{1, \ldots, p\}$ **do**
6 $\mathbf{u}^{\boldsymbol{i}} \leftarrow (\mathbf{u}_1, \ldots, \mathbf{u}_{i-1}, \boldsymbol{y}^*_i, \mathbf{u}_{i+1}, \ldots, \mathbf{u}_p)$
/* Computing the defining points of each bounded component */
7 $\mathcal{D}_j(\mathbf{u}^{\boldsymbol{i}}) \leftarrow \{\boldsymbol{y} \in \mathcal{D}_j(\mathbf{u}),\ \boldsymbol{y}_i < \boldsymbol{y}^*_i\},\ \forall j \in \{1, \ldots, p\}, \mathbf{u}^{\boldsymbol{i}}_j \neq \infty$
8 $\mathcal{D}_i(\mathbf{u}^{\boldsymbol{i}}) \leftarrow \{\boldsymbol{f}(\boldsymbol{x}^*)\}$
9 **if** $\mathcal{D}_j(\mathbf{u}^{\boldsymbol{i}}) \neq \emptyset,\ \forall j \in \{1, \ldots, p\},\ \mathbf{u}^{\boldsymbol{i}}_j \neq \infty$ // The child is maximal
10 *and* $\nexists(\boldsymbol{v}, j, opt) \in \mathcal{A}_\Pi : \mathbf{u}^{\boldsymbol{i}}_{-j} \leq \boldsymbol{v}_{-j}, opt \geq \phi$ // No archived problem triggers Proposition 3
11 *and* $\nexists(\boldsymbol{v}, j, opt) \in \mathcal{A}_P : \mathbf{u}^{\boldsymbol{i}}_{-j} \leq \boldsymbol{v}_{-j}, opt \geq \mathbf{u}^{\boldsymbol{i}}_j$ // No archived problem triggers Proposition 4
12 **then**
13 children $\leftarrow$ children $\cup \{\mathbf{u}^{\boldsymbol{i}}\}$
14 **else if** $\boldsymbol{y}^* \leqq \mathbf{u}$ **then**
/* $\boldsymbol{y}^*$ may be a new defining point for $\mathbf{u}$ */
15 $\mathcal{D}_j(\mathbf{u}) \leftarrow \mathcal{D}_j(\mathbf{u}) \cup \{\boldsymbol{y}^*\}\ \forall j \in \{1, \ldots, p\},\ \boldsymbol{y}^*_{-j} \prec \mathbf{u}_{-j}$
16 $\mathcal{U} \leftarrow \mathcal{U} \cup$ children
/* Application of the reduction rules */
17 **foreach** $\mathbf{u} \in \mathcal{U}$ **do**
18 **if** $\mathbf{u}_{-\ell} \leqq \mathbf{u}^*_{-\ell}$ **then**
19 **if** $\bar{\boldsymbol{y}}_\ell \geq \mathbf{u}_\ell$ **then**
/* Proposition 4 */
20 $\mathcal{U} \leftarrow \mathcal{U} \setminus \{\mathbf{u}\}$
21 **else**
/* Proposition 3 */
22 $l_\Phi(\mathbf{u}) \leftarrow \max\ \{l_\Phi(\mathbf{u}), \Phi(\hat{\boldsymbol{x}})\}$
23 **if** $l_\Phi(\mathbf{u}) \geq \phi$ **then**
24 $\mathcal{U} \leftarrow \mathcal{U} \setminus \{\mathbf{u}\}$ // The lower bound is worse than the current estimation

variables, as done for most other algorithms. Moreover we propose to compare TV with a state of the art algorithm. Natural candidates are the most recent algorithms which were themselves compared to previous ones and shown to achieve the best performances. The two most recent algorithms are [1, 11]. Comparisons in [1] report significantly better results with respect to the algorithm proposed in [5]. Comparisons in [11], for which no implementation is available, report contrasted results in particular between algorithms presented in these two papers. Moreover, algorithms in [11] deal with a special case where function Φ is a weighted combination of the objectives, with at least one negative weight. Therefore, we selected the algorithm proposed in [1] (referred to as *BCS* in the following) as a reference algorithm, using the C++ implementation provided by the authors.

Experiments have been conducted on a *Linux NixOS virtual machine* (*AMD EPYC 7702 64-Core*) running at 2000 Mhz and having 32 G of RAM. The experiments are restricted to run on a single thread, but without memory limit (less than 32G). The underlying discrete solver is *IBM Cplex* 22.10. Our code is written using the Haskell programming language and a handcrafted API for Cplex. This code is available online[2]. If an instance takes more than two hours to be solved, the tested approach is considered to have timed out.

6.1 Instances

Our approach has been validated on two sets of instances that are described in this section. In each instance, the function Φ to be minimized is randomly generated in a similar way as the objective functions.

6.1.1 MOKP

Given a set of n items, each item i having p profit values v_i^j, $j \in \{1,\dots,p\}$ and a weight w_i, the *multi-objective knapsack problem* (MOKP) consists of selecting a subset of items considering the total values on each objective, without exceeding a certain weight capacity W. This problem can be stated as:

$$(MOKP) \quad \begin{cases} \max & f_j(x) = \sum_{i=1}^n v_i^j x_i \quad \forall j \in \{1,\dots,p\} \\ \text{s.t.} & \sum_{i=1}^n w_i x_i \leq W \\ & x_i \in \{0,1\} \qquad i \in \{1,\dots,n\} \end{cases}$$

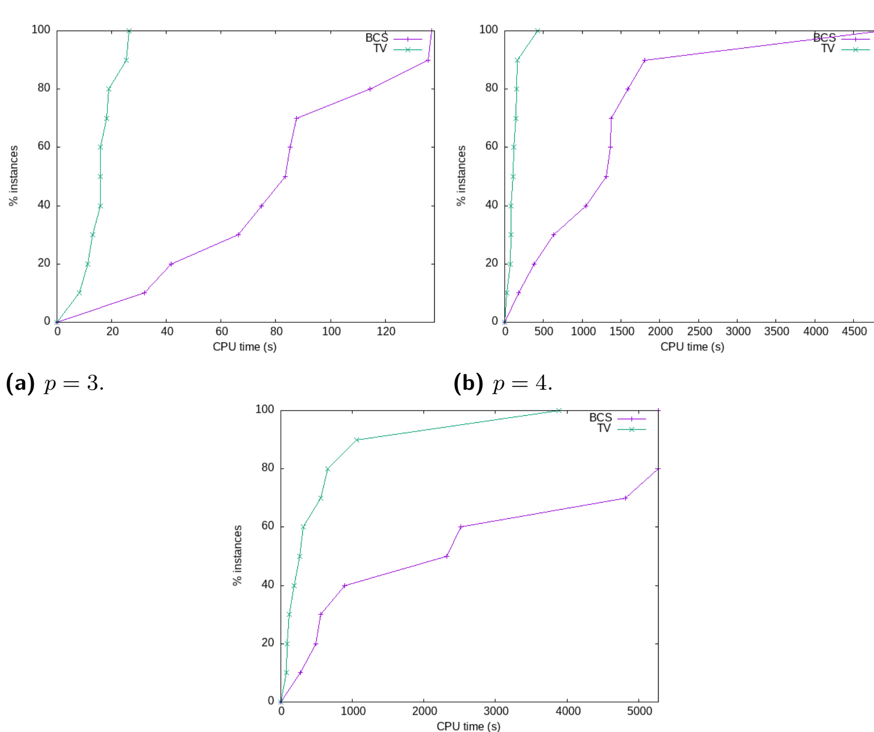
Coefficients v_i^j and w_i are uniformly sampled in $\{1,\dots,100\}$, and W is set to $\frac{\sum_{i=1}^n w_i}{2}$. 10 instances of size $n = 100$ and $p = 3, 4, 5$ have been generated.

6.1.2 MOAP

Given n tasks to be performed on n machines and p costs c_{ik}^j of assigning task i to machine k, the *multi-objective assignment problem* consists of determining an assignment considering the total cost on each objective. This problem can be stated as:

$$(MOAP) \quad \begin{cases} \min & \sum_{i=1}^n \sum_{k=1}^n c_{ik}^j x_{ik} \quad \forall j \in \{1,\dots,p\} \\ \text{s.t.:} & \sum_{i=1}^n x_{ik} = 1 \qquad k \in \{1,\dots,n\} \\ & \sum_{k=1}^n x_{ik} = 1 \qquad i \in \{1,\dots,n\} \\ & x_{ik} \in \{0,1\} \qquad i \in \{1,\dots,n\}, k \in \{1,\dots,n\} \end{cases}$$

[2] https://github.com/tambysatya/EfficientSetOptmizer



(a) $p = 3$.

(b) $p = 4$.

(c) $p = 5$.

Figure 1 Performance profiles on multi-objective knapsack problem (higher is better).

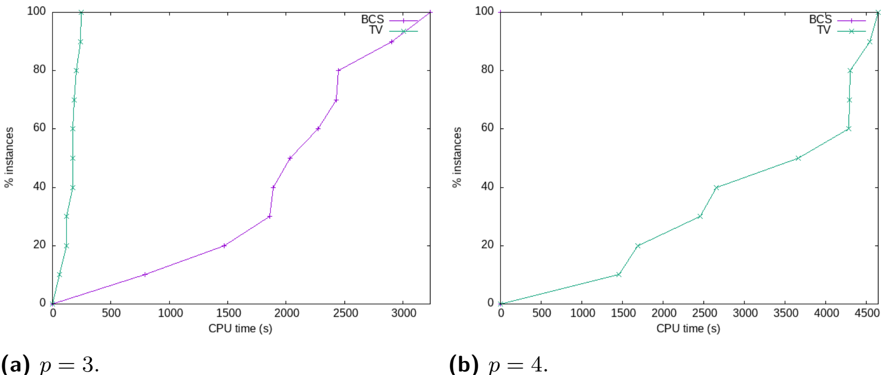
Coefficients c^j_{ik} are uniformly sampled in $\{1, \ldots, 25\}$. 10 instances of size $n = 30$ and $p = 3, 4$ have been generated.

6.2 Analysis

We first propose a comparative analysis of the CPU time required by *BCS* and *TV* on both families of instances. Performance profiles are reported in Figures 1 and 2. These plots represent the percentages of instances solved in less than t seconds, for $t \leq 7200$s. We can see that *TV* clearly outperforms *BCS* on both *MOKP* and *MOAP* instances. In particular, on the tri-objective *MOKP* and on all *MOAP* instances, *TV* solves each instance faster than the most fastly solved instance by *BCS*. In addition, *BCS* is unable to solve any instance of *MOAP* with 4 objectives and two instances of *MOKP* with 5 objectives while *TV* solves all of them in less than two hours. For the *MOKP* with 4 objectives, *TV* solves all instances in less than 500 seconds while *BCS* solves only 2 of these in this timelapse.

Second, *BCS* and *TV* are evaluated according to several measures in Tables 1 and 2. For both algorithms, the average *cpu-time*, *number of iterations* and the *number of generated non-dominated points* are presented. Additional information is reported for *TV*: the *maximum and average size of the search region*, the *percentage of zones that are discarded by reduction rules induced by Propositions 3 and 4 and using the archives.*

The average CPU time spent on each test set obviously matches the observations made from the performance profiles, validating the performance of *TV* against *BCS*, and will thus not be discussed. The number of iterations required to compute the optimum for each test set also shows that our approach converge faster. These two measures are inter-related, especially since each iteration of *BCS* involves solving problems with disjunctive constraints which are likely more difficult to be optimized and which can be infeasible while *TV* solves



(a) $p = 3$. **(b)** $p = 4$.

Figure 2 Performance profiles on multi-objective assignment problem (higher is better).

Table 1 Performance measures for *MOKP*

						mean			
p	n		CPU (s)	#It	$\|N\|$	$\|\mathcal{U}\|$max	$\|\mathcal{U}\|$avg	Reductions (%)	Archive (%)
3	100	TV	16.9	218.3	172.7	91.8	44.67	27.65	0.50
		BCS	85.8	287.0	155.5				
4	100	TV	135.8	1003.8	560.5	1237.4	696.02	14.63	0.88
		BCS	1457.1	1242.7	436.3				
5	100	TV	720.3	3123.2	1071.9	12493.2	6852.99	8.38	0.86
		BCS	-						

Table 2 Performance measures for *MOAP*.

						mean			
p	$n \times n$		CPU (s)	#It	$\|N\|$	$\|\mathcal{U}\|$max	$\|\mathcal{U}\|$avg	Reductions (%)	Archive (%)
3	30 × 30	TV	169.8	561.9	475.5	177.3	93.78	25.41	0.32
		BCS	2134.0	2042.7	1020.5				
4	30 × 30	TV	3396.0	5140.9	3267.5	10609.4	5319.58	13.88	0.33
		BCS	-						

only feasible problems that are augmented with budget constraints only. Regarding MOKP, we can observe that, while converging faster than *BCS*, *TV* generates a slightly larger number non-dominated points (for $p = 3, 4$). This is no longer true for MOAP, where *BCS* requires the generation of about four times more non-dominated points (for $p = 3$).

To perform a more detailed analysis of *TV*, several points must be discussed. First, the maximum and average size of the current search region remains "reasonable", suggesting that time is mainly spent in exploring zones and justifying our aim of reducing the number of calls to the underlying discrete solver and helping it by providing feasible problems for which initial feasible solutions are also provided (warm start). Second, we can see that reduction rules are quite efficient in particular for tri-criteria case where about a quarter of the children zones are discarded, both for MOKP and MOAP. As expected given the conditions for triggering these rules, this proportion decreases for 4 objectives (around 15%) to become less than 10% when $p = 5$. Conversely, despite being significantly smaller, the proportion of zones that are discarded using the archives is rather stable when the number of objectives grows.

7 Conclusion

While strongly relying on mechanisms developed in [14] that already proved quite successful for problem (MOP) (reduction rules to reject zones without probing them, or providing an initial solution for every integer program to be solved), this algorithm, proposed for problem (MOP_E) , takes advantage of new results, notably the computation of local lower bounds on the function to be minimized over the efficient set. Additional reduction rules are thus proposed, allowing pruning the search space and converging faster. Our experiments on the multi-objective knapsack and assignment problems show promising results since this approach seems to perform significantly better than the state of the art algorithms.

Besides technically improving the current method, further works may concern studying the optimization of specific functions. For instance, a natural extension would be to apply this method to the computation of the nadir point, whose components are the worst possible values taken by the non-dominated points. Our perspective in this respect is to make use of specific properties of this point so as to adapt our approach to this problem.

References

1 Natashia Boland, Hadi Charkhgard, and Martin Savelsbergh. A new method for optimizing a linear function over the efficient set of a multiobjective integer program. *European Journal of Operational Research*, 260(3):904–919, 2017.

2 Natashia Boland, Hadi Charkhgard, and Martin W. P. Savelsbergh. The *L*-shape search method for triobjective integer programming. *Mathematical Programming Computation*, 8(2):217–251, 2016.

3 Djamal Chaabane and Marc Pirlot. A method for optimizing over the integer efficient set. *Journal of Industrial and Management Optimization*, 6(4):811–823, 2010.

4 Kerstin Dächert, Kathrin Klamroth, Renaud Lacour, and Daniel Vanderpooten. Efficient computation of the search region in multi-objective optimization. *European Journal of Operational Research*, 260(3):841–855, 2017.

5 Jesus Jorge. An algorithm for optimizing a linear function over an integer efficient set. *European Journal of Operational Research*, 195(3):98–103, 2009.

6 Gokhan Kirlik and Serpil Sayin. A new algorithm for generating all nondominated solutions of multiobjective discrete optimization problems. *European Journal of Operational Research*, 232(3):479–488, 2014.

7 Gokhan Kirlik and Serpil Sayin. Computing the nadir point for multiobjective discrete optimization problems. *Journal of Global Optimization*, 62(1):79–99, 2015.

8 Kathrin Klamroth, Renaud Lacour, and Daniel Vanderpooten. On the representation of the search region in multi-objective optimization. *European Journal of Operational Research*, 245(3):767–778, 2015.

9 Dieter Klein and Edward L. Hannan. An algorithm for the multiple objective integer linear programming problem. *European Journal of Operational Research*, 9(4):378–385, 1982.

10 Murat Köksalan and Banu Lokman. Finding nadir points in multi-objective integer programs. *Journal of Global Optimization*, 62(1):55–77, 2015.

11 Banu Lokman. Optimizing a linear function over the nondominated set of multiobjective integer programs. *International Transactions in Operational Research*, 28(4):2248–2267, 2021.

12 Banu Lokman and Murat Köksalan. Finding all nondominated points of multi-objective integer programs. *Journal of Global Optimization*, 57(2):347–365, 2013.

13 John Sylva and Alejandro Crema. A method for finding the set of non-dominated vectors for multiple objective integer linear programs. *European Journal of Operational Research*, 158(1):46–55, 2004.

14 Satya Tamby and Daniel Vanderpooten. Enumeration of the nondominated set of multiobjective discrete optimization problems. *INFORMS Journal on Computing*, 33(1):72–85, 2021.

Solving Directed Feedback Vertex Set by Iterative Reduction to Vertex Cover

Sebastian Angrick
Hasso Plattner Institute,
Universität Potsdam, Germany

Ben Bals
Hasso Plattner Institute,
Universität Potsdam, Germany

Katrin Casel
Humboldt-Universität zu Berlin, Germany

Sarel Cohen
The Academic College of Tel Aviv-Yaffo, Israel

Tobias Friedrich
Hasso Plattner Institute,
Universität Potsdam, Germany

Niko Hastrich
Hasso Plattner Institute,
Universität Potsdam, Germany

Theresa Hradilak
Hasso Plattner Institute,
Universität Potsdam, Germany

Davis Issac
Hasso Plattner Institute,
Universität Potsdam, Germany

Otto Kißig
Hasso Plattner Institute,
Universität Potsdam, Germany

Jonas Schmidt
Hasso Plattner Institute,
Universität Potsdam, Germany

Leo Wendt
Hasso Plattner Institute,
Universität Potsdam, Germany

Abstract

In the *Directed Feedback Vertex Set* (DFVS) problem, one is given a directed graph $G = (V, E)$ and wants to find a minimum cardinality set $S \subseteq V$ such that $G - S$ is acyclic. DFVS is a fundamental problem in computer science and finds applications in areas such as deadlock detection. The problem was the subject of the 2022 PACE coding challenge. We develop a novel exact algorithm for the problem that is tailored to perform well on instances that are mostly bi-directed. For such instances, we adapt techniques from the well-researched vertex cover problem. Our core idea is an iterative reduction to vertex cover. To this end, we also develop a new reduction rule that reduces the number of not bi-directed edges. With the resulting algorithm, we were able to win third place in the exact track of the PACE challenge. We perform computational experiments and compare the running time to other exact algorithms, in particular to the winning algorithm in PACE. Our experiments show that we outpace the other algorithms on instances that have a low density of uni-directed edges.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases directed feedback vertex set, vertex cover, reduction rules

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.10

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.6645235`
Software (Source Code): `https://github.com/BenBals/mount-doom/tree/exact`
archived at `swh:1:dir:a8ce8a824241821bdff98f5380594c74d2d6c327`

1 Introduction

In the Directed Feedback Vertex Set (DFVS) problem, we are given a directed graph $G = (V, E)$ and the objective is to find a minimum cardinality set $S \subseteq V$ such that $G - S$ is acyclic. DFVS is a fundamental computational problem that appeared in Karp's seminal paper [19]. The problem is equivalent to the Feedback Arc Set (FAS) problem where we want to delete edges instead of vertices i.e., there are reductions in both directions that preserve the value of the solution and blow up the graph size only polynomially. Both

© Sebastian Angrick, Ben Bals, Katrin Casel, Sarel Cohen, Tobias Friedrich, Niko Hastrich, Theresa Hradilak, Davis Issac, Otto Kißig, Jonas Schmidt, and Leo Wendt;
licensed under Creative Commons License CC-BY 4.0

problems have applications in areas such as deadlock detection [13], compiler optimization [34], program verification [31], VLSI chip design [5], Computer Aided Design (CAD) [23], and chemical engineering [2].

From a theoretical point of view, DFVS is NP-hard, thus polynomial exact algorithms are very unlikely. The currently best known theoretical runtime known in terms of input size n for an exact algorithm is the $\mathcal{O}(1.9977^n)$ algorithm by Razgon [28], only barely beating the trivial $\mathcal{O}(2^n)$ time brute-force algorithm. From the perspective of parameterized algorithms (see [8] for an introduction to parameterized complexity), the current best fixed parameter tractable algorithm for DFVS is a $4^k k! n^{\mathcal{O}(1)}$ time algorithm by Chen et al. [7], where the parameter k is the size of the directed feedback vertex set. The best known polynomial time approximation algorithm known for DFVS is a $\mathcal{O}(\log k \log \log k)$ approximation by Even et al. [12].

From a practical point of view, there are only few recent exact algorithms that can be used for large general instances. Historically, Smith and Walford [32] gave the first exact algorithm for DFVS in 1975 but their algorithm can only handle small graphs [21]. There are also a few *branch and bound* based exact algorithms. Chakradhar et al. [5] in 1995 gave a branch and bound exact algorithm using an ILP relaxation for finding DFVS of flip-flop dependency graphs arising from sequential circuits. Orenstein et al. [25], also in 1995, used branch and bound to find optimal DFVS in digital circuit graphs and tested them on ISCAS89 benchmarks. Lin and Jou [23] in 1999 gave an improved *branch and bound* exact algorithm together with reduction rules, and experimentally evaluated them on the ISCAS89 benchmarks coming from CAD applications.

More recent exact algorithms use different techniques. Fleischer et al. [13] experimentally evaluate Chen's algorithm and some reduction rules. Based on their experiments, the algorithm already becomes impractical when the solution size k is as small as 10. Baharev et al. [2] gave an integer-program based exact algorithm for Feedback Arc Set that uses a lazy enumeration of cycles, and can be also applied to DFVS. To apply an FAS algorithm to DFVS, one can split each vertex into an in-vertex and out-vertex with a uni-directed edge between them.

Some heuristic algorithms have been also developed for DFVS e.g, based on greedy randomized adaptive search [26] (1998), markov chains [22] (2008), and local search [15] (2013). We do not describe them in further detail as exact algorithms are the focus of this work.

Perhaps also due to this lack of fast exact algorithms for DFVS, it was selected as the problem for the PACE 2022 coding challenge [30].

A much more extensively studied problem, that was also subject of the PACE challenge in 2019 [10], is Vertex Cover. It is closely related to other important problems such as Maximum Clique, Maximum Independent Set, Odd Cycle Tansversal etc. Often, a formulation in terms of vertex cover have been useful in solving these problems [24].

From a theoretical point of view, Vertex Cover is "easier" than DFVS, since there is an easy reduction from Vertex Cover to DFVS that just replaces undirected edges with bi-directional arcs. This is also reflected in the much better theoretical exact algorithms. The current best exact algorithm measured with respect to the number of vertices n runs in $\mathcal{O}(1.1996^n)$ [35]. Also, just recently, the parameterized $1.2738^k n^{\mathcal{O}(1)}$ algorithm by Chen, Kanj and Xia [6] was improved to $1.25298^k n^{\mathcal{O}(1)}$ by Harris and Narayanaswamy [17].

Also, and perhaps even more so, there are much more powerful practical solvers for Vertex Cover than for DFVS. Vertex Cover has a straight forward ILP formulation that can be then fed into commercial ILP solvers such as Gurobi and CPLEX. Akiba and

Iwata [1] gave a branch and reduce algorithm and showed it to be competitive with the CPLEX solver. Faster solvers were developed as a result of the PACE 2019 challenge [10]. The winning solver is the WeGotYouCovered solver by Hespe et al. [18] which uses branch and reduce and branch and bound. Later, Plachetta and van der Grinten [27] developed a branch and reduce algorithm using a SAT solver that was competitive with the PACE winner. Stallmann, Ho and Goodrich [33] enhanced the solver by Akiba and Iwata [1] using targeted reductions depending on the instance profile. Also, vertex cover has extremely fast heuristic solvers, e.g. [4]. Thus, our approach to DFVS is to take advantage of this wealth of results for Vertex Cover.

In a recent ALENEX 2023 paper [20], Kiesel and Schidler describe their DAGer algorithm for DFVS, which won the first place in the PACE 2022 challenge [29]. In a nuthsell, their algorithm first applies an extensive set of preprocessing rules to reduce the size of the instance. The reduced instance is then solved exactly using a modified SAT-solver. As initialization, a set of clauses corresponding to cycles that need to be hit, is given. This set might not be exhaustive. To ensure feasibility, the SAT-solver is modified to maintain a topological ordering on the vertices and dynamically detect cycles and adding the corresponding clauses to the problem. They refer to this approach as Cycle Propagation, which builds on the idea that already a limited number of the constraints in a propositional encoding is usually sufficient for finding an optimal solution, thus their algorithm starts with a small number of constraints and cycle propagation adds additional constraints when necessary. Our approach deviates from DAGer, as we describe in the following section.

Our Contribution

We develop a novel exact algorithm Mount-Doom for the DFVS problem. Our approach is different from all the previous exact as well as heuristic approaches. We develop a method to iteratively reduce DFVS to the Vertex Cover problem and exploit fast existing algorithms for this problem. In order to strategically reduce to Vertex Cover, we also develop a new reduction rule called SHORTONE that reduces the number of uni-directed edges.

With these strategies, Mount-Doom was able to secure third place in the exact track of PACE 2022 challenge.

We perform experiments to compare our algorithm to the PACE winning solver DAGer [20, 29] and the state-of-the-art ILP based solver Sdopt [2] for DFAS. We use the complete set of 200 instances from the PACE competition as well as our own randomly generated instances, and observe that in a fair share of the instances our running times are significantly better. As characteristic of instances in which our algorithm performs better, we consider the density of uni-directed edges. This measure can be seen as a distance to the DFVS instance being a Vertex Cover instance. We show experimentally that when the density of uni-directed edges is small then we are considerably faster than other algorithms in the PACE instances.

We also perform another set of experiments to demonstrate the utility of our new reduction rule SHORTONE. Our experiments reveal that in many of the PACE instances there is significantly more reduction in the instance size by using this new rule in addition to the existing reduction rules. On our less structured set of generated graphs (Erdős-Rényi graphs), we are more often outperformed by the competitors. This indicates that our algorithm heavily takes advantage of structured input.

2 Preliminaries

We use $G = (V, E)$ to denote directed and undirected graphs. For any $S \subseteq V$ we write $G - S$ for the graph obtained from G by deleting all vertices in S together with their adjacent edges. With these notions, the Directed Feedback Vertex Set problem (DFVS for short) is the task to find a minimum cardinality subset $S \subseteq V$ such that $G - S$ is acyclic.

We call an edge $vw \in E$ *bi-directed* if $wv \in E$. Let $PIE \subseteq E$ be the set of all bi-directed edges and $DIR = E \setminus PIE$. We define $G[PIE]$ to be the *undirected* graph obtained from G by deleting all edges in DIR, and replacing any pair of bi-directed edges by an undirected edge. Further, we define the graph $G[DIR] = (V, DIR)$, the subgraph of G containing only the not bi-directed edges. We call theses edges *uni-directed*. Therefore, we define the uni-directed edge density of a graph as the number of uni-directed edges divided by the number of possible edges $\binom{|V|}{2}$. We use $uv \in E$ to denote an uni-directed edge from u to v, and $\{u, v\}$ for a bi-directed edge.

For any $v \in V$, $N^+(v)$ denotes the set of outgoing neighbors of v, i.e. $N^+(v) = \{u \mid vu \in E\}$. Similarly, $N^-(v)$ denotes the set of incoming neighbors. We define the set of bi-directed neighbors $N(v)$ of v as the set of its neighbors in $G[PIE]$, formally $N(v) = \{u \mid uv \in E \wedge vu \in E\}$. Additionally, we call $D \subseteq V$ a *diclique*, if $D \setminus \{u\} \subseteq N(u)$ for each $u \in D$.

For $v \in V$, we define the graph obtained from G by *shortcutting* v as the graph $G' = (V', E')$ with vertex set $V' = V \setminus \{v\}$ and edge set $E' = (E \cap (V' \times V')) \cup (N^-(v) \times N^+(v))$. Note that in the context of the DFVS problem, shortcutting v corresponds to the assumption that v is not part of the solution.

3 Mount Doom Solver

As already mentioned, our overall idea is to profit from the plethora of results obtained for the Vertex Cover problem. We do this both by adapting known reduction rules for and also by a direct reduction to Vertex Cover.

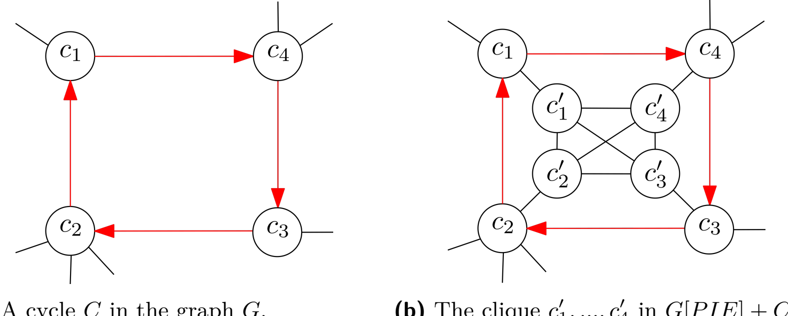
3.1 Reduction to Vertex Cover

Our reduction to Vertex Cover relies on the simple but surprisingly powerful observation, that any directed feedback vertex set has to in particular cover all bi-directed edges. In the special case of a graph that only contains bi-directed edges, DFVS is equivalent to Vertex Cover. More generally, one can observe the following relation to the underlying undirected graph; recall that we defined $G[PIE]$ to be the graph obtained from G by replacing bi-directed edges by undirected edges, and then deleting the remaining directed edges.

▶ **Observation 1.** *If S is a minimum vertex cover for $G[PIE]$ and $G - S$ is acyclic, then S is a minimum feedback vertex set for G.*

This observation implies that if we happen to find a feedback vertex set by solving the Vertex Cover problem on $G[PIE]$, then we have solved DFVS on G. On the other hand, if such a minimum vertex cover S for $G[PIE]$ is not a feedback vertex set for G, then there exists a cycle in $G[DIR]$ that is not covered by S. For our reduction, we add a *clique gadget* to such cycles.

Formally, let $G = (V, E)$ be an undirected graph and let $C \subseteq V$. Our goal is to create a graph G' such that that any minimum vertex cover for G' is a vertex cover for G and also contains a vertex in C. Let C be a set of r vertices $c_1, \ldots, c_r$. For our clique gadget, we



(a) A cycle C in the graph G. **(b)** The clique $c_1', \ldots, c_4'$ in $G[PIE] + C$.

Figure 1 An example reduction to vertex cover. Red denotes uni-directed and black denotes bi-directed edges.

add r new vertices $c_1', \ldots, c_r'$ to G, turn them into a clique, and add the edges $\{c_i, c_i'\}$ for all $1 \leq i \leq r$. We denote this operation by $G + C$. See Figure 1 for an example application. We first prove an exchange argument for vertex covers for $G + C$.

▶ **Lemma 2.** *For any cycle C in $G[DIR]$ and vertex cover S for $G[PIE] + C$, a vertex cover S' for $G[PIE] + C$ with $|S'| \leq |S|$ and such that $C \cap S' \neq \emptyset$ can be created in linear time.*

Proof. Assume that S contains no vertices of $C = \{c_1, \ldots, c_r\}$, as otherwise we are already done. In order to cover the edges $\{c_i, c_i'\}$ for $1 \leq i \leq r$, S then has to contain all vertices c_i', $1 \leq i \leq r$ added by the clique gadget. To create S' from S, we replace c_1' by c_1. This choice of S' obviously satisfies $|S'| \leq |S|$. Further, S' is also a vertex cover for $G[PIE] + C$, since $N(c_1') = \{c_1, c_2', \ldots, c_r'\} \subseteq S'$ and thus all edges that were covered by S through c_1' remain covered by S'. ◀

With this exchange argument, we can assume that adding the clique gadget results in a vertex cover for $G[PIE] + C$ that contains at least one vertex from C. The overall idea of our reduction to Vertex Cover is to iteratively add clique gadgets for so far not covered cycles, until we find a feedback vertex set for the original graph.

Consider any set of cycles $\mathcal{C} = \{C_1, C_2, \ldots, C_q\}$ in $G[DIR]$. Denote the graph resulting from iteratively applying the clique gadget for each cycle in $\mathcal{C}$ by $G[PIE] + \mathcal{C}$. Note that with this gadgetry we add multiple *disjoint* sets C_i' of new vertices. Formally, for $C_i = \{c_1^i, \ldots, c_{r_i}^i\}$, we add new vertices $C_i' = \{d_1^i, \ldots, d_{r_i}^i\}$, add edges to turn C_i' into a clique, and add edges $\{c_j^i, d_j^i\}$ for all $1 \leq i \leq r_i$, for each $1 \leq i \leq q$. For this iterative application of the clique gadget, we observe the following.

▶ **Lemma 3.** *For any set of $\mathcal{C} = \{C_1, C_2, \ldots, C_q\}$ in $G[DIR]$ and minimum vertex cover S for $G[PIE] + \mathcal{C}$, if $G - S$ is acyclic, then $S \cap V$ is a minimum feedback vertex set for G.*

Proof. Let S^* be a minimum feedback vertex set for G. We claim that $|S^*| \geq |S \cap V|$, which proves the lemma. Since S^* is a feedback vertex set for G, it follows that S^* is a vertex cover for $G[PIE]$, and contains at least one vertex from each cycle in $\mathcal{C}$. (Note that any edge in $G[PIE]$ and any cycle in $G[DIR]$ corresponds to a cycle in G that needs to be covered.) Assume w.l.o.g. that S^* contains vertex c_1^i for each C_i.

Let $S^*_{\mathcal{C}}$ be the set created from S^* by adding $d_2^i, \ldots, d_{r_i}^i$ for each i, $1 \leq i \leq q$. Observe that $S^*_{\mathcal{C}}$ then is a vertex cover for $G[PIE] + \mathcal{C}$. The vertices added from the C_i' cover the clique-edges among the C_i', and all edges $\{c_j^i, d_j^i\}$ with $j \geq 2$. Since S^* contains vertex c_1^i, also the edge $\{c_1^i, d_1^i\}$ is covered for each i.

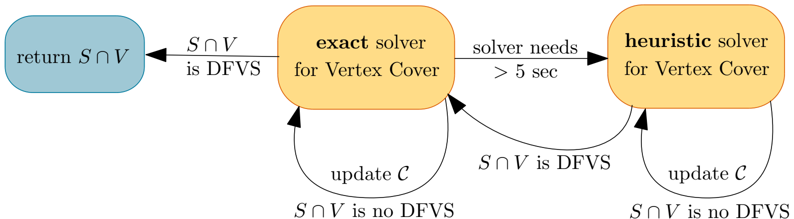


Figure 2 Overview of the reduction to Vertex Cover.

Since S is a minimal vertex cover for $G[PIE] + \mathcal{C}$, it follows that $|S^*_{\mathcal{C}}| \geq |S|$. At last, note that $S \cap V$ contains at least $r_i - 1$ out of the r_i vertices in C'_i for each i, since these build a clique in $G[PIE] + C$ that needs to be covered. Also, we added exactly $r_i - 1$ out of the r_i vertices in C'_i for each i to S^* to create $S^*_{\mathcal{C}}$.

Thus $|S \cap V| \leq |S| - \sum_{i=1}^{q}(r_i - 1) \leq |S^*_{\mathcal{C}}| - \sum_{i=1}^{q}(r_i - 1) = |S^*|$. ◀

Building from this idea, we iteratively choose new cycles and add their corresponding clique gadgets, growing a set $\mathcal{C}$. By using the exchange argument from Lemma 2 on the vertex cover for $G[PIE] + \mathcal{C}$, we are certain that this process converges to finding a directed feedback vertex set for the original graph, at the latest when $\mathcal{C}$ contains all cycles in G. We start with $\mathcal{C} = \emptyset$, and iteratively do the following until we find a solution. Compute a minimum vertex cover S for $G[PIE] + \mathcal{C}$ and use the exchange argument from Lemma 2 to create a set that contains at least one vertex for each cycle in $\mathcal{C}$. If $S \cap V$ is not a feedback vertex set for G, we compute a set of vertex disjoint cycles in $G[DIR] - S$ (simply with a depth first search) and add them to $\mathcal{C}$.

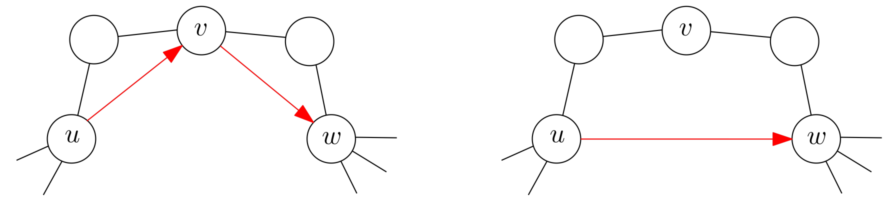
Since this iterative built up of $\mathcal{C}$ can result in multiple calls to a Vertex Cover solver, we do the following. If the exact Vertex Cover solver takes more than five seconds, we switch to a heuristic solver to find a set $\mathcal{C}$ that gives a feedback vertex set. (See Figure 2 for an overview of how we switch between solvers.) Then we run the exact Vertex Cover solver only once on $G[PIE] + \mathcal{C}$ for this set $\mathcal{C}$ to find an optimal solution. In theory, it could happen that the set $\mathcal{C}$ by the heuristic solver gives a feedback vertex set does not work for the exact solver and we would have to iterate further, adding more cycles to $\mathcal{C}$. To our knowledge, this did not happen in our experiments.

3.2 Reduction Rules

Before reducing to Vertex Cover, we apply a number of reduction rules to shrink the input. We adapt some reduction rules that were introduced for the Vertex Cover problem, and also introduce a new rule that is tailored specifically for instances with many bi-directed and few not bi-directed edges.

We can again use that any feedback vertex set in particular is a vertex cover for $G[PIE]$ and inherit the following reduction rules from [14] for Vertex Cover.

VC-DOME. Let u be isolated in $G[DIR]$ such that $N(u) \setminus \{v\} \subseteq N(v)$ for some $v \in N(u)$ (u is dominated by v in $G[PIE]$). Add v to the solution and delete it from G.



(a) Before application of SHORTONE. **(b)** After application of SHORTONE.

Figure 3 An example application of the SHORTONE reduction rule. Red denotes uni-directed and black denotes bi-directed edges.

Degree 2 Fold. Let $v, u \in V$ be two isolated vertices in $G[DIR]$ (i.e. they are adjacent only to bi-directed edges in G). Also, let v have degree two in $G[PIE]$ with $N(v) = \{u, w\}$. Add a new vertex t to G and connect t in such a way, that $N^+(t) = N^+(w) \cup N(u)$ and $N^-(t) = N^-(w) \cup N(u)$ and remove u, v and w.

To unfold this reduction after a solution S is computed for the reduced graph do the following. If $t \in S$, the solution to the original instance is $(S \setminus \{t\}) \cup \{u, w\}$, and otherwise, the solution to the original instance is $S \cup \{v\}$.

Funnel Fold. Let v be isolated in $G[DIR]$ and $w \in N(v)$ such that $N(v) \setminus w$ is a diclique. Add $C = N(v) \cap N(w)$ to the solution and delete these vertices. Then add edges, such that each $x \in N(v) \setminus C$ is a bi-directed neighbor of every $y \in N(w) \setminus C$. Finally delete v and w.

To unfold this reduction after a solution S is computed for the reduced graph do the following. If $N(v) \setminus \{w\} \subseteq S$, the solution to the original instance is $S \cup \{w\}$, otherwise $S \cup \{v\}$.

Particularly for instances with low degree in $G[DIR]$, we design the following reduction rule to decrease the number of edges that are not bi-directed.

SHORTONE. Let v be a node with exactly one incoming edge uv and one outgoing edge vw in $G[DIR]$ such that $N(v) \subseteq N(u) \cup N(w)$ (any bi-directed neighbor of v is also a bi-directed neighbor of u or w). Remove the edges uv and vw and add uw.

▶ **Lemma 4.** *Reduction rule SHORTONE is correct.*

Proof. Let G be the original graph, and G' be the graph obtained from G by applying SHORTONE. First consider any directed feedback vertex set S for G. If $v \notin S$, then S is also a solution for G'. If $v \in S$, assume S is not a solution for G'. Then $u, w \notin S$ since any cycle that is present in G' and not in G was introduced by SHORTONE and hence must use uw. Since all bi-directed neighbors of v in G are also bi-directed neighbors of u or w in G, those must all be included in S. Thus we can simply replace v by u (or w) and obtain a solution for G' of the same size as S.

Conversely, consider any solution S' for G' and assume that it is not a solution for G. Thus $G - S'$ still contains a cycle C. Since S' is a solution for G', C has to contain uv or vw since these are the only arcs deleted to create G'. Further, C cannot contain both uv and vw, since G' contains the added uw that could be used to turn C into a cycle in G'. Thus, C has to contain edge of the form vx or xv for some $x \notin \{u, w\}$. However, all possible choices of x are bi-directed neighbors of v in G', and are thus contained in S'. ◀

See Figure 3 for an example application of the SHORTONE reduction rule.

Aside from these reduction rules imported or inspired by techniques developed for Vertex Cover, we use and slightly alter some of those rules for DFVS.

These following two rules can be found in [22], where we do a slight novel alteration of the second one.

PIE. If uv is an edge between different strongly connected components in $G[DIR]$, then delete uv.

Improved CORE. A vertex a is a *core* of a diclique, if the graph induced by a and its neighbors is a diclique. Traditionally, one now deletes $N(a)$ from G since if S' is optimal for $G - N(v)$, $S' \cup N(v)$ is optimal for G. We proceed differently and shortcut the node a if $N^+(a)$ or $N^-(a)$ are dicliques. While this extension is easy to prove, it is, to the best of our knowledge, novel.

As first obvious reduction rules that are always applied, all nodes with self loops are collected into the solution and removed from the graph, and isolated nodes are removed as well. Further we use the classical domination rule, formally.

DOME. An edge $ab \in DIR$ is called dominated if all outgoing neighbors of b are also outgoing neighbors of a or if all incoming neighbors of a are also incoming neighbors of b. It is well known (see e.g. [9]) that such a dominated edge can safely be deleted.

3.3 Implementation Details

We fist only use the reduction rules Improved CORE and PIE, and deletion of self loops and isolated nodes. If we do not completely solve the instance within five seconds, using only these reductions, we proceed with all reduction rules listed above.

In both cases, if the instance is not solved after exhaustive application of all reduction rules, we proceed with our clique gadget to reduce to Vertex Cover. To solve the Vertex Cover instances $G[PIE] + \mathcal{C}$, we initially reduce them using the kernelization procedure, implemented by the winning solver of the 2019 PACE challenge [18]. For solving these reduced instances heuristically, we use the local-search solver NuMVC by Cai et al. [4]. When the heuristic solver has found a successful set of cycles $\mathcal{C}$, we compute an exact minimal vertex cover for $G[PIE] + \mathcal{C}$. For this exact computation, we use the solver SAT-and-Reduce by Plachetta and van der Grinten [27], which we augment by implementing better upper bounds using the aforementioned local-search solver (SAT-and-Reduce internally uses the SAT solver CaDiCaL by Biere et al. [3].)

4 Experiments

In this section we give the details of the computational experiments conducted to evaluate our solver Mount-Doom.

4.1 Data

We evaluate both the effect of our new reduction rule SHORTONE and the overall solver on two data sets. Firstly, we examine the performance on the public and private instances from the PACE 2022 challenge. Those are designed to provide a wide range of graph types with practical relevance [30].

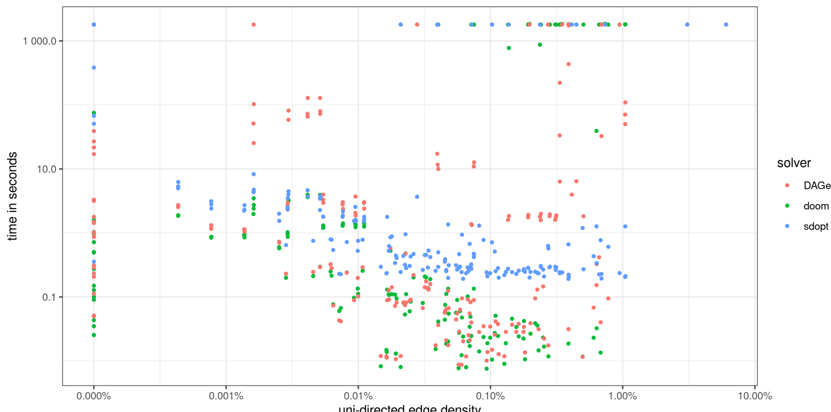


Figure 4 Running time on PACE graphs by density of uni-directed edges.

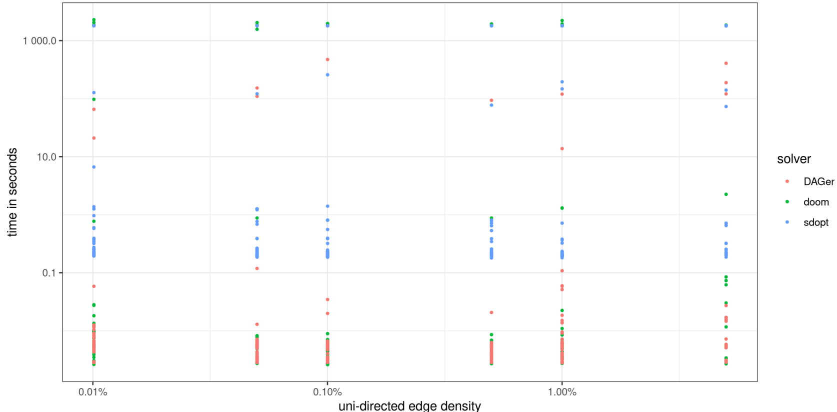


Figure 5 Running time on Erdős-Rényi graphs by density of uni-directed edges.

Secondly, we evaluate on randomly generated graphs of varying sizes and densities. In the usual Erdős-Rényi model the number of vertices is fixed and each possible edge is present uniformly and independently at random with some constant probability [11]. Since we are interested in the effect of the uni-directed edge density on the solver performance, we modify this model as follows: For $n \in \mathbb{N}$ and $p, q \in \mathbb{R}^+$ s.t. $p + q \leq 1$ generate a graph $G(n, p, q)$ with n vertices letting each possible edge be a bi-directed edge with probability p, a single edge with probability q and not present with probability $1 - p - q$. In the case of single edges, pick one of the two possible directions with equal probability. We then examine different parameters, with n between 10 and 500, p and q between 0.0001 and 0.05. Overall, we generated 253 such graphs.

These two data sources help us gain a good understanding of two different aspects: The PACE instances provide large scale real-world relevant instance which contain up to a million edges while our random instances provide a more systematic view into the effect of the uni-directed edge density.

4.2 Competing Algorithms

We compare our algorithm with the winner of the PACE 2022 implementation challenge DAGer [20, 29] and a state-of-the-art-solver for the Directed Feedback Arc Set problem (DFAS) Sdopt [2].

The solver DAGer first applies an extensive set of preprocessing rules to reduce the size of the instance. The reduced instance is then solved exactly using a modified SAT-solver. As initialization, a set of clauses corresponding to cycles that need to be hit, is given. This set might not be exhaustive. To ensure feasibility, the SAT-solver is modified to maintain a topological ordering on the vertices and dynamically detect cycles and adding the corresponding clauses to the problem.

To use the solver Sdopt, which is designed to solve the DFAS problem, we first use the canonical reduction from the DFVS problem to the DFAS problem. Initially the solver enumerates a set of cycles and solves the integer programming formulation for hitting all of theses cycles. If the obtained solution is a directed feedback arc set, this solution is returned, otherwise the current solution is augmented to a feasible DFAS and the found optimum is updated. Additionally, it searches for cycles that are not hit and adds some of them to the integer program relaxation. This process is repeated until a solution is found.

4.3 Machine Specifications

All experiments were conducted on an AMD EPYC 7742 at 2.25GHz CPU running Fedora 34. Both DAGer and Mount-Doom were compiled with GCC 11.3.1. Sdopt was run with Python 3.9 against Gurobipy 9.5.2 [16]. All experiments were conducted on a single thread.

4.4 Experiment Description

To examine the efficacy of our solver as a whole and the effect of our proposed reduction rule SHORTONE, we conducted two experiments.

For our first experiment, we take the two data sets described above and run the three described solvers (DAGer, Sdopt, Mount-Doom) on them.

We hypothesize that on instances with a large number of bi-directed edges, our solver outperforms both competing solvers. To support this conclusion, the randomly generated graphs contain instances with a wide range of bi-directed and uni-directed edges and overall densities. The PACE instances also vary widely in these parameters.

In the second experiment, we want to examine the effect of our novel reduction rule SHORTONE. We compare the reduction achieved by our whole set of reduction rules to this set without SHORTONE on both of the data sets.

4.5 Results and Discussion

Next, we present the results of the conducted experiments and summarize our findings.

Figure 4 shows the running times of the three algorithms Mount-Doom, DAGer, and Sdopt on the PACE instances and Figure 5 shows the running times on the Erdős-Rényi graphs. It is apparent from the plots that Mount-Doom and DAGer outpace the Sdopt solver. Hence we make further one-to-one comparisons between DAGer and Mount-Doom in Table 1. Although DAGer clearly wins on the PACE instances in the total number of instances solved before the timeout, our Mount-Doom solver wins in the number of instances solved faster. On Erdős-Rényi graphs, DAGer also wins in the number of fast solved instances. This indicates that our solver is more suited for structured and less random instances.

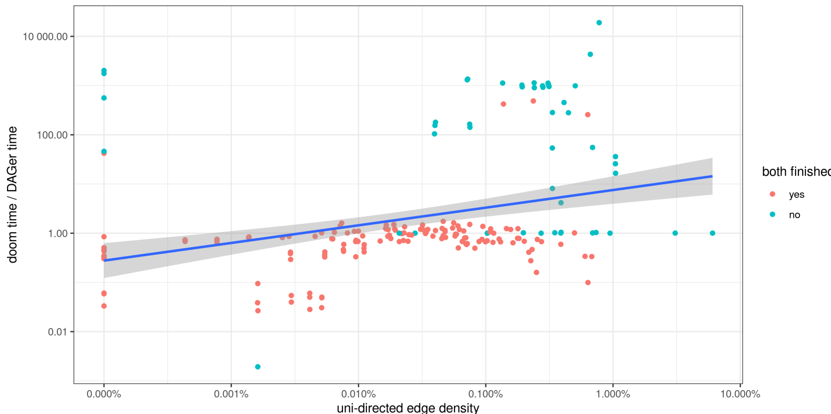


Figure 6 Ratio of Mount-Doom and DAGer running times on the PACE instances sorted by uni-directed edge density. On blue instances at least one of the solvers did not finish within 30 minutes. The unfinished solver's time is denoted by 30 minutes. The line shows a linear regression model with a confidence interval of 95%.

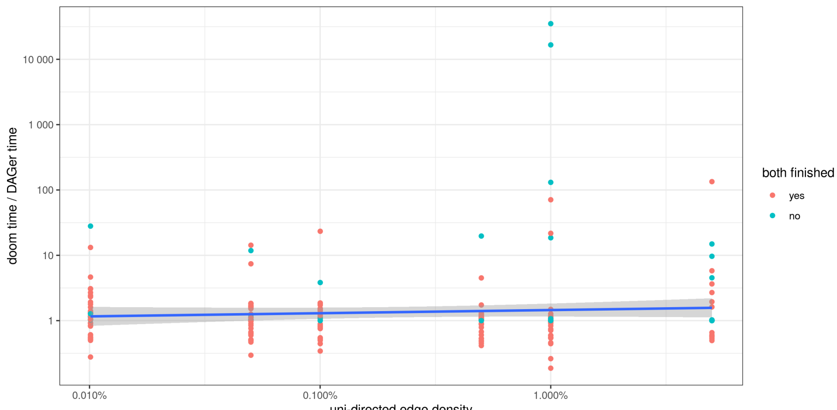


Figure 7 Ratio of Mount-Doom and DAGer running times on the Erdős-Rényi instances. On blue instances at least one of the solver's did not finish within 30 minutes. The unfinished solvers time is denoted by 30 minutes. The line shows a linear regression model with a confidence interval of 95%.

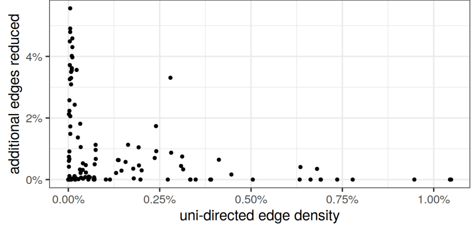


Figure 8 Percentage of total edges which are reduced with SHORTONE but not without by density of directed edges. The instances are taken from the set of all PACE instances after removing graphs which were fully reduced without SHORTONE (60 instances) and those that are not reduced at all even after including SHORTONE (2 instances).

Table 1 Comparison of number of instances solver faster by each solver for different graph types.

Data set	Solver	#Solved	#Solved faster	#Solved faster by factor 2
PACE	Mount-Doom	153	109	46
	DAGer	186	78	38
Erdős-Rényi	Mount-Doom	166	75	17
	DAGer	213	102	28

The instances in the plots in Figure 4 and Figure 5 are ordered by the density of the uni-directed edges. We can see that in Figure 4 i.e. the PACE instances, we are in general faster than the other two algorithms in those instances with few uni-directed edges. Figure 6 shows the ratios between running times of Mount-Doom and DAGer with a similar X-axis and a linear regression line with a confidence interval of 95%. The regression line clearly shows the trend that Mount-Doom is better on low uni-directed density instances. But such a trend cannot be seen in Erdős-Rényi graphs from similar plots in Figures Figure 5 and Figure 7, again showing that Mount-Doom is better suited for structured instances.

Our SHORTONE reduction rule relies on the existence of a vertex v with exactly one incoming and one outgoing uni-directed neighbor, such that for every $w \in N(v)$ there is a uni-directed neighbor of v adjacent to w in $G[PIE]$. Therefore, when we have a sparse directed graph with relatively many bi-directed edges, we expect SHORTONE to be well applicable. This can be attested by Figure 8 showing the performance of SHORTONE on PACE instances. When we have only a small percentage of uni-directed edges, we can reduce a considerable percentage of initial edges which we do not reduce without SHORTONE. With up to 5% of the initial edges, we see a clear advantage compared to not using SHORTONE. This advantage decreases with increasing density of uni-directed edges. But even for a uni-directed edge density of about 1%, where $G[DIR]$ becomes quite dense, we are still able to reduce some edges.

However, on the generated Erdős-Rényi graphs, which exhibit a uniform edge distribution, the required edge structure for the rule to be applied is much less likely to occur. This was confirmed by experimental results, where the inclusion of SHORTONE did not reduce the instances any further.

5 Conclusion

We give the description of our new exact algorithm Mount-Doom for DFVS, which won the third place in the PACE coding challenge 2022. We also present the results of extensive experiments we conducted to compare Mount-Doom to the PACE winner DAGer and a state of art solver Sdopt. We demonstrated that we clearly outpace Sdopt in terms of running time. Although DAGer beats our algorithm in many cases, our algorithm is still competitive, and is considerably faster in a big share of the instances. Especially, on structured instances with low uni-directed edge density Mount-Doom has an upper hand.

We also introduced a new reduction rule SHORTONE in our algorithm. We demonstrated that the reduction rule comes to good use in many of the instances, especially the structured instances with small uni-directed edge density.

The efficiency of our solver seems to not just depend on the uni-directed edge density but even more on structure, as can be seen by the different performance between the PACE instances and the Erdős-Rényi graphs. As further work, it would be interesting to study more

closely which structures are beneficial not just for our particular algorithm, but generally for the approach to reduce DFVS to VERTEX COVER. One could, for example, also directly attempt to develop an algorithm that has theoretical efficiency in the low uni-directed density regime.

References

1. Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. *Theoretical Computer Science*, 609:211–225, 2016.
2. Ali Baharev, Hermann Schichl, Arnold Neumaier, and Tobias Achterberg. An exact method for the minimum feedback arc set problem. *ACM Journal of Experimental Algorithmics*, 26:1.4:1–1.4:28, 2021. doi:10.1145/3446429.
3. Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. In *Proceedings of the SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1, pages 51–53, 2020.
4. Shaowei Cai, Kaile Su, Chuan Luo, and Abdul Sattar. NuMVC: An Efficient Local Search Algorithm for Minimum Vertex Cover. *Journal of Artificial Intelligence Research*, 46:687–716, 2013. doi:10.1613/jair.3907.
5. Srimat T Chakradhar, Arun Balakrishnan, and Vishwani D Agrawal. An exact algorithm for selecting partial scan flip-flops. *Journal of Electronic Testing*, 7(1):83–93, 1995.
6. Jianer Chen, Iyad A Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40-42):3736–3756, 2010.
7. Jianer Chen, Yang Liu, Songjian Lu, Barry O'Sullivan, and Igor Razgon. A Fixed-Parameter Algorithm for the Directed Feedback Vertex Set Problem. *Journal of the ACM*, 55:21:1–21:19, 2008.
8. Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer, 2015.
9. Reinhard Diestel. Graph Theory 3rd ed. *Graduate Texts in Mathematics*, 173, 2005.
10. M. Ayaz Dzulfikar, Johannes K. Fichte, and Markus Hecher. The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration (Invited Paper). In *International Symposium on Parameterized and Exact Computation (IPEC)*, volume 148, pages 25:1–25:23, 2019. doi:10.4230/LIPIcs.IPEC.2019.25.
11. Paul Erdős and Alfréd Rényi. On random graphs. *Publicationes Mathematicae*, 6:290–297, 1959.
12. Guy Even, Joseph Naor, Baruch Schieber, and Madhu Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, 1998.
13. Rudolf Fleischer, Xi Wu, and Liwei Yuan. Experimental study of fpt algorithms for the directed feedback vertex set problem. In *European Symposium on Algorithms (ESA)*, pages 611–622. Springer, 2009.
14. Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A Measure & Conquer Approach for the Analysis of Exact Algorithms. *Journal of the ACM*, 56(5):25:1–25:32, 2009. doi:10.1145/1552285.1552286.
15. Philippe Galinier, Eunice Lemamou, and Mohamed Wassim Bouzidi. Applying local search to the feedback vertex set problem. *Journal of Heuristics*, 19(5):797–818, 2013.
16. Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL: https://www.gurobi.com.
17. David G. Harris and N. S. Narayanaswamy. A faster algorithm for vertex cover parameterized by solution size. *CoRR*, abs/2205.08022, 2022. doi:10.48550/arXiv.2205.08022.
18. Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. WeGotYouCovered: The winning solver from the PACE 2019 challenge, vertex cover track. In *Proceedings of the SIAM Workshop on Combinatorial Scientific Computing (CSC)*, pages 1–11, 2020. doi:10.1137/1.9781611976229.1.

19 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

20 Rafael Kiesel and André Schidler. A Dynamic MaxSAT-based Approach to Directed Feedback Vertex Sets. In Gonzalo Navarro and Julian Shun, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 39–52. SIAM, 2023. doi:10.1137/1.9781611977561.ch4.

21 D.H. Lee and S.M. Reddy. On determining scan flip-flops in partial-scan designs. In *IEEE/ACM International Conference on Computer-Aided Design, (ICCAD)*, pages 322–325, 1990. doi:10.1109/ICCAD.1990.129914.

22 Mile Lemaic. *Markov-Chain-Based Heuristics for the Feedback Vertex Set Problem for Digraphs*. PhD thesis, Universität zu Köln, 2008. URL: https://kups.ub.uni-koeln.de/2547/1/Dissertation.pdf.

23 Hen-Ming Lin and Jing-Yang Jou. Computing minimum feedback vertex sets by contraction operations and its applications on cad. In *Proceedings of the IEEE International Conference On Computer Design, VLSI in Computers and Processors, (ICCD)*, pages 364–369. IEEE, 1999.

24 Daniel Lokshtanov, NS Narayanaswamy, Venkatesh Raman, MS Ramanujan, and Saket Saurabh. Faster parameterized algorithms using linear programming. *ACM Transactions on Algorithms (TALG)*, 11(2):1–31, 2014.

25 Tatiana Orenstein, Zvi Kohavi, and Irith Pomeranz. An optimal algorithm for cycle breaking in directed graphs. *Journal of Electronic Testing*, 7(1):71–81, 1995.

26 Panos M Pardalos, Tianbing Qian, and Mauricio GC Resende. A greedy randomized adaptive search procedure for the feedback vertex set problem. *Journal of Combinatorial Optimization*, 2(4):399–412, 1998.

27 Rick Plachetta and Alexander van der Grinten. Sat-and-reduce for vertex cover: Accelerating branch-and-reduce by sat solving. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 169–180, 2021. doi:10.1137/1.9781611976472.13.

28 Igor Razgon. Computing minimum directed feedback vertex set in $O(1.9977^n)$. In *Italian Conference on Theoretical Computer Science (ICTCS)*, pages 70–81. World Scientific, 2007.

29 Andreas Schindler and Rafael Kiesel. Dager. URL: https://github.com/ASchidler/dfvs.

30 Christian Schulz, Ernestine Großmann, Tobias Heuer, and Darren Strash. Pace 2022: Directed feedback vertex set. In *17th International Symposium on Parameterized and Exact Computation (IPEC 2022)*, 2022.

31 Adi Shamir. A linear time algorithm for finding minimum cutsets in reducible graphs. *SIAM Journal on Computing*, 8(4):645–655, 1979.

32 G Smith and R Walford. The identification of a minimal feedback vertex set of a directed graph. *IEEE Transactions on Circuits and Systems*, 22(1):9–15, 1975.

33 Matthias F Stallmann, Yang Ho, and Timothy D Goodrich. Graph profiling for vertex cover: Targeted reductions in a branch and reduce solver. *arXiv preprint arXiv:2003.06639*, 2020.

34 Ching-Chy Wang, Errol L Lloyd, and Mary Lou Soffa. Feedback vertex sets and cyclically reducible graphs. *Journal of the ACM*, 32(2):296–313, 1985.

35 Mingyu Xiao and Hiroshi Nagamochi. Exact algorithms for maximum independent set. *Information and Computation*, 255:126–146, 2017. doi:10.1016/j.ic.2017.06.001.

CompDP: A Framework for Simultaneous Subgraph Counting Under Connectivity Constraints

Kengo Nakamura
NTT Communication Science Laboratories, Kyoto, Japan
Graduate School of Informatics, Kyoto University, Japan

Masaaki Nishino
NTT Communication Science Laboratories, Kyoto, Japan

Norihito Yasuda
NTT Communication Science Laboratories, Kyoto, Japan

Shin-ichi Minato
Graduate School of Informatics, Kyoto University, Japan

Abstract

The subgraph counting problem computes the number of subgraphs of a given graph that satisfy some constraints. Among various constraints imposed on a graph, those regarding the connectivity of vertices, such as "these two vertices must be connected," have great importance since they are indispensable for determining various graph substructures, e.g., paths, Steiner trees, and rooted spanning forests. In this view, the subgraph counting problem under connectivity constraints is also important because counting such substructures often corresponds to measuring the importance of a vertex in network infrastructures. However, we must solve the subgraph counting problems multiple times to compute such an importance measure for every vertex. Conventionally, they are solved separately by constructing decision diagrams such as BDD and ZDD for each problem. However, even solving a single subgraph counting is a computationally hard task, preventing us from solving it multiple times in a reasonable time. In this paper, we propose a dynamic programming framework that simultaneously counts subgraphs for every vertex by focusing on similar connectivity constraints. Experimental results show that the proposed method solved multiple subgraph counting problems about 10–20 times faster than the existing approach for many problem settings.

2012 ACM Subject Classification Mathematics of computing → Graph theory

Keywords and phrases Subgraph counting, Connectivity, Zero-suppressed Binary Decision Diagram

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.11

Supplementary Material *Software*: `https://github.com/nttcslab/compdp-counting`

Funding This work was supported by JSPS KAKENHI Grant Number JP20H05963 and JST CREST Grant Number JPMJCR22D3.

1 Introduction

Given graph $G = (V, E)$, the *subgraph counting problem* computes the (possibly weighted) count of the subgraphs of G that satisfy some constraints such as each vertex's degree and the existence of cycles. More specifically, given edge weights $w_e^+, w_e^- \in \mathbb{R}$ for $e \in E$, this problem (exactly) computes the following value:

$$W(\mathcal{E}) := \sum_{E' \in \mathcal{E}} \prod_{e \in E'} w_e^+ \cdot \prod_{e \in E \setminus E'} w_e^-, \tag{1}$$

where $\mathcal{E} \subseteq 2^E$ is a family of the subsets of edges, i.e., *subgraphs*, that satisfy given constraints.

© Kengo Nakamura, Masaaki Nishino, Norihito Yasuda, and Shin-ichi Minato;
licensed under Creative Commons License CC-BY 4.0

21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis; Article No. 11; pp. 11:1–11:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This problem has been studied as a fundamental task in computer science [2, 1, 5, 4], and extensively studied in the area of *network reliability analysis* [21]. The most fundamental problem of network reliability analysis is computing the probability that the predetermined vertices will remain connected assuming that each edge fails independently with a given probability. This task is equivalent to the subgraph counting problem under *connectivity constraints*; a connectivity constraint is a topological constraint requiring that some pairs of vertices are connected and other pairs are disconnected.

A connectivity constraint is also fundamental in determining various graph substructures, such as paths, Steiner trees, spanning trees, and rooted spanning forests, in combination with other constraints, as described in Section 2. Counting these substructures also has great importance, especially for evaluating the importance of a vertex. For example, paths and Steiner trees on communication networks correspond to the routing of point-to-point and multi-site communication (e.g., see [10, 26]). Thus, the number of paths or Steiner trees passing vertex v is an importance measure for v in this communication network, since its failure causes the lost of this number of communication routing. A cycle passing through source vertex s and another vertex v constitutes a vertex-disjoint two paths between s and v, and counting such cycles corresponds to the number of non-blocking pairs of communication routings from s to v [11]. A rooted spanning forest (RSF) rooted at $r_1, \ldots, r_k$ corresponds to a (electrical) distribution network whose substations are located at $r_1, \ldots, r_k$ [12]. When we add a new substation to v, the number of RSFs rooted at $r_1, \ldots, r_k, v$ (given other constraints such as electric constraints) provides flexibility of the distribution network.

In evaluating such an importance measure for every vertex v, we generally have to solve the subgraph counting problem for every v. That is, we must compute multiple count values $W(\mathcal{E}_{v_1}), \ldots, W(\mathcal{E}_{v_n})$ for different families of subgraphs $\mathcal{E}_{v_1}, \ldots, \mathcal{E}_{v_n}$. However it was proven that the network reliability evaluation described above is in #P-complete [32], a computationally challenging class, and computing $W(\mathcal{E})$ in the presence of other constraints is equally as difficult in general. Even a practically fast algorithm for computing $W(\mathcal{E})$ described below may take several minutes or more for a graph with hundreds of edges. Subgraph counting for every vertex described above seems computationally much more difficult since we have to repeatedly solve cumbersome counting tasks.

This paper proposes a practically fast algorithm for *simultaneously* counting subgraphs for every vertex (formally defined in Section 2). Here, "simultaneously" means that we build only one data structure for obtaining all count values $W(\mathcal{E}_{v_1}), \ldots, W(\mathcal{E}_{v_n})$. Our contribution is summarized as follows:

- Our proposed method enables us to simultaneously count such graph substructures as paths, cycles, Steiner trees, and RSFs by sophisticated dynamic programming (DP) on the built data structure.
- Complexity analyses show that the proposed method solves subgraph counting for every vertex $O(n)$ times faster than the baseline method that separately solves each counting, where n is the number of vertices.
- We empirically confirmed that the proposed algorithm solved subgraph counting for every vertex around 10–20 times faster than the baseline method.

1.1 Related Works

The study of network reliability problems, i.e., subgraph counting problems under constraints of the from "specified vertices must be connected," has a long history. This problem is known as #P-complete [32], meaning that it is computationally tough. Meanwhile, various algorithms have been proposed for this problem, e.g., sum-of-disjoint product [6] with

tieset [25] or cutset enumeration [31], and factoring [33]. However, currently the practically fastest algorithm, originated by Hardy et al. [8] and Herrmann [9], directly constructs a data structure called a *binary decision diagram* (*BDD*) [3]. Their method successfully computes network reliability for real topologies with around 200 edges. Here BDD is used as a tool that represents the family of subgraphs where predetermined vertices are connected; after it is built, the counting problem can be solved by a simple DP on it.

Other studies exist for counting subgraphs with additional constraints other than connectivity. The simplest method enumerates all the substructures [23, 28] such as paths and spanning trees. Especially, the enumeration of spanning trees [28] corresponds to computing the Tutte polynomial of a graph, which can be used for many kinds of graph counting problems. However, since there might be an exponential number of substructures, enumeration suddenly becomes intractable with the growth of graph size. For practically fast counting, indexing the constrained subgraphs into BDD-like structures has also been studied. Sekine et al. [27] designed an algorithm to build a BDD representing all the spanning trees to compute the Tutte polynomial. Knuth [18] proposed a very efficient scheme called Simpath that indexes all the simple paths in a *zero-suppressed BDD* (*ZDD*) [20], which is a variant of BDD. By expanding such research, Kawahara et al. [17] proposed a *frontier-based search* (FBS), which can build a ZDD of various graph substructures. Since ZDD also allows a simple DP for counting, FBS can be used for practically fast subgraph counting. Our proposed algorithm is also based on FBS. Subgraph counting is also studied in the context of parameterized complexity theory [5, 4], although their interest often focuses on theoretical aspects. To the best of our knowledge, no works have outperformed the BDD/ZDD-based methods in practically solving subgraph counting problems, including network reliability evaluations.

In 2021, Nakamura et al. [22] proposed an algorithm for network reliability that simultaneously computes the probability of connecting to servers for every client. It essentially solves counting problems for every vertex and runs much faster than the baseline where each client's reliability is computed separately. However, it can only deal with the constraints of the form "all the specified vertices are connected" and cannot accept the constraints of disconnection and others. Technically, the proposed method can deal with these constraints by utilizing the ZDD structures [20] and the FBS [17]. While the existing method [22] relies on a BDD-like structure that is not truly a BDD, we build a legitimate ZDD by FBS, enabling us to combine such existing ZDD algorithms as Apply [20] and subsetting [15].

1.2 Organization of Paper

The rest of this paper is organized as follows. Section 2 describes the preliminary and the formal statement of the problem we solved. Section 3 gives the overview of the proposed method. Section 4 introduces the ZDD and the frontier-based search that are used in the proposed method. Section 5 details the proposed method, and Section 6 analyzes the computational complexity of it. Section 7 empirically compares the proposed method with the baseline in terms of computational time, and Section 8 gives a conclusion.

We give the list of acronyms and notations used in this paper in Table 1.

2 Problem Statement

Before proceeding to our problem statement, we introduce a notion that represents the connectivity constraint in the same manner as Kawahara et al. [17]. As described in Section 1, a connectivity constraint requires that some pairs of vertices are connected and other pairs are disconnected. We represent the connectivity constraint by *subpartition* P

Table 1 Acronyms and notations.

Acronym	
RSF	Rooted Spanning Forest
DP	Dynamic Programming
BDD	Binary Decision Diagram
ZDD	Zero-suppressed binary Decision Diagram
FBS	Frontier-Based Search
DAG	Directed Acyclic Graph
Notation	
Frequently used notations	
$G = (V, E)$	undirected graph with vertex set V and edge set E
n, m	$= \|V\|, \|E\|$: the number of vertices and edges in the graph
w_e^+, w_e^-	$\in \mathbb{R}$: edge weights for $e \in E$
$W(\mathcal{E})$	the value of subgraph counting given $\mathcal{E} \subseteq 2^E$ (family of subgraphs)
P	subpartition of V representing connectivity constraint
P^*	subpartition of $V \cup \{*\}$ containing one wildcard $*$
$P^*[v]$	connectivity constraint obtained by substituting $*$ in P^* with $v \in V$
$P^*[]$	connectivity constraint obtained by removing $*$ from P^*
$\mathcal{C}(P)$	$\subseteq 2^E$: family of subgraphs satisfying connectivity constraint P
$\mathcal{F}$	$\subseteq 2^E$: base set in our problem
$\mathsf{count}(v)$	$= W(\mathcal{F} \cap \mathcal{C}(P^*[v]))$: count value for vertex $v \in V$ computed in our problem
Notations for ZDD	
$Z = (N, A)$	ZDD with node set N and arc set A
$\top, \bot$	terminal nodes of ZDD
$\hat{r}$	$\in N$: root node of ZDD
$\mathsf{lo}(\hat{n}), \mathsf{hi}(\hat{n})$	$\in A$: lo-arc and hi-arc outgoing from ZDD node $\hat{n}$
$\hat{n}^-, \hat{n}^+$	$\in N$: lo-child and hi-child of ZDD node $\hat{n}$
$\mathsf{lb}(\hat{n})$	$\in \mathbb{Z}$: label of ZDD node $\hat{n}$
L_i	i-th level of ZDD, i.e., the set of ZDD nodes whose label is i
R	(directed) path in ZDD
$\mathcal{R}_Z(\hat{n}, \hat{n}')$	set of paths between ZDD nodes $\hat{n}$ and $\hat{n}'$ in ZDD Z
$\mathcal{S}(Z)$	$\subseteq 2^E$: family of subgraphs represented by ZDD Z
$E(R)$	$\subseteq E$: subgraph represented by path R in ZDD
$W_p(R)$	$\in \mathbb{R}$: path product of path R in ZDD defined in (3)
$\hat{n}.\mathsf{p}$, $\hat{n}.\mathsf{r}$	sum of path products of the paths in $\mathcal{R}_Z(\hat{r}, \hat{n})$ and $\mathcal{R}_Z(\hat{n}, \top)$
Notations for explaining existing and proposed methods	
$E_{<i}$	$= \{e_1, \ldots, e_{i-1}\}$: processed edges
$E_{\geq i}$	$= \{e_i, \ldots, e_m\}$: unprocessed edges
F_i	$\subseteq V$: i-th frontier, the vertices appearing in both $E_{<i}$ and $E_{\geq i}$
V_P	$\subseteq V$: set of vertices appearing in connectivity constraint P
V'_{P*}	$\subseteq V$: set of vertices in the set in P^* containing $*$
V''_{P*}	$\subseteq V$: set of vertices present in P^* and not included in V'_{P*}
$\hat{n}.\mathsf{comp}$	partition of F_i maintaining the connectivity among F_i
$\hat{n}.\mathsf{vset}$	connectivity constraint maintaining the connectivity among $F_i \cup V_P$
$\mathcal{R}_v$	$\subseteq \mathcal{R}_Z(\hat{r}, \top)$: set of paths whose corresponding subgraph satisfies $(\#_v)$
$\mathcal{R}_{v,\hat{n}}$	$\subseteq \mathcal{R}_v$: set of paths that passes through ZDD node $\hat{n}$
B	$\in \hat{n}.\mathsf{comp}$: set contained in comp, i.e., connected component
$\mathcal{R}_{\hat{n},B}$	$\subseteq \mathcal{R}_Z(\hat{n}, \top)$: set of paths associated with ZDD node $\hat{n}$ and set $B \in \hat{n}.\mathsf{comp}$
$\hat{n}.\mathsf{q}[B]$	sum of path products of the paths in $\mathcal{R}_{\hat{n},B}$
$\hat{n}.\mathsf{q}^-[B]$	sum of path products of the paths in $\mathcal{R}_{\hat{n},B}$ traversing $\mathsf{lo}(\hat{n})$
$\hat{n}.\mathsf{q}^+[B]$	sum of path products of the paths in $\mathcal{R}_{\hat{n},B}$ traversing $\mathsf{hi}(\hat{n})$
$V'_{\hat{n}.\mathsf{vset}}$	$\subseteq V$: set of vertices in the set in $\hat{n}.\mathsf{vset}$ containing $*$
$V''_{\hat{n}.\mathsf{vset}}$	$\subseteq V$: set of vertices present in $\hat{n}.\mathsf{vset}$ and not included in $V'_{\hat{n}.\mathsf{vset}}$
$Z_\mathcal{F}$	ZDD representing base set $\mathcal{F}$
Notations for conducting complexity analysis	
$Z_{\mathrm{FBS}(P)}$	ZDD bulit by FBS with connectivity constraint P
c_P	the number of set in connectivity constraint P
v_P	the number of vertices in connectivity constraint P excluding $*$
fw	$= \max_i \|F_i\|$: frontier width
D_k	k-th Bell number

of vertex set V of graph $G = (V, E)$, where a subpartition of V is a set of pairwise disjoint subsets of V. P imposes the following constraints: (i) for any pair of vertices v, v' in the same set in P, they must be connected, and (ii) for any pair of vertices v, v' in different sets in P, they must not be connected.

We extend the notion by introducing exactly one *wildcard* $*$, which will be replaced by a vertex to represent various connectivity constraints. Let P^* be a subpartition of $V \cup \{*\}$ that must contain exactly one $*$. For $v \in V$, let $P^*[v]$ be the connectivity constraint obtained by substituting $*$ in P^* with v. Additionally, let $P^*[]$ be the connectivity constraint obtained by simply removing $*$ from P. Note that if $v \in V$ is already present in P^*, $P^*[v]$ equals (i) $P^*[]$ if v is in the same set in P^* that contains $*$; or (ii) an inconsistent constraint. For example, for $P^* = \{\{v_1, v_2, *\}, \{v_3\}\}$, $P^*[v_4] = \{\{v_1, v_2, v_4\}, \{v_3\}\}$, $P^*[v_2] = P^*[] = \{\{v_1, v_2\}, \{v_3\}\}$, and $P^*[v_3]$ is an inconsistent constraint.

Now we proceed to our problem definition. In our problem, we are given connected undirected graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, connectivity constraint P^* containing exactly one wildcard $*$, family $\mathcal{F}$ of subgraphs, i.e., subsets of edges, and weights $w_e^+, w_e^- \in \mathbb{R}$ for each $e \in E$. For connectivity constraint P, let $\mathcal{C}(P)$ be the family of subgraphs satisfying P. Our goal is to compute the following value

$$\mathsf{count}(v) := W(\mathcal{F} \cap \mathcal{C}(P^*[v])) \quad \text{for every } v \in V. \tag{2}$$

The problems described in Section 1 can be covered by our problem.

- *Path*: Given $s, t \in V$, we set $P^* = \{\{s, t, *\}\}$ and let $\mathcal{F}$ be a family of subgraphs where (i) the degree of s and t is 1, and the others have degree 0 or 2, and (ii) there are no cycles. Then $\mathsf{count}(v)$ equals the number of simple s, t-paths that pass through v.
- *Cycle*: Given $s \in V$, we set $P^* = \{\{s, *\}\}$, and let $\mathcal{F}$ be a family of subgraphs where (i) the degree of each vertex is 0 or 2, and (ii) there is exactly one connected component. Then $\mathsf{count}(v)$ equals the number of cycles starting from s that pass through v.
- *Steiner tree*: Given $T \subseteq V$, we set $P^* = \{T \cup \{*\}\}$ and let $\mathcal{F}$ be a family of subgraphs where there are no cycles and exactly one connected component. Then $\mathsf{count}(v)$ equals the number of T-Steiner trees containing v, where T-Steiner trees are the trees connecting all the T vertices.
- *Rooted spanning forest*: Given $T = \{r_1, \ldots, r_k\} \subseteq V$, we set $P^* = \{\{r_1\}, \ldots, \{r_k\}, \{*\}\}$, and let $\mathcal{F}$ be a family of subgraphs where (i) every vertex has degree at least 1, (ii) there are no cycles, and (iii) there are exactly $(k+1)$ connected components. Then $\mathsf{count}(v)$ equals the number of rooted spanning forests rooted at $r_1, \ldots, r_k$ and v.

In addition, the problem setting of Nakamura et al. [22] essentially counts the subgraphs where given vertex set $T \subseteq V$ and vertex v are connected for every v and can be recovered by $P^* = \{T \cup \{*\}\}$ and $\mathcal{F} = 2^E$. Although here we list only topological constraints for $\mathcal{F}$, we can also impose non-topological constraints with $\mathcal{F}$, such as knapsack constraints.

3 Overview of Proposed Algorithm

First, we explain case $\mathcal{F} = 2^E$. Given connectivity constraint P^*, the baseline method, which separately computes the count value for every v by FBS [17], builds a ZDD with connectivity constraint $P^*[v]$ for every $v \in V$. Here ZDD compactly represents a family of subgraphs by a rooted directed acyclic graph. By FBS with $P^*[v]$, a ZDD representing $\mathcal{C}(P^*[v])$ is built and allows a simple DP for computing $\mathsf{count}(v) = W(\mathcal{C}(P^*[v]))$. The details of ZDD and FBS are explained in Section 4.

Figure 1 Overview of proposed algorithm.

Similarly, the proposed method builds a ZDD by FBS. However, unlike the baseline method, we build only one ZDD Z for P^*, which represents $\mathcal{C}(P^*[])$. Instead, we retain the information used in the FBS for building Z, comp and vset in each node of Z, both of which are discarded after the FBS in the baseline method. Since comp and vset provide rich information for connectivity among vertices, we fully exploit them to perform a more sophisticated DP, yielding for each node of Z three kinds of values, p, q, and r. Their definitions are described in Section 5.1. By using them, we can compute count(v) values for every $v \in V$. Since the computation of p, q, r, and count(v) can be performed in time proportional to the execution of FBS, the proposed algorithm runs faster than the baseline. We fully describe the computation of the count(v) values in Section 5.1 and those of p, q, and r (the DP procedure) in Section 5.2.

Finally, we deal with case $\mathcal{F} \neq 2^E$. For it, we first construct a ZDD $Z_{\mathcal{F}}$ by the existing methods. Then, we construct one ZDD Z that represents $\mathcal{F} \cap \mathcal{C}(P^*[])$ whose nodes have comp and vset. This can be performed by exploiting existing techniques of constructing a ZDD of set intersection, such as Apply [20] and subsetting [15]. After Z is built, we can perform the same DP scheme as above. We describe taking the set intersection in Section 5.3. An overview of the proposed method is given in Fig. 1.

By changing base set $\mathcal{F}$ and connectivity constraint P^*, the proposed algorithm can solve various subgraph counting problems, as in Section 2. We named our proposed algorithm *compDP* since it fully uses information comp.

3.1 Intuition and Idea behind the Proposed Algorithm

We describe the high-level idea behind the proposed algorithm. As described later, ZDD, which is a rooted and layered directed acyclic graph, represents a family of subgraphs as the set of paths from the root to a terminal node. By defining the *path product* of a path by the weights along this path (precise definition is later), the count value equals the sum of path products of these paths. The intuitive for the proposed algorithm is as follows: Let $\mathcal{E}_v = \mathcal{F} \cap \mathcal{C}(P^*[v])$. To compute count($v$) $= W(\mathcal{E}_v)$ for every $v \in V$, it is sufficient to build a ZDD representing $\mathcal{E}_v$ for every v. However, since $\mathcal{E}_v$ and $\mathcal{E}_w$ ($v \neq w$) are similar families stemming from the common constraint P^*, the ZDDs representing them also are expected to exhibit similar structures. We use such similarities to reduce the computation.

More specifically, we use the following step-by-step ideas: First, since $\mathcal{C}(P^*[v]) \subseteq \mathcal{C}(P^*[])$ for any $v \in V$, $\mathcal{F} \cap \mathcal{C}(P^*[v])$ is represented by the subset of the paths within the ZDD representing $\mathcal{F} \cap \mathcal{C}(P^*[])$. Second, these paths can be decomposed into former and latter parts; the former is the paths from the root to a specific layer, and the latter is the paths from the specific layer to the terminal. The count value count(v) can be represented by the sums of path products of the former part and those of the latter part. Third, when considering such a decomposition for every $v \in V$, we can reuse the values of "the sums of path products of the former part" (p in the proposed algorithm) and "those of the latter part" (q and r in the proposed algorithm). Thus, by pre-computing them by a DP, we can compute count(v) for every $v \in V$ with these values.

4 ZDD and Frontier-based Search

4.1 Zero-suppressed Binary Decision Diagrams (ZDDs)

Zero-suppressed binary decision diagram (ZDD) [20] $Z = (N, A)$ is a rooted directed acyclic graph (DAG)-shaped data structure representing a family of subsets of edges E;[1] the root node is denoted by $\hat{r} \in N$. Node set N has two special nodes $\top$ and $\bot$ called *terminals* and other internal nodes. Each internal node $\hat{n}$ has exactly two outgoing arcs called *lo-arc* $\mathsf{lo}(\hat{n})$ and *hi-arc* $\mathsf{hi}(\hat{n})$ and *label* $\mathsf{lb}(\hat{n})$ that is an integer of range $[1, m]$. We respectively call the child nodes of $\hat{n}$ pointed by $\mathsf{lo}(\hat{n})$ and $\mathsf{hi}(\hat{n})$ *lo-child* $\hat{n}^-$ and *hi-child* $\hat{n}^+$ of $\hat{n}$. Labels must be ordered in an ascending manner, i.e., $\mathsf{lb}(\hat{n}) < \mathsf{lb}(\hat{n}^-)$ and $\mathsf{lb}(\hat{n}) < \mathsf{lb}(\hat{n}^+)$ must hold for every $\hat{n}$. For convenience, we set the labels of the terminal nodes to $m + 1$. A ZDD is *normalized* if for every internal node $\hat{n}$, each of $\hat{n}^-$ and $\hat{n}^+$ is either $\bot$ or a node whose label is exactly $\mathsf{lb}(\hat{n}) + 1$. Size $|Z|$ of ZDD Z is defined as the number of nodes in Z.

For $\hat{n}, \hat{n}' \in N$, $\mathcal{R}_Z(\hat{n}, \hat{n}')$ denotes the set of paths from $\hat{n}$ to $\hat{n}'$. Given a predefined order of edges, $e_1, \ldots, e_m$, ZDD Z represents a family of subgraphs $\mathcal{S}(Z) \subseteq 2^E$ by $\mathcal{R}_Z(\hat{r}, \top)$. For path R in Z, we associate subset $E(R) \subseteq E$ where $e_i \in E(R)$ if and only if R traverses a hi-arc outgoing from a node with label i. Then $\mathcal{S}(Z) = \{E(R) \mid R \in \mathcal{R}_Z(\hat{r}, \top)\}$. For example, in Fig. 2(b), path 1-(hi)-2-(lo)-3-(lo)-4-(hi)-5-(hi)-$\top$ indicates that $\{e_1, e_4, e_5\} \in \mathcal{S}(Z)$.

Normalized ZDDs can be used as an efficient tool for subgraph counting. Let Z be a normalized ZDD, and let $R \in \mathcal{R}_Z(\hat{n}, \hat{n}')$ be an arbitrarily chosen path. Given weights $w_e^+, w_e^- \in \mathbb{R}$ for each $e \in E$, we define the *path product* of R by

$$W_p(R) := \prod_{e \in E(R)} w_e^+ \cdot \prod_{e \in \{e_{\mathsf{lb}(\hat{n})}, \ldots, e_{\mathsf{lb}(\hat{n}')-1}\} \setminus E(R)} w_e^- . \tag{3}$$

In other words, $W_p(R)$ is the product of w_e^+ for the edges in $E(R)$ and w_e^- for the edges not in $E(R)$. We also define value $\hat{n}.\mathsf{p}$ for ZDD node $\hat{n}$ by the sum of path products of the paths in $\mathcal{R}_Z(\hat{r}, \hat{n})$. By definition, $W(\mathcal{S}(Z))$ equals $\top.\mathsf{p}$. Moreover, although $\hat{n}.\mathsf{p}$ is defined as the sum of a possibly exponential number of path products, its value can efficiently be computed by a DP. Simple calculation shows the following equation for a node other than $\hat{r}$ or $\bot$:

$$\hat{n}.\mathsf{p} = \sum_{\hat{n}' : (\hat{n}')^- = \hat{n}} w^-_{e_{\mathsf{lb}(\hat{n})}} \cdot \hat{n}'.\mathsf{p} + \sum_{\hat{n}' : (\hat{n}')^+ = \hat{n}} w^+_{e_{\mathsf{lb}(\hat{n})}} \cdot \hat{n}'.\mathsf{p}. \tag{4}$$

Starting with $\hat{r}.\mathsf{p} = 1$, by applying (4) in a top-down manner along Z, we can compute the value of $\top.\mathsf{p} = W(\mathcal{S}(Z))$ in time proportional to the number of nodes in Z.

4.2 Frontier-based Search

A frontier-based search (FBS) [17] is an efficient method for constructing a normalized ZDD that represents a family of subgraphs satisfying some constraints. Below we explain its procedure of given connectivity constraint P (without $*$).

We start with a naive way for constructing ZDD. Given order of edges $e_1, \ldots, e_m$, we decide one by one whether e_i is excluded or included in the subgraph. This generates a binary decision tree like Fig. 2(c). After deciding every link's exclusion or inclusion, all subgraphs are enumerated at the bottom of the decision tree, and we can judge whether the constraints

[1] Although a ZDD can represent a family of subsets of arbitrary base set X, here we explain it as a tool for representing a family of edge subsets for simplicity.

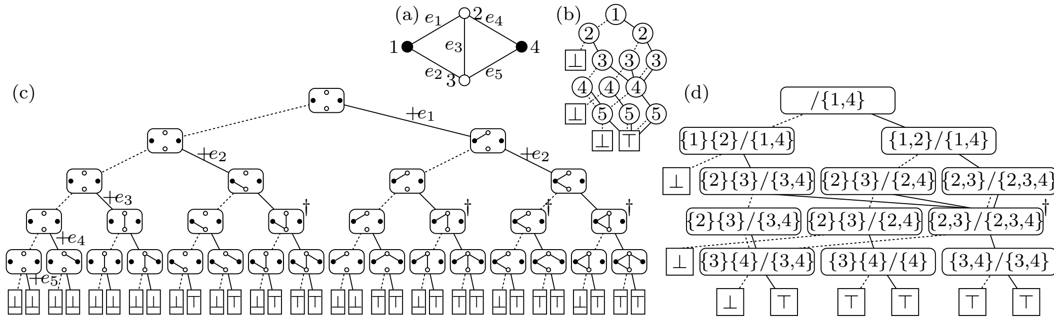


Figure 2 (a) Example of graph. (b) Example of (normalized) ZDD. Dashed and solid lines indicate lo- and hi-arcs, and an integer inside a node represents a label. (c) Binary decision tree made by deciding one by one whether e_i is excluded or included. (d) ZDD made by FBS where $P = \{\{1, 4\}\}$. Two subpartitions of vertices inside a node represent comp and vset.

are satisfied one by one. The example in Fig. 2(c) judges whether the black vertices (1 and 4) are connected. This decision tree has a property where the collection of paths from root to $\top$ corresponds to the family of subgraphs satisfying the constraints. After making the decision tree, ZDD can be constructed by merging the identical subtrees of it. For example, the four nodes marked † in Fig. 2(c) have an identical pattern of leaves: $\bot, \top, \top, \top$. Even if we merge them, the property that the collection of paths from root to $\top$ corresponds to a family of satisfying subgraphs is not broken.

However, since there are 2^m subgraphs, the size of the decision tree must be exponential. For efficient construction of ZDD, we try to detect identical subtrees without constructing them. Formally, the identicalness of subtrees can be stated as follows. Let $E_{<i} = \{e_1, \dots, e_{i-1}\}$ and $E_{\geq i} = \{e_i, \dots, e_m\}$. We call the subset of edges after deciding e_{i-1}'s exclusion or inclusion the *i-th subgraph*. Note that the i-th subgraphs are subsets of $E_{<i}$. We consider the following equivalence relation for i-th subgraphs $X, X' \subseteq E_{<i}$: (§) For any $Y \subseteq E_{\geq i}$, whether subgraph $X \cup Y$ or $X' \cup Y$ satisfies the constraints is equivalent. If (§) holds, the subtree rooted at X and that rooted at X' are identical.

Condition (§) is met for $X, X' \subseteq E_{<i}$ if X and X' share identical connectivity among V, i.e., for any $v, v' \in V$, whether v and v' are connected is equivalent. This is because if X and X' share identical connectivity, so do $X \cup Y$ and $X' \cup Y$. Moreover, if X and X' have an identical connectivity, so do $X \cup \{e_i\}$ and $X' \cup \{e_i\}$. This enables us to build a decision "diagram" by the following procedure. In building a decision tree like Fig. 2(c) in a breadth-first manner, we merge some subgraphs if they exhibit an identical connectivity.

The FBS further refines this idea by focusing on the connectivity among a limited subset of vertices. Let F_i, called *i-th frontier*, be the vertices appearing in both $E_{<i}$ and $E_{\geq i}$. Also, let V_P be the set of vertices appearing in connectivity constraint P. Then, it can be proved that condition (§) is satisfied if X and X' have an identical connectivity among $F_i \cup V_P$. Intuitively, this is because every vertex $v \in V \setminus V_P$ that does not appear in $E_{\geq i}$ can be equated with vertex $v' \in F_i \cup V_P$ if v is connected to v', or it can be ignored if it is not connected to any vertex in $F_i \cup V_P$. FBS generates a diagram by a breadth-first manner such that the nodes exhibiting identical connectivity among $F_i \cup V_P$ are all merged.

Algorithm 1 is pseudocode of FBS. The connectivity among $F_i \cup V_P$ is maintained by two subpartitions of vertices: comp and vset. Here comp maintains the connectivity among F_i while vset maintains the connectivity among the sets in comp and the vertices in V_P. More specifically, comp is a partition of F_i such that $v, v' \in F_i$ are in the same set if and only if

Algorithm 1 Frontier-based search for connectivity constraint P. Underlined part in line 16 is added if it is used for a subroutine of proposed method.

```
1  r̂.comp ← {}, r̂.vset ← P, L_1 ← {r̂}, L_i ← ∅ (i = 2, ..., m+1)
2  for i ← 1 to m do                                          // e_i = {v, v'}
3    foreach n̂ ∈ L_i do
4      foreach f ∈ {−, +} do
5        n̂' ← n̂                         // V_{n̂'.vset} is the vertices present in n̂'.vset
6        if f = + and v, v' ∈ V_{n̂'.vset} and n̂'.vset[v] ≠ n̂'.vset[v'] then
7          n̂' ← ⊥ and goto finish                    // v and v' must not be connected
8        foreach u ∈ {v, v'} \ F_i do                 // Vertices entering the frontier
9          n̂'.comp ← n̂'.comp ∪ {{u}}                   // Add u as an isolated vertex
10       if f = + and n̂'.comp[v] ≠ n̂'.comp[v'] then      // Connecting two components
11         Merge n̂'.comp[v] and n̂'.comp[v'] into one
12         if v ∈ V_{n̂'.vset} and v' ∉ V_{n̂'.vset} then Add vertices in n̂'.comp[v'] to n̂'.vset[v]
13         if v ∉ V_{n̂'.vset} and v' ∈ V_{n̂'.vset} then Add vertices in n̂'.comp[v] to n̂'.vset[v']
14       foreach u ∈ {v, v'} \ F_{i+1} do              // Vertices leaving from frontier
15         if {u} ∈ n̂'.comp then              // Component containing u leaves frontier
16           if u ∈ V_{n̂'.vset} and {u} ∉ n̂'.vset and {u, *} ∉ n̂'.vset then
17             n̂' ← ⊥ and goto finish            // u must be connected to w ∈ n̂'.vset[u]
18         Remove u from n̂'.comp and n̂'.vset if exists
19       if n̂' ≠ ⊥ then
20         if i = m then n̂' ← ⊤                        // All conditions are satisfied
21         else if ∃n̂'' ∈ L_{i+1} s.t. n̂''.comp = n̂'.comp and n̂''.vset = n̂'.vset then
22           n̂' ← n̂''                                      // Already generated node
23         else L_{i+1} ← L_{i+1} ∪ {n̂'}                       // Newly generated node
24       finish: n̂^f ← n̂'                            // Set lo- or hi-child of n̂ to n̂'
```

they are connected, and vset maintains the connectivity constraint such that the vertices in the same set must be connected and those in different sets must be disconnected. Here comp$[v]$ denotes the set in comp containing v, and as is the same with vset$[v]$. By starting with comp $= \{\}$ and vset $= P$ (Line 1), the algorithm repeatedly updates comp and vset by excluding ($f = -$) or including ($f = +$) e_i; lines 5–13 are the update procedure. We set the destination of lo-arc ($f = -$) or hi-arc ($f = +$) to the node with the updated comp and vset (line 24). If the node having identical comp and vset has already been generated, we set it to the already generated node (lines 21–22); otherwise, we newly generate a node (line 23). When either of the following occurs, we *prune* the node, i.e., setting the destination of an arc to $\bot$: (I) Two vertices are connected that are in different sets of vset (lines 6–7). This violates the disconnection requirement of vset. (II) comp$[v]$ leaves the frontier, but vset$[v]$ has vertex v' other than v (lines 16–17). In this case v will never be connected with v', violating the connection requirement of vset. If the search proceeds to the final level without being pruned, it reaches $\top$, i.e., constraint P is satisfied (line 20).

For example, Fig. 2(d) is the result of FBS given the graph in Fig. 2(a) and the constraint that vertices 1 and 4 must be connected. We now focus on the left, 2nd level node: "{1}{2}/{1,4}". Here comp $= \{1\}\{2\}$ denotes two components, the one including 1 and the one including 2, and vset $= \{1, 4\}$ represents that 1 and 4 must be connected. If we exclude $e_2 = \{1, 3\}$, the component including 1 is left isolated because $F_3 = \{2, 3\}$ does not include 1. However, since this contradicts that 1 and 4 must be connected, the lo-child is $\bot$ (pruned). If we include e_2, the component including 1 becomes one that includes 3 at the next level. The constraint that 1 and 4 must be connected can be rewritten as that 3 and 4 must be connected. Thus, hi-child's comp is {2}{3} and vset is {3,4}. Here the four nodes marked † in Fig. 2(c) are treated as only one marked node in Fig. 2(d).

5 Details of Proposed Method

First, we assume $\mathcal{F} = 2^E$; this assumption is removed in Section 5.3. As in Section 3, the proposed method first builds ZDD Z representing $\mathcal{C}(P^*[])$. To explain the meaning and procedure for this, we observe the relationship between $P^*[]$ and $P^*[v]$. Let V'_{P^*} be the vertices in the set in P^* containing $*$, and let V''_{P^*} be the other vertices present in P^*. From the definition, $P^*[v]$ imposes the following additional constraint on $P^*[]$:

($\#_v$) v must be connected with the vertices in V'_{P^*}, and v must be disconnected from the vertices in V''_{P^*}.

That is, $\mathcal{C}(P^*[v]) = \{X \mid X \in \mathcal{C}(P^*[]),\ X \text{ satisfies } (\#_v)\}$. Now the constraint $P^*[v]$ is decomposed into $(\#_v)$ and $P^*[]$, where $(\#_v)$ involves only the connectivity around v and $P^*[]$ represents the other constraints. The fact that $(\#_v)$ concerns only the connectivity around v enables us to compute $\mathsf{count}(v)$ for every v with only one ZDD Z representing $\mathcal{C}(P^*[])$, as described in the subsequent sections.

Meanwhile, during the procedure of FBS, we must remember which set in vset corresponds to the set in P containing $*$ since we use it for the subsequent computation. To achieve this, we just consider $*$ in P^* a special vertex. More specifically, we let $\hat{r}.\mathsf{vset} \leftarrow P^*$ in line 1 of Algorithm 1 and add the underlined part of line 16. By adding the underlined part, the pruning condition (II) simply discards $*$ even if $\mathsf{vset}[v]$ contains $*$. Thus, Z finally represents $\mathcal{C}(P^*[])$, while each vset has at most one set containing $*$.

5.1 Computation with Intermediate Level of Diagram

Let $\mathcal{R}_v$ be the paths in $\mathcal{R}_Z(\hat{r}, \top)$ whose corresponding subgraph satisfies $(\#_v)$. As stated above, $\mathsf{count}(v)$ equals the sum of path products of the paths in $\mathcal{R}_v$. Here we focus on i-th level L_i of Z where $v \in F_i$.[2] For node $\hat{n} \in L_i$ with label i, let $\mathcal{R}_{v,\hat{n}}$ be the paths in $\mathcal{R}_v$ passing through $\hat{n}$. Since Z is normalized, every $\hat{r}$-$\top$ path in Z passes exactly one node in L_i. This means that $\mathsf{count}(v)$ can be represented as the sum of $\sum_{R \in \mathcal{R}_{v,\hat{n}}} W_p(R)$ over $\hat{n} \in L_i$.

We further decompose $\sum_{R \in \mathcal{R}_{v,\hat{n}}} W_p(R)$ by focusing on $\hat{n} \in L_i$. Since $\hat{n}.\mathsf{comp}$ maintains the connectivity among F_i, the sets in $\hat{n}.\mathsf{comp}$ are indeed connected components. Since the connectivity around v can be translated into that around connected component $B = \hat{n}.\mathsf{comp}[v]$, $(\#_v)$ can be restated as a constraint on $B = \hat{n}.\mathsf{comp}[v]$:

($\#'_B$) Connected component B must be connected with the vertices in V'_{P^*}, and B must be disconnected from the vertices in V''_{P^*}.

Let $\mathcal{R}_{\hat{n},B} \subseteq \mathcal{R}_Z(\hat{n}, \top)$ be a set of paths such that $R' \in \mathcal{R}_{\hat{n},B}$ if and only if $E(R) \cup E(R')$ satisfies $(\#'_B)$ for arbitrarily chosen $R \in \mathcal{R}_Z(\hat{r}, \hat{n})$. $\mathcal{R}_{\hat{n},B}$ is well-defined, i.e., kept unchanged regardless of the choice of R because $E(R)$'s connectivity among components and the vertices in $V'_{P^*} \cup V''_{P^*}$ is completely determined in $\hat{n}.\mathsf{vset}$. This means that $\mathcal{R}_{v,\hat{n}}$ can be written as *direct product* $\mathcal{R}_Z(\hat{r}, \hat{n}) \sqcup \mathcal{R}_{\hat{n},\hat{n}.\mathsf{comp}[v]}$ where $A \sqcup B := \{a \cup b \mid a \in A, b \in B\}$. In other words, every path $R \in \mathcal{R}_{v,\hat{n}}$ can be decomposed into $R' \in \mathcal{R}_Z(\hat{r}, \hat{n})$ and $R'' \in \mathcal{R}_{\hat{n},\hat{n}.\mathsf{comp}[v]}$. From the definition of path product, $W_p(R) = W_p(R')W_p(R'')$. Thus, by defining $\hat{n}.\mathsf{q}[B]$ as the sum of path product of the paths in $\mathcal{R}_{\hat{n},B}$, the following holds:

$$\sum_{R \in \mathcal{R}_{v,\hat{n}}} W_p(R) = \sum_{R' \in \mathcal{R}_Z(\hat{r},\hat{n})} \sum_{R'' \in \mathcal{R}_{\hat{n},\hat{n}.\mathsf{comp}[v]}} W_p(R')W_p(R'') = \hat{n}.\mathsf{p} \cdot \hat{n}.\mathsf{q}[\hat{n}.\mathsf{comp}[v]]. \tag{5}$$

[2] If v's degree is more than 1, there is at least one i such that $v \in F_i$. For example, if e_i is the first edge containing v within the edge ordering, $v \in F_i$. The treatment of degree 1 vertices is in Appendix A.1.

Finally, $\mathsf{count}(v)$ can be represented as

$$\mathsf{count}(v) = \sum_{\hat{n} \in L_i} \sum_{R \in \mathcal{R}_{v,\hat{n}}} W_p(R) = \sum_{\hat{n} \in L_i} \hat{n}.\mathsf{p} \cdot \hat{n}.\mathsf{q}[\hat{n}.\mathsf{comp}[v]]. \tag{6}$$

By choosing i such that $v \in F_i$ for every v, we can compute $\mathsf{count}(v)$ for every v by (6) if p and q are computed. In the next section, we show that q can easily be computed by DP.

5.2 Dynamic Programming

First, we define a correspondence of the components in comp between $\hat{n}$ and its child nodes.

▶ **Definition 1.** *Let $\hat{n} \in L_i$ be a node of a normalized ZDD whose label is i, and let f be either $-$ or $+$. Assuming $\hat{n}^f \neq \bot$, for $B \in \hat{n}.\mathsf{comp}$, we define B^f as follows: (i) If B contains vertex v in F_{i+1}, $B^f = \hat{n}^f.\mathsf{comp}[v]$. (ii) If no such vertex exists, $B^- = \emptyset$, i.e., no corresponding component. For $f = +$, let v, v' be the endpoints of e_i. If $v' \in F_i$ and $v \in B$, $B^+ = \hat{n}^+.\mathsf{comp}[v']$. If $v \in F_i$ and $v' \in B$, $B^+ = \hat{n}^+.\mathsf{comp}[v]$. Otherwise, $B^+ = \emptyset$.*

Intuitively, B^f is a component in $\hat{n}^f.\mathsf{comp}$ that represents the same component as B.

We derive a formula for $\hat{n}.\mathsf{q}[B]$ by decomposing the set of paths $\mathcal{R}_{\hat{n},B}$. Since every path in $\mathcal{R}_{\hat{n},B}$ passes either $\mathsf{lo}(\hat{n})$ or $\mathsf{hi}(\hat{n})$, we have a case analysis. Let $\hat{n}.\mathsf{q}^-[B]$ ($\hat{n}.\mathsf{q}^+[B]$) be the sum of path products of the paths in $\mathcal{R}_{\hat{n},B}$ that traverse $\mathsf{lo}(\hat{n})$ ($\mathsf{hi}(\hat{n})$). Now

$$\hat{n}.\mathsf{q}[B] = \hat{n}.\mathsf{q}^-[B] + \hat{n}.\mathsf{q}^+[B]. \tag{7}$$

We now focus on $\hat{n}.\mathsf{q}^-[B]$, which means that e_i is excluded. If $\hat{n}^- = \bot$, constraint $\mathcal{C}(P^*[])$ is not satisfied, so no path in $\mathcal{R}_{\hat{n},B}$ passes through $\mathsf{lo}(\hat{n})$. Thus, $\hat{n}.\mathsf{q}^-[B] = 0$. Otherwise, B^- is defined as in Definition 1. If $B^- \neq \emptyset$, since B^- is the same component as B, constraint $(\#'_B)$ is satisfied if and only if constraint $(\#'_{B^-})$ for $\hat{n}^-$ is satisfied. Thus, the set of paths in $\mathcal{R}_{\hat{n},B}$ that traverse $\mathsf{lo}(\hat{n})$ can be written as $\{\mathsf{lo}(\hat{n})\} \sqcup \mathcal{R}_{\hat{n}^-,B^-}$. This means that $\hat{n}.\mathsf{q}^-[B] = w^-_{e_i} \cdot \hat{n}^-.\mathsf{q}[B^-]$.

The remaining case is $\hat{n}^- \neq \bot$ and $B^- = \emptyset$. In this case, the component B does not exist in the next level L_{i+1} and thus we cannot translate constraint $(\#'_B)$ into the one concerning the lo-child $\hat{n}^-$. In other words, we must judge whether constraint $(\#'_B)$ is satisfied with only the information on $\hat{n}$. Fortunately, it is possible because $\hat{n}.\mathsf{vset}$ completely determines the connectivity among $B \in \hat{n}.\mathsf{comp}$ and $V'_{P^*} \cup V''_{P^*}$.

We have case analysis on how B is connected with $V'_{P^*} \cup V''_{P^*}$; how to distinguish these cases are described later. If B is connected with some (but not all) vertices in V'_{P^*}, it violates the constraint $P^*[]$ that all the vertices in V'_{P^*} are connected. If B is connected with both the vertices in V'_{P^*} and those in V''_{P^*}, it again violates the constraint $P^*[]$ that the vertices in the different sets of $P^*[]$ are disconnected. Therefore, since at least $P^*[]$ is not violated by $R' \in \mathcal{R}_Z(\hat{r}, \hat{n})$, only one of the three cases must hold: (i) B is disconnected from any vertex in P^*, (ii) B is connected with all the vertices in V'_{P^*}, and (iii) B is connected with some vertices in V''_{P^*}. When $V'_{P^*} \neq \emptyset$, only case (ii) satisfies $(\#'_B)$. When $V'_{P^*} = \emptyset$, case (i) also satisfies $(\#'_B)$. For both scenarios, the set of paths in $\mathcal{R}_{\hat{n},B}$ that traverse $\mathsf{lo}(\hat{n})$ can be written as $\{\mathsf{lo}(\hat{n})\} \sqcup \mathcal{R}_Z(\hat{n}^-, \top)$, since $(\#'_B)$ is always satisfied regardless of the choice of the path from $\hat{n}^-$ to $\top$. By defining $\hat{n}'.\mathsf{r}$ as the sum of path products of the paths in $\mathcal{R}_Z(\hat{n}, \top)$ for any node $\hat{n}'$, $\hat{n}.\mathsf{q}^-[B] = w^-_{e_i} \cdot \hat{n}^-.\mathsf{r}$ for these cases. To sum up, the following holds:

Algorithm 2 CompDP: dynamic programming with information of comp.

```
1  r̂.p ← 1, set all other p values to 0, ⊤.r ← 1, ⊥.r ← 0, set all q values to 0
2  for i ← 1 to m do                                                    // Top-down DP
3    foreach n̂ ∈ L_i do
4      if n̂⁻ ≠ ⊥ then n̂⁻.p += w⁻_{e_i} · n̂.p                               // (4)
5      if n̂⁺ ≠ ⊥ then n̂⁺.p += w⁺_{e_i} · n̂.p
6  for i ← m to 1 do                                                    // Bottom-up DP
7    foreach n̂ ∈ L_i do
8      n̂.r ← w⁻_{e_i} · n̂⁻.r + w⁺_{e_i} · n̂⁺.r                             // (9)
9      foreach B ∈ n̂.comp do
10       foreach f ∈ {−, +} do
11         if n̂^f ≠ ⊥ and B^f ≠ ∅ then n̂.q^f[B] ← w^f_{e_i} · n̂^f.q[B^f]     // (8), 1st case
12         else if n̂^f ≠ ⊥ and (B ∩ V'_{n̂.vset} ≠ ∅ or (V'_{P*} = ∅ and B ∩ V''_{n̂.vset} = ∅)) then
13           n̂.q^f[B] ← w^f_{e_i} · n̂^f.r                                    // (8), 2nd case
14     Process corner cases
```

$$
\hat{n}.\mathsf{q}^-[B] = \begin{cases} w^-_{e_{\mathsf{lb}(\hat{n})}} \cdot \hat{n}^-.\mathsf{q}[B^-] & (\hat{n}^- \neq \bot,\ B^- \neq \emptyset) \\ w^-_{e_{\mathsf{lb}(\hat{n})}} \cdot \hat{n}^-.\mathsf{r} & (\hat{n}^- \neq \bot,\ B^- = \emptyset, \text{ case (ii) or (case (i) and } V'_{P^*} = \emptyset)) \\ 0 & (\text{otherwise}) \end{cases}, \tag{8}
$$

$$
\text{where} \quad \top.\mathsf{r} = 1,\ \bot.\mathsf{r} = 0,\ \hat{n}.\mathsf{r} = w^-_{e_{\mathsf{lb}(\hat{n})}} \cdot \hat{n}^-.\mathsf{r} + w^+_{e_{\mathsf{lb}(\hat{n})}} \cdot \hat{n}^+.\mathsf{r}. \tag{9}
$$

Note that the recurrence formula (9) for r can easily be derived from the definition in the same way as the formula (4) for p.

The remaining is how to distinguish the cases (i)–(iii). Since the connectivity among B and the vertices in P^* is stored in $\hat{n}$.vset, it can be achieved by the comparison of B and $\hat{n}$.vset. Let $V'_{\hat{n}.\mathsf{vset}}$ be the vertices in the set in $\hat{n}$.vset containing $*$ and let $V''_{\hat{n}.\mathsf{vset}}$ be the other vertices present in $\hat{n}$.vset. Then, case (i) holds when the vertices in B do not exist in $\hat{n}$.vset. Case (ii) holds when B has a node in $V'_{\hat{n}.\mathsf{vset}}$. Case (iii) holds when B has a node in $V''_{\hat{n}.\mathsf{vset}}$. Let us see the example by Fig. 2(d). If we perform FBS with $P^* = \{\{1, 4, *\}\}$, the center node of 5th level, say $\hat{n}$, becomes $\{3\}\{4\}/\{4, *\}$. When traversing $\mathsf{lo}(\hat{n})$, both $\{3\}$ and $\{4\}$ leave from the frontier. Here $\{3\}$ falls into case (i) and $\{4\}$ falls into case (ii), thus we have $\hat{n}.\mathsf{q}^-[\{3\}] = 0$ and $\hat{n}.\mathsf{q}^-[\{4\}] = w^-_{e_5} \cdot \hat{n}^-.\mathsf{r} = w^-_{e_5} \cdot \top.\mathsf{r} = w^-_{e_5}$.

Equation (8) also holds even if we substitute $-$ with $+$, which means e_i is included, except for the following corner case. Let $e_i = \{v, v'\}$. We consider the case where $v, v' \notin F_{i+1}$ and $\hat{n}.\mathsf{comp}[v]$ leaves the frontier with case (i). When $V'_{P^*} \neq \emptyset$, if $v' \in V'_{\hat{n}.\mathsf{vset}}$ or $\hat{n}.\mathsf{comp}[v']$ leaves the frontier with case (ii), $\hat{n}.\mathsf{q}^+[\hat{n}.\mathsf{comp}[v]] = w^+_{e_i} \cdot \hat{n}^+.\mathsf{r}$ since $\hat{n}.\mathsf{comp}[v]$ is finally connected with V'_{P^*}. Similarly, when $V'_{P^*} = \emptyset$, if $v' \in V''_{\hat{n}.\mathsf{vset}}$ or $\hat{n}.\mathsf{comp}[v']$ leaves the frontier with case (iii), $\hat{n}.\mathsf{q}^+[\hat{n}.\mathsf{comp}[v]] = 0$ since $\hat{n}.\mathsf{comp}[v]$ is finally connected with V''_{P^*}.

Algorithm 2 describes the pseudocode for DP. After p, q and r values are computed by Algorithm 2, the $\mathsf{count}(v)$ value for every v can be obtained by (6).

5.3 Intersection with Base Set

Next we generalize for case $\mathcal{F} \neq 2^E$. In the previous sections, we used the fact that $P^*[v]$ is a constraint made by adding another constraint $(\#_v)$ to $P^*[]$. This also holds even if $\mathcal{F}$ is constrained, i.e., $\mathcal{F} \cap \mathcal{C}(P^*[v]) = \{X \mid X \in \mathcal{F} \cap \mathcal{C}(P^*[]),\ X \text{ satisfies } (\#_v)\}$. Therefore, by constructing a ZDD Z representing $\mathcal{F} \cap \mathcal{C}(P^*[])$ with the information of comp and vset, we can reuse the discussions in Sections 5.1 and 5.2 and run Algorithm 2 on Z to obtain $\mathsf{count}(v)$

for every $v \in V$. More specifically, let Z' be the normalized ZDD of $\mathcal{C}(P^*[])$ built by FBS with P^*. Then Z should be a normalized ZDD representing $\mathcal{F} \cap \mathcal{C}(P^*[])$ that satisfies the following condition for every i: for any i-th subgraph $X \subseteq E_{<i}$, if X corresponds to i-th level nodes $\hat{n}$ in Z and $\hat{n}'$ in Z', $\hat{n}$ must have the same comp and vset as $\hat{n}'$.

After building $Z_\mathcal{F}$ that represents $\mathcal{F}$ by some means, Z can be built by combining FBS with the existing methods. One approach is to use Apply [20]. First, we build Z' representing $\mathcal{C}(P^*[])$ by FBS. Then by taking the set intersection of $Z_\mathcal{F}$ and Z' with Apply while keeping the information of comp and vset, we can construct Z. The other is to use subsetting [15]. It enables us to directly construct Z from $Z_\mathcal{F}$ in a similar manner as the FBS. For the sake of completeness, we describe the pseudocode of subsetting with FBS in Appendix A.2.

6 Complexity Analysis

We here conduct a complexity analysis of the proposed algorithm. For connectivity constraint P possibly including $*$, let $Z_{\mathrm{FBS}(P)}$ be a ZDD built by FBS with P, let c_P be the number of sets in P, and let v_P be the number of vertices in P (excluding $*$). Additionally, let $\mathsf{fw} = \max_i |F_i|$ called *frontier width.* We give detailed proofs in Appendix A.3.

First, we bound the running time of our algorithm by the ZDD size.

▶ **Proposition 2.** *Our proposed algorithm runs in $O(\mathsf{fw} \cdot |Z_\mathcal{F}||Z_{\mathrm{FBS}(P^*)}|)$ time.*

Next we bound the ZDD sizes. The bound of $|Z_{\mathrm{FBS}(P)}|$ for P excluding $*$ is given in Proposition 3 and that of $|Z_{\mathrm{FBS}(P^*)}|$ for P^* including $*$ is in Proposition 4.

▶ **Proposition 3.** *The size of $Z_{\mathrm{FBS}(P)}$ for connectivity constraint P excluding $*$ is bounded by $O(mD_\mathsf{fw} \cdot \min\{(c_P+1)^\mathsf{fw}, (\mathsf{fw}+1)^{v_P}\})$, where D_fw is the fw-th Bell number.*

▶ **Proposition 4.** *For connectivity constraint P^* including $*$, $|Z_{\mathrm{FBS}(P^*)}| \le c_{P^*}|Z_{\mathrm{FBS}(P^*[])}|$.*

Combining Propositions 2–4 yields the following theorem.

▶ **Theorem 5.** *The proposed algorithm runs in $O(\mathsf{fw} \cdot c_{P^*}|Z_\mathcal{F}||Z_{\mathrm{FBS}(P^*[])}|)$ time, which is bounded by $O(|Z_\mathcal{F}| \cdot mc_{P^*} \cdot \mathsf{fw} \cdot D_\mathsf{fw} \cdot \min\{(c_{P^*}+1)^\mathsf{fw}, (\mathsf{fw}+1)^{v_{P^*}}\})$.*

If fw can be considered as a constant, the proposed algorithm runs in $O(mc_{P^*}|Z_\mathcal{F}|)$ time. It is known that fw is closely related to the *pathwidth* [24] of a graph. If a graph's pathwidth is pw, there is a edge ordering with $\mathsf{fw} = \mathsf{pw}$ [13]. The value of pw is often much smaller than n and m for sparse graphs, e.g., [22]. Although obtaining such an order is NP-hard in general, we can use pathwidth optimization heuristics [13] for obtaining better ordering.

We compare this complexity with the baseline method where we separately build a ZDD representing $\mathcal{F} \cap \mathcal{C}(P^*[v])$ by FBS. The overall complexity is $O(\mathsf{fw} \cdot \sum_v |Z_\mathcal{F}||Z_{\mathrm{FBS}(P^*[v])}|)$, analyzed in the same way as Proposition 2, which is bounded by $O(|Z_\mathcal{F}| \cdot mn \cdot \mathsf{fw} \cdot D_\mathsf{fw} \cdot \min\{(c_{P^*}+1)^\mathsf{fw}, (\mathsf{fw}+1)^{v_{P^*}+1}\})$. If fw is constant, it is $O(|Z_\mathcal{F}|mn)$. Compared with Theorem 5, the proposed method runs faster by an $O(n)$ factor.

Here we mention the ZDD sizes. The complexity bounds of the proposed and baseline methods heavily depend on $Z_\mathcal{F}$'s size. Here $|Z_\mathcal{F}|$ also remains small for various constraints if fw is small. For example, the constraints appeared in the example of Section 2, e.g., the degree constraints and the existence of cycles, can all be represented as a ZDD whose size is proportional to m if fw is constant [27, 18, 17]. This boosts the effectiveness of both the proposed and baseline methods for practical use because fw is often much smaller than n and m for graphs in real worlds. Moreover, $|Z|$ is often much smaller than expected from the above analysis, as demonstrated by Kawahara et al. [17].

Table 2 Computational time for grid graphs and Rocketfuel dataset in seconds.

Instance	n	m	fw	Path Ours	Path Base	Cycle Ours	Cycle Base	Steiner tree Ours	Steiner tree Base	RSF Ours	RSF Base
Grid-8x8	64	112	8	**0.06**	0.48	**0.06**	0.54	**4.93**	29.87	**8.87**	38.17
Grid-8x16	128	232	8	**0.16**	2.54	**0.16**	2.81	**14.16**	183.16	**25.72**	234.17
Grid-8x24	192	352	8	**0.26**	6.29	**0.27**	6.86	**23.27**	470.60	**42.81**	>600
Grid-8x32	256	472	8	**0.36**	11.73	**0.38**	12.65	**32.55**	>600	**60.28**	>600
Grid-9x9	81	144	9	**0.22**	2.13	**0.22**	2.34	**40.13**	298.57	**62.44**	397.57
Grid-10x10	100	180	10	**0.78**	9.70	**0.89**	11.69	**284.17**	>600	**430.04**	>600
Grid-11x11	121	220	11	**2.88**	42.26	**3.22**	50.97	>600	>600	>600	>600
Grid-12x12	144	264	12	**11.24**	183.87	**15.25**	241.94	>600	>600	>600	>600
Grid-13x13	169	312	13	**45.51**	>600	**56.25**	>600	>600	>600	>600	>600
Rocketfuel-1221	318	758	10	**157.01**	>600	**111.52**	>600	**181.38**	>600	>600	>600
Rocketfuel-1755	172	381	12	**43.94**	>600	**32.48**	>600	>600	>600	>600	>600
Rocketfuel-6461	182	294	10	**3.66**	65.64	**4.80**	128.86	**71.10**	>600	**95.47**	>600

We close this section by mentioning the space complexity of the proposed and the baseline methods. The proposed algorithm uses at most $O(\mathsf{fw} \cdot |Z_{\mathcal{F}}||Z_{\mathrm{FBS}(P^*)}|)$ words of space, since it retains $O(\mathsf{fw})$ words of information for each node of $Z_{\mathrm{FBS}(P^*)}$. The baseline method typically uses at most $O(\mathsf{fw} \cdot |Z_{\mathcal{F}}||Z_{\mathrm{FBS}(P^*[v])}|)$ words of space for the computation of $\mathsf{count}(v)$. If it is assumed that $|Z_{\mathrm{FBS}(P^*[v])}|$ is close to $|Z_{\mathrm{FBS}(P^*[])}|$, the space complexity of the baseline method is at most only $O(c_{P^*})$ times smaller due to Proposition 4.

7 Experiments

We empirically compared the proposed and the baseline methods with respect to the computational time. Here the baseline method is to separately build a ZDD by FBS for each constraint. Both methods were implemented in C++ and compiled by g++ with -O3 option. We used TdZdd [14] for the baseline method, which is a highly optimized C++ library for FBS. We also used TdZdd for the proposed method to construct ZDD $Z_{\mathcal{F}}$ of base set. Experiments are conducted on a single thread of a Linux machine with AMD EPYC 7763 2.45 GHz CPU and 2048 GB RAM; note that we used less than 256 GB of memory during the experiments. We set the time limit of every run to 600 seconds.

We used both synthetic graphs and real benchmarks as tested graphs. The synthetic ones are grid graphs; Grid-wxh represents a grid graph with $w \times h$ vertices. For the others, we used the Rocketfuel [29] and Romegraph datasets [7]. Rocketfuel was also used in [22]. From Romegraph, we chose all the graphs with $n = 100$: there were 140 such graphs. Identical edge ordering was used for both methods, and it was decided as follows: For the grid graphs, we used the edge ordering of Iwashita et al. [16], which is better for the DP on grid graphs. For the other graphs, we used beam-search heuristics [13] to determine the edge ordering.

We evaluated four problem settings in Section 2: path, cycle, Steiner tree, and rooted spanning forest (RSF). The given vertices for these settings were determined as follows. Let $d(v, v')$ be the shortest distance between vertices v and v'. For the path problem, we chose the most distant vertex pair as s, t, i.e., s, t satisfies $d(s, t) = \max_{v,v'} d(v, v')$. For the cycle problem, we chose the graph center as s, i.e., $s \in \operatorname{argmin}_v \max_{v'} d(v, v')$. For the other problems, we chose four vertices as T such that the sum of the distances between distinct vertices, $\sum_{v,v' \in T: v \neq v'} d(v, v')$, is maximized.

Table 2 shows the result for the grid graphs and the Rocketfuel dataset. For all the graphs and problem settings solved by both methods within the time limit, the proposed method ran about 10–20 times faster than the baseline method. The complexity analyses in Section 6

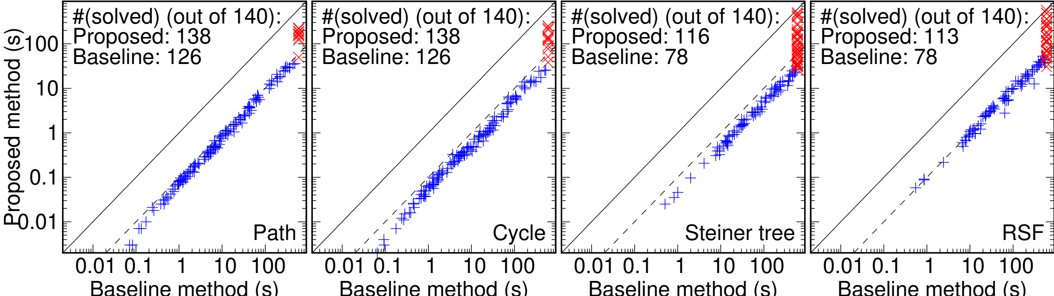


Figure 3 Computational time for Romegraph dataset: Blue points indicate instances solved by both methods, and red points indicate those solved only by proposed method. Solid black lines indicate elapsed time for both methods is identical, and dashed lines indicate proposed method is 10 times faster than baseline method.

suggest that the proposed method becomes faster than the existing method when n is large. Table 2 exhibits such a tendency. For example, for the Grid-8xh graphs, the proposed method becomes much faster than the baseline method when $n = 8h$ is increased. In addition, both methods ran faster for graphs with smaller fw value, reflecting the complexity analyses.

Fig. 3 plots the result for the Romegraph dataset and also describes the number of graphs solved by each method within the time limit. Here each point corresponds to a graph, where the blue ones are those solved by both methods and the red ones are those solved only by the proposed method. Although the computational time itself varied from less than 0.01 to 600 seconds, for almost all the graphs the proposed method ran about 10–20 times faster than the existing method. This ratio is kept because the graphs all have the same number of vertices: 100. We give detailed results for Romegraph in Appendix A.4.

8 Conclusion

We proposed a novel framework, compDP, for solving multiple subgraph counting problems with similar connectivity constraints simultaneously. A complexity analysis showed that the proposed method ran $O(n)$ times faster than the baseline approach, and the experiments revealed the proposed method's efficiency.

As a future work, we will consider dealing with the reachability in directed graphs. There are approaches for building BDDs of reachability constraints [19, 30], and we want to consider whether they can be incorporated into our framework.

References

1 N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.

2 Eric T. Bax. Algorithms to count paths and cycles. *Information Processing Letters*, 52(5):249–252, 1994.

3 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, 1986.

4 Radu Curticapean. Counting problems in parameterized complexity. In *Proc. of the 13th International Symposium on Parameterized and Exact Computation (IPEC)*, pages 1:1–1:18, 2018.

5 Jörg Flum and Martin Grohe. The parameterized complexity of counting problems. *SIAM J. Comput.*, 33(4):892–922, 2004.

6 Luigi Fratta and Ugo Montanari. A Boolean algebra method for computing the terminal reliability in a communication network. *IEEE Trans. Circuit Theory*, 20(3):203–211, 1973.

7 graphdrawing.org. RomeGraph. `http://www.graphdrawing.org/data.html`.

8 G. Hardy, C. Lucet, and N. Limnios. K-terminal network reliability measures with binary decision diagrams. *IEEE Trans. Rel.*, 56(3):506–515, 2007.

9 J.U. Herrmann. Improving reliability calculation with augmented binary decision diagrams. In *Proc. of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 328–333, 2010.

10 Makoto Imase and Bernard M. Waxman. Dynamic Steiner tree problem. *SIAM J. Discrete. Math.*, 4(3):369–384, 1991.

11 Takeru Inoue. Reliability analysis for disjoint paths. *IEEE Trans. Rel.*, 68(3):985–998, 2018.

12 Takeru Inoue, Norihito Yasuda, Shunsuke Kawano, Yuji Takenobu, Shin-ichi Minato, and Yasuhiro Hayashi. Distribution network verification for secure restoration by enumerating all critical failures. *IEEE Trans. Smart Grid*, 6(2):843–852, 2015.

13 Yuma Inoue and Shin-ichi Minato. Acceleration of ZDD construction for subgraph enumeration via pathwidth optimization. Technical Report TCS-TR-A-16-80, Division of Computer Science, Hokkaido University, 2016.

14 Hiroaki Iwashita, Jun Kawahara, and Kohei Shinohara. TdZdd. `https://github.com/kunisura/TdZdd`.

15 Hiroaki Iwashita and Shin-ichi Minato. Efficient top-down ZDD construction techniques using recursive specifications. Technical Report TCS-TR-A-13-69, Division of Computer Science, Hokkaido University, 2013.

16 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.

17 Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato. Frontier-based search for enumerating all constrained subgraphs with compressed representation. *IEICE Trans. Fundamentals*, E100-A(9):1773–1784, 2017.

18 Donald E. Knuth. *The art of computer programming: Vol. 4A. Combinatorial Algorithms, Part 1.* Addison-Wesley Professional, 2011.

19 Takanori Maehara, Hirofumi Suzuki, and Masakazu Ishihata. Exact computation of influence spread by binary decision diagrams. In *Proc. of the 26th International World Wide Web Conference (WWW)*, pages 947–956, 2017.

20 Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of the 30th ACM/IEEE Design Automation Conference (DAC)*, pages 272–277, 1993.

21 Fred Moskowitz. The analysis of redundancy networks. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 77(5):627–632, 1958.

22 Kengo Nakamura, Takeru Inoue, Masaaki Nishino, and Norihito Yasuda. Efficient network reliability evaluation for client-server model. In *Proc. of IEEE Global Communication Conference (GLOBECOM)*, pages 1–6, 2021.

23 R. C. Read and R. E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975.

24 Neil Robertson and P.D. Seymour. Graph minors. I. excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983.

25 Arnie Rosenthal. Computing the reliability of complex networks. *SIAM J. Appl. Math.*, 32(2):384–393, 1977.

26 Shinsaku Sakaue and Kengo Nakamura. Differentiable equilibrium computation with decision diagrams for Stackelberg models of combinatorial congestion games. In *Proc. of the 35th Conference on Neural Information Processing Systems (NeurIPS)*, pages 9416–9428, 2021.

27 Kyoko Sekine, Hiroshi Imai, and Seiichiro Tani. Computing the Tutte polynomial of a graph of moderate size. In *Proc. of the 6th International Symposium on Algorithms and Computation (ISAAC)*, pages 224–233, 1995.

28 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.

29 Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring ISP topologies with Rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, 2004.

30 Hirofumi Suzuki, Masakazu Ishihata, and Shin-ichi Minato. Exact computation of strongly connected reliability by binary decision diagrams. In *Proc. of the 12th Annual International Conference on Combinatorial Optimization and Applications (COCOA)*, pages 281–295, 2018.

31 S. Tsukiyama, I. Shirakawa, H. Ozaki, and H. Ariyoshi. An algorithm to enumerate all cutsets of a graph in linear time per cutset. *J. ACM*, 27(4):619–632, 1980.

32 Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

33 R. Kevin Wood. Factoring algorithms for computing K-network network reliability. *IEEE Trans. Rel.*, 35(3):269–278, 1986.

A Appendix

A.1 Treatment of Degree 1 Vertices

As in Section 5.1, we can compute the $\mathsf{count}(v)$ value by focusing on the i-th level of Z where $v \in F_i$. However, if v's degree is 1, no such i exists. Let $e_i = \{v, v'\}$ be the only edge incident to v. Then for $i' < i$, v is not present in $E_{\geq i'}$, and for $i' \geq i$, it is not present in $E_{<i'}$, so no frontier contains v. Therefore, we have an alternative formula for computing $\mathsf{count}(v)$ like Eq. (6). Let us focus on the i-th level where $e_i = \{v, v'\}$ is the only edge incident to v.

First, we address the case where the other endpoint, v', is in F_i. Let $\hat{n} \in L_i$ be an arbitrarily chosen i-th level node of Z. Since e_i is the only edge incident to v, if e_i is excluded, v remains as an isolated vertex. Thus, if $V'_{P^*} \neq \emptyset$, constraint $(\#_v)$ will never be satisfied. Otherwise, if $V'_{P^*} = \emptyset$, constraint $(\#_v)$ is always satisfied. In this case, the set of paths in $\mathcal{R}_Z(\hat{r}, \top)$ that passes $\hat{n}$ whose corresponding subgraph satisfies $(\#_v)$, i.e., $\mathcal{R}_{v,\hat{n}}$, can be written as $\mathcal{R}_Z(\hat{r}, \hat{n}) \sqcup \{\mathsf{lo}(\hat{n})\} \sqcup \mathcal{R}_Z(\hat{n}^-, \top)$. The sum of their path products is $\hat{n}.\mathsf{p} \cdot w^-_{e_i} \cdot \hat{n}^-.\mathsf{r}$ given that $\hat{n}^- \neq \bot$. If e_i is included, v is connected with v', and so condition $(\#_v)$ is met if and only if condition $(\#'_B)$ for $B = \hat{n}.\mathsf{comp}[v']$ is met. In this case, the set of paths in $\mathcal{R}_Z(\hat{r}, \top)$ that passes $\hat{n}$ whose corresponding subgraph satisfies $(\#_v)$ can be written as $\mathcal{R}_Z(\hat{r}, \hat{n}) \sqcup \mathcal{R}^+_{\hat{n},\hat{n}.\mathsf{comp}[v']}$, where $\mathcal{R}^+_{\hat{n},B}$ is the set of paths in $\mathcal{R}_{\hat{n},B}$ that passes through $\mathsf{hi}(\hat{n})$. The sum of their path products is $\hat{n}.\mathsf{p} \cdot \hat{n}.\mathsf{q}^+[\hat{n}.\mathsf{comp}[v']]$ using the notion q^+ introduced in Section 5.2. The value $\mathsf{count}(v)$ can be computed by their sum over $\hat{n} \in L_i$:

$$\mathsf{count}(v) = \sum_{\hat{n} \in L_i} \hat{n}.\mathsf{p} \cdot \hat{n}.\mathsf{q}^+[\hat{n}.\mathsf{comp}[v']] + \begin{cases} 0 & (V'_{P^*} \neq \emptyset) \\ \sum_{\hat{n} \in L_i : \hat{n}^- \neq \bot} \hat{n}.\mathsf{p} \cdot w^-_{e_i} \cdot \hat{n}^-.\mathsf{r} & (V'_{P^*} = \emptyset) \end{cases}. \quad (10)$$

The remaining issue is how to cope with case $v' \notin F_i$. For it, we can assume $v' \in F_{i+1}$; otherwise, v' is also a degree 1 vertex that means graph G consists of only e_i since G is connected, which is trivial. The case where e_i is excluded is treated in the same way as above. If e_i is included, let $\hat{n}$ be an arbitrary i-th level node of Z. Since v is connected with v', constraint $(\#_v)$ is met if and only if constraint $(\#'_B)$ for $B = \hat{n}^-.\mathsf{comp}[v']$ is met. Therefore, the set of paths in $\mathcal{R}_Z(\hat{r}, \top)$ that passes $\hat{n}$ whose corresponding subgraph satisfies $(\#_v)$ can be written as $\mathcal{R}_Z(\hat{r}, \hat{n}) \sqcup \{\mathsf{hi}(\hat{n})\} \sqcup \mathcal{R}_{\hat{n}^+,\hat{n}^+.\mathsf{comp}[v']}$, given that $\hat{n}^+ \neq \bot$. The sum of their path products is $\hat{n}.\mathsf{p} \cdot w^+_{e_i} \cdot \hat{n}^+.\mathsf{q}[\hat{n}^+.\mathsf{comp}[v']]$. The value $\mathsf{count}(v)$ can be computed by

Algorithm 3 Frontier-based search with subsetting for connectivity constraint P and base ZDD $Z_{\mathcal{F}}$ representing the base family of subgraphs $\mathcal{F}$.

```
1  r̂.comp ← {}, r̂.vset ← P, r̂.base ← r̂_F (Z_F's root), L_1 ← {r̂}, L_i ← ∅ (i = 2,...,m+1)
2  for i ← 1 to m do                                                  // e_i = {v, v'}
3    foreach n̂ ∈ L_i do
4      foreach f ∈ {−, +} do
5        n̂' ← n̂
6        if f = + and i < lb(n̂'.base) then
7          n̂' ← ⊥ and goto finish                        // No further subgraphs in F
8        if i = lb(n̂'.base) then
9          n̂'.base ← (n̂'.base)^f                         // base proceeds to child node
10         if n̂'.base = ⊥ then
11           n̂' ← ⊥ and goto finish                      // No further subgraphs in F
12       if f = + and v, v' ∈ V_{n̂'.vset} and n̂'.vset[v] ≠ n̂'.vset[v'] then
13         n̂' ← ⊥ and goto finish                        // v and v' must not be connected
14       foreach u ∈ {v, v'} \ F_i do                     // Vertices entering the frontier
15         n̂'.comp ← n̂'.comp ∪ {{u}}                     // Add u as an isolated vertex
16       if f = + and n̂'.comp[v] ≠ n̂'.comp[v'] then      // Connecting two components
17         Merge n̂'.comp[v] and n̂'.comp[v'] into one
18         if v ∈ V_{n̂'.vset} and v' ∉ V_{n̂'.vset} then Add vertices in n̂'.comp[v'] to n̂'.vset[v]
19         if v ∉ V_{n̂'.vset} and v' ∈ V_{n̂'.vset} then Add vertices in n̂'.comp[v] to n̂'.vset[v']
20       foreach u ∈ {v, v'} \ F_{i+1} do                 // Vertices leaving the frontier
21         if {u} ∈ n̂'.comp then                  // Component containing u leaves frontier
22           if u ∈ V_{n̂'.vset} and {u} ∉ n̂'.vset and {u, *} ∉ n̂'.vset then
23             n̂' ← ⊥ and goto finish             // u must be connected to w ∈ n̂'.vset[u]
24         Remove u from n̂'.comp and n̂'.vset if exists
25       if n̂' ≠ ⊥ then
26         if i = m then n̂' ← ⊤                                 // All conditions are meet
27         else if ∃n̂'' ∈ L_{i+1} s.t. n̂''.comp = n̂'.comp and n̂''.vset = n̂'.vset and
             n̂''.base = n̂'.base then
28           n̂' ← n̂''                                           // Already generated node
29         else L_{i+1} ← L_{i+1} ∪ {n̂'}                        // Newly generated node
30       finish:
31       n̂^f ← n̂'                                        // Set lo- or hi-child of n̂ to n̂'
```

$$\mathsf{count}(v) = \sum_{\hat{n} \in L_i : \hat{n}^+ \neq \bot} \hat{n}.\mathsf{p} \cdot w^+_{e_i} \cdot \hat{n}^+.\mathsf{q}[\hat{n}^+.\mathsf{comp}[v']] + \begin{cases} 0 & (V'_{P^*} \neq \emptyset) \\ \sum_{\hat{n} \in L_i : \hat{n}^- \neq \bot} \hat{n}.\mathsf{p} \cdot w^-_{e_i} \cdot \hat{n}^-.\mathsf{r} & (V'_{P^*} = \emptyset) \end{cases}. \quad (11)$$

A.2 Pseudocode for FBS with Subsetting

Next we explain the subsetting [15], which is used for taking the set intersection of the ZDDs, and describe the pseudocode for the FBS with subsetting. Given ZDD $Z_{\mathcal{F}}$ that represents the base set $\mathcal{F}$ and connectivity constraint P, it constructs a ZDD that represents $\mathcal{F} \cap \mathcal{C}(P)$. Starting with $\hat{r}_{\mathcal{F}}$ where $\hat{r}_{\mathcal{F}}$ is the root node of $Z_{\mathcal{F}}$, we traverse the lo-arc of $Z_{\mathcal{F}}$ if e_i is excluded in the FBS and its hi-arc of if e_i is included in the FBS. We now record the present node in $Z_{\mathcal{F}}$ as $\hat{n}$.base for each node $\hat{n}$. If it reaches $\bot$ in $Z_{\mathcal{F}}$, there are no further subgraphs in $\mathcal{F}$, and so pruning is executed. Here we can identify the two nodes of Z if their base as well as comp and vset are identical.

Based on these procedures, the FBS with subsetting can be described as Algorithm 3. Here the red part is newly added elements that are not included in Algorithm 1. Note that these codes are a bit complicated than the above explanation since we also cope with the case where $Z_{\mathcal{F}}$ is not normalized.

A.3 Proof of Propositions in Complexity Analysis

Proof of Proposition 2. By storing comp and vset as an integer sequence whose length is fw, as in the previous work [22], FBS with an intersection runs in $O(\mathsf{fw} \cdot |Z|)$ time where Z is the resultant ZDD. Since $|Z|$ can be bounded by the product of $|Z_{\mathcal{F}}|$ and $|Z_{\mathrm{FBS}(P)}|$ [20], $|Z| = O(|Z_{\mathcal{F}}||Z_{\mathrm{FBS}(P^*)}|)$ in the proposed algorithm. For each node, p, r, and $\mathsf{q}[B]$ can be computed in constant time, and there are at most fw sets in comp. Thus, the DP computation is completed in $O(\mathsf{fw} \cdot |Z|)$ time. The complete computation of $\mathsf{count}(v)$ can be done in $O(|Z|)$ time; by choosing i such that e_i is the first edge containing v for every v, each level of Z is scanned at most twice for computing $\mathsf{count}(v)$ for every v. ◀

Proof of Proposition 3. We consider the number of possible patterns for the comp and vset pair. We focus on an i-th level. Since comp is simply a partition of F_i, the number of possible patterns for it is $D_{|F_i|}$. The number of possible patterns for vset can be bounded in two ways. First, vset retains the information of how the components in vset are connected to each set in P. Each component of comp is connected to at most one set in P, since if more than two sets are connected, connectivity constraint P is violated. Since there are at most $|F_i|$ components, the number of vset patterns is bounded by $(c_P + 1)^{|F_i|}$, where $+1$ deals with the case where no component is connected to any sets in P. Second, vset can be seen as retaining the information of how the vertices in P are connected to the component in comp. Thus, the number of vset patterns is bounded by $(|F_i| + 1)^{v_P}$, where $+1$ deals with the case where a vertex in P is not connected to any component in comp. To sum up, the number of patterns of the comp and vset pair can be bounded by $O(D_{|F_i|} \cdot \min\{(c_P + 1)^{|F_i|}, (|F_i| + 1)^{v_P}\})$.

Since there are m levels and $|F_i| \leq \mathsf{fw}$, the overall size is bounded by $O(mD_{\mathsf{fw}} \cdot \min\{(c_P + 1)^{\mathsf{fw}}, (\mathsf{fw} + 1)^{v_P}\})$. ◀

Proof of Proposition 4. Since running Algorithm 1 with P^* and $P^*[]$ yields the same representing family of sets, $\mathcal{C}(P^*[])$, we only have to address the number of patterns of comp and vset. The only difference is that when running FBS with P^*, we must determine which set in vset has $*$. Since there are at most c_{P^*} sets in vset, there will be at most c_{P^*} patterns of vset for a node in $Z_{\mathrm{FBS}(P^*[])}$ when running FBS with P^*. Thus, $|Z_{\mathrm{FBS}(P^*)}| \leq c_{P^*}|Z_{\mathrm{FBS}(P^*[])}|$ holds. ◀

A.4 Detailed Experimental Results for Romegraph Dataset

Next we describe a detailed experimental results for the Romegraph dataset, which has 140 graphs whose number of nodes is exactly 100. With beam-search heuristics [13], the frontier width fw of each graph ranges from 6 to 14. The number of graphs per value of fw is described in Table 3.

Table 3 also shows the number of graphs solved within the time limit for each method, each problem setting, and each frontier width value. In addition, Fig. 4 plots the computational time for the Romegraph dataset aggregated by frontier width fw. For both methods, the graphs with a larger fw value are clearly difficult to solve, i.e., time-consuming; this outcome reflects the complexity results in Section 6. However, our proposed method can also treat graphs with a larger fw value than the baseline method. This again clearly indicates the efficiency of our proposed method.

Table 3 Comparison of the number of solved graphs in Romegraph dataset within time limit for each frontier width.

fw	#(graphs)	Path		Cycle		Steiner tree		RSF	
		Ours	Base	Ours	Base	Ours	Base	Ours	Base
6	3	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**
7	7	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**
8	26	**26**	**26**	**26**	**26**	**26**	**26**	**26**	**26**
9	28	**28**	**28**	**28**	**28**	**28**	27	**28**	**28**
10	33	**33**	**33**	**33**	**33**	**33**	15	**33**	14
11	25	**25**	24	**25**	24	**18**	0	**15**	0
12	10	**10**	5	**10**	5	**1**	0	**1**	0
13	6	**6**	0	**6**	0	0	0	0	0
14	2	0	0	0	0	0	0	0	0
Total	140	**138**	126	**138**	126	**116**	78	**113**	78

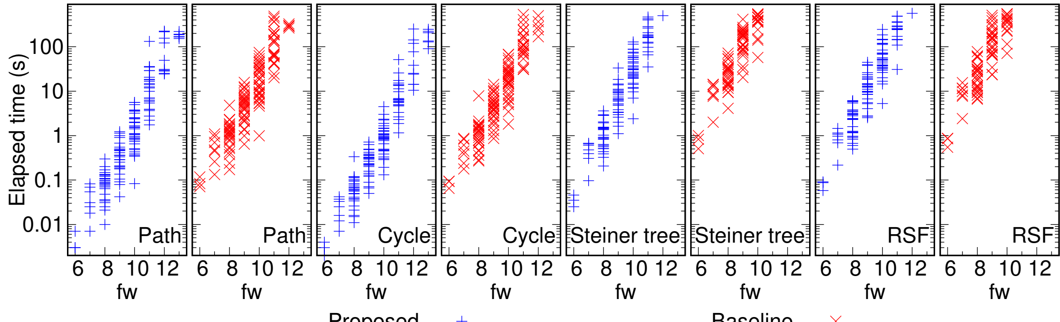


Figure 4 Computational time for Romegraph dataset aggregated by frontier width fw. Blue points indicate results of proposed method, and red points indicate results of baseline method.

Multilinear Formulations for Computing a Nash Equilibrium of Multi-Player Games

Miriam Fischer
Department of Computing, Imperial College London, UK

Akshay Gupte[1]
School of Mathematics, The University of Edinburgh, UK

Abstract

We present multilinear and mixed-integer multilinear programs to find a Nash equilibrium in multi-player noncooperative games. We compare the formulations to common algorithms in Gambit, and conclude that a multilinear feasibility program finds a Nash equilibrium faster than any of the methods we compare it to, including the quantal response equilibrium method, which is recommended for large games. Hence, the multilinear feasibility program is an alternative method to find a Nash equilibrium in multi-player games, and outperforms many common algorithms. The mixed-integer formulations are generalisations of known mixed-integer programs for two-player games, however unlike two-player games, these mixed-integer programs do not give better performance than existing algorithms.

2012 ACM Subject Classification Theory of computation → Exact and approximate computation of equilibria; Theory of computation → Nonconvex optimization

Keywords and phrases Noncooperative n-person games, Nash equilibrium, Multilinear functions, Nonconvex problems, Mixed-integer optimization

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.12

Supplementary Material
Dataset: `https://github.com/economicsandcomputing/MultilinearNashEquilibria`

Funding *Miriam Fischer*: Supported by a PhD scholarship from DeepMind.

Acknowledgements This research was initiated as part of the MSc dissertation of the first author in the School of Mathematics at the University of Edinburgh.

1 Introduction

A noncooperative game has n players, where $n \geq 2$ is finite, with each player having finitely many pure strategies which they do not discuss or reveal to each other. A mixed strategy for a player is a probability distribution over the player's pure strategies. Each player has a known payoff function which maps any combination of pure strategies of all the n players to a real number. Mixed strategies of all the players form a tuple whose payoff is calculated by taking expectation over the probability distributions. In his seminal work [11], Nash showed that every such game has a tuple of mixed strategies that is an equilibrium in the sense that no player increases their payoff if they were to change their mixed strategy while the others keep theirs fixed. Although existence of Nash equilibrium is guaranteed, uniqueness does not always hold, and there are also characterisations of when there exists an equilibrium formed solely by pure strategies.

[1] Corresponding author

© Miriam Fischer and Akshay Gupte;
licensed under Creative Commons License CC-BY 4.0
21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis; Article No. 12; pp. 12:1–12:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This paper deals with the question of algorithmic and numerical computation of Nash equilibria. From a complexity perspective, computing an equilibrium was only somewhat recently formally settled to being PPAD-complete [1, 2] even for two-player games. There is a lot of literature for two-player bimatrix games, and the most well-known and established exact method to compute an equilibrium is the Lemke-Howson algorithm [8]. This gives a very good computational performance on many instances in practice, although its worst-case performance can take exponentially many pivoting steps [17].

However, for multi-player games, there do not seem to be commonly established approach for computing the equilibrium. Although there is a generalisation of the Lemke-Howson method to n-person games [14, 21], popular algorithmic approaches include a global Newton method [6], an iterated polymatrix approximation approach [7], a simplicial subdivision method [20], a simple search algorithm aiming to find an equilibrium with small support size [13], and a quantal response equilibrium method which gives an approximation to a Nash equilibrium [19]. Many of these methods are implemented in the game-theoretic library `Gambit` [10]. Experiments comparing different methods have been undertaken [16, 13], however it is rather unclear which of the methods is best for multi-player games. For example, the global Newton method gives solid performance for small games, however does not to scale well to larger games [5, 18]. Support enumeration algorithms are fast for games with pure equilibria but will be much slower for a game that only has equilibria of medium to large support size. There are also approximation algorithms, which tend to approximate a Nash equilibrium for large games [19, 5, 3].

We adopt the optimization approach, and propose different optimization formulations for computing a Nash equilibrium for n-person games for $n \geq 2$. Particularly, we present a multilinear polynomial continuous feasibility program of degree n (= number of players), which is a generalisation of the bilinear optimization problem for 2 players [9]. Further, we extend the two-player mixed-integer formulations of [16] to multi-player games, and give a large variety of mixed-integer formulations to find a Nash equilibrium in multi-player games. All our formulations find a Nash equilibrium of a $n \geq 2$ player game. We compare our programs to `gambit-gnm` (global Newton), `gambit-simpdiv` (simplicial subdivision), `gambit-logit` (quantal response equilibrium) algorithms in `Gambit`, focusing on random games and covariant games with negative covariance. We find that the mixed-integer formulations do not give better performance than existing algorithms, and our analysis of those is aimed to get an understanding of which mixed-integer formulations are most suited for finding a Nash equilibrium. We find that our multilinear continuous feasibility program is faster than all the methods in Gambit we compare it to, including the `gambit-logit` method, which is so far recommended for large games. Thus, we provide an alternative approach to computing Nash equilibrium in multi-player games.

The next section presents our continuous and mixed-integer multilinear optimization formulations. For each of them, their correctness, i.e., the fact that their optimal/feasible solutions correspond to Nash equilibria of the game, is proved in the Appendix.

2 Formulations

The multi-player multilinear formulation is an extension of a bilinear formulation for bimatrix games [9]. To motivate the multilinear formulation, we shortly recall the bilinear program that is equivalent to finding a Nash equilibrium in a bimatrix game. To do so, we introduce some notation. Let $A, B \in \mathbb{R}^{m \times n}$ be the payoff matrix of player 1 and player 2, with m pure strategies of player 1 and n pure strategies of player 2. Let $\boldsymbol{x} \in \mathbb{R}^m$ with $\boldsymbol{x} \geq 0$ and

$\sum_{i=1}^{m} x_i = 1$ be a (possibly mixed) strategy of player 1, with x_s being the probability placed on pure strategy s. Let $\boldsymbol{y} \in \mathbb{R}^n$ with $\boldsymbol{y} \geq 0$ and $\sum_{j=1}^{n} y_j = 1$ be a (possibly mixed) strategy of player 2. Let $\mathbf{1}_n$ and $\mathbf{1}_m$ denote vectors of all ones of dimension n and m. Any globally *optimal* solution (x, y, p, q) to the bilinear optimization problem in BLP is equivalent to a Nash equilibrium in a bimatrix game.

$$\text{(BLP)} \quad \begin{aligned} \max_{x,y,p,q} \quad & x^\top Ay + x^\top By - p - q && \text{(1a)} \\ \text{s.t.} \quad & Ay \leq p\mathbf{1}_m, \quad B^\top x \leq q\mathbf{1}_n && \text{(1b)} \\ & \sum_{i=1}^{m} x_i = 1, \quad \sum_{j=1}^{n} y_j = 1, \quad x, y \geq \mathbf{0}. && \text{(1c)} \end{aligned}$$

It is easy to see that any feasible mixed strategies x, y will have objective function value less or equal to zero, as given player 2's (mixed) strategy, any pure strategy of player 1 can give payoff at most p, and given player 1's (mixed) strategy, any pure strategy of player 2 can give payoff at most q. This implies that any combination of pure strategies (i.e. any mixed strategy) of player 1 gives payoff at most p, and any combination of pure strategies of player 2 gives payoff at most q. Further, any Nash equilibrium x^*, y^* has objective function value equal to zero, thus maximises the objective function. This is because players play best responses, and thus $p^* = {x^*}^\top Ay^*$ and $q^* = {x^*}^\top By^*$. Importantly, *only* the Nash equilibria have objective function value of zero. This is because for any optimal solution (x^*, y^*, p^*, q^*) and any (x, y) with $x \geq 0$, $\sum_{i=1}^{m} x_i = 1$, $y \geq 0$, $\sum_{i=1}^{n} y_i = 1$, $x^\top Ay^* \leq p^*$, ${x^*}^\top By \leq q^*$, and thus ${x^*}^\top Ay^* \leq p^*$, ${x^*}^\top By^* \leq q^*$. As a Nash equilibrium has objective function value of zero and is guaranteed to exist, the optimal value of the bilinear formulation must be zero (as it is non-positive). Thus ${x^*}^\top Ay^* = p^*$, ${x^*}^\top By^* = q^*$. This implies $x^\top Ay^* \leq {x^*}^\top Ay^*$, ${x^*}^\top By \leq {x^*}^\top By^*$.

In this work, we propose a multilinear feasibility program whose every feasible solution is a Nash equilibrium to a corresponding multi-player game with $n \geq 2$ players. The formulation is based on an extension of the bilinear formulation to multi-player games. Although such an extension is straightforward, is has not been given much empirical analysis. We compare the multilinear feasibility formulation to established algorithms used to find an equilibrium in multi-player games. We find that our multilinear program is faster than a variety of algorithms in Gambit [10].

2.1 Multilinear formulation

Let us define some notation. Let $n \geq 2$ be the number of players, and $[n] = \{1, \ldots, n\}$ the set of players. Every player i comes with a finite set of pure strategies S_i, with $|S_i| = n_i$. Let $\mathcal{S} = S_1 \times S_2 \times \cdots \times S_n$ be the set of all n-tuples of pure strategy combinations of all players. We will further denote by $\mathcal{S}_{-i} = S_1 \times \cdots \times S_{i-1} \times S_{i+1} \times \cdots \times S_n$ the set of all pure strategy tuples of all players except i. Let $\boldsymbol{s} = (s_1, \ldots, s_n) \in \mathcal{S}$ be a pure strategy tuple of all players and $\hat{\mathbf{s}} = (s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n) \in \mathcal{S}_{-i}$ be a pure strategy tuple of all players other than i. We define payoff matrix $A_i : \mathbf{s} \in \mathcal{S} \to \mathbb{R}$ for player i. As an example, if we had three players, $A_1[s_1, s_2, s_3]$ denotes player 1's payoff when player 1 plays pure strategy s_1, player 2 plays pure strategy s_2 and player 3 plays pure strategy s_3. Likewise, $A_2[s_1, s_2, s_3]$ and $A_3[s_1, s_2, s_3]$ denote player 2 and 3's payoff for the strategy combination $(s_1, s_2, s_3) \in \mathcal{S}$. For pure strategy s of player i and pure strategies $(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n) = \hat{\mathbf{s}} \in \mathcal{S}_{-i}$ of the other players, we write $A_i[s, \hat{\mathbf{s}}]$ to denote the payoff of player i when player i plays pure strategy $s \in \mathcal{S}_i$ and the other players play pure strategies $\hat{\mathbf{s}} \in \mathcal{S}_{-i}$. For every player i, we define strategy vector $\mathbf{x}^i \in \mathbb{R}^{n_i}$, with $\mathbf{x}^i \geq \mathbf{0}$ and $\sum_{s \in S_i} x^i_s = 1$. $\mathbf{x}^i$ is a probability

distribution over player i's pure strategies, and thus a *mixed strategy.* Note that any pure strategy is also a mixed strategy[2]. Let $\mathbf{x} = (\mathbf{x}^1, \dots, \mathbf{x}^n)$ be a mixed strategy profile of all players, and $\mathbf{x}^{-i} = (\mathbf{x}^1, \dots, \mathbf{x}^{i-1}, \mathbf{x}^{i+1}, \mathbf{x}^n)$ be a mixed strategy profile of all players other than i. The product term $\prod_{s_j \in \hat{\mathbf{s}}} x^j_{s_j}$ for $\hat{\mathbf{s}} \in \mathcal{S}_{-i}$ denotes the combined probability of all players except i to play the pure strategy tuple $\hat{\mathbf{s}} \in \mathcal{S}_{-i}$. As an example, if we have three players, player 2 has pure strategies $s_{2,1}$ and $s_{2,2}$ and player 3 has pure strategies $s_{3,1}$ and $s_{3,2}$, then $\mathcal{S}_{-1} = \{(s_{2,1}, s_{3,1}), (s_{2,1}, s_{3,2}), (s_{2,2}, s_{3,1}), (s_{2,2}, s_{3,2})\}$. If player 2 plays $s_{2,1}$ with probability 1/2 and player 3 plays $s_{3,1}$ with probability 1/4, then $\prod_{s_j \in (s_{2,1}, s_{3,1})} = 1/2 * 1/4$, $\prod_{s_j \in (s_{2,1}, s_{3,2})} = 1/2 * 3/4$, $\prod_{s_j \in (s_{2,1}, s_{3,1})} = 1/2 * 1/4$, $\prod_{s_j \in (s_{2,2}, s_{3,2})} = 1/2 * 3/4$. Further, we define vector $\mathbf{p} \in \mathbb{R}^n$. p^i corresponds to player i's highest expected payoff.

▶ **Definition 1.** *Let $\Gamma = \langle \{1, \dots, n\}, (S_i), (A_i) \rangle$ be a game with n, S_i, A_i, $\mathbf{x}^i$ defined as above. Let $\mathbf{x}^i \geq 0$ with $\sum_{s \in S_i} x^i_s = 1$ be a mixed strategy of player i. Then, $\mathbf{x}^* = ({\mathbf{x}^*}^1, \dots, {\mathbf{x}^*}^n)$ with ${\mathbf{x}^*}^i \geq 0$ and $\sum_{s \in S_i} {x^*}^i_s = 1$ for all players i is a (mixed) Nash equilibrium if for all players i and every mixed strategy $\mathbf{x}^i$, we have $\mathbb{E}\left[A_i[\mathbf{x}^*]\right] \geq \mathbb{E}\left[A_i[\mathbf{x}^i, {\mathbf{x}^*}^{-i}]\right]$.*

We now present the multilinear optimization formulation.

(MLP1)

$$\max_{\mathbf{x},\mathbf{p}} \quad \sum_{i=1}^{n} \left(\sum_{\substack{(s,\hat{\mathbf{s}}) \\ \in S_i \times S_{-i}}} A_i[s, \hat{\mathbf{s}}] x^i_s \prod_{s_j \in \hat{\mathbf{s}}} x^j_{s_j} \right) - \sum_{i=1}^{n} p^i \tag{2a}$$

$$\text{s.t.} \quad \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} x^j_{s_j} \leq p^i \qquad \forall i \in [n], s \in S_i \tag{2b}$$

$$\sum_{s \in S_i} x^i_s = 1 \qquad \forall i \in [n] \tag{2c}$$

$$0 \leq x^i_s \leq 1 \qquad \forall i \in [n], s \in S_i \tag{2d}$$

▶ **Theorem 2.** *A (mixed) strategy $(\mathbf{x}^1, \dots, \mathbf{x}^n)$ is a (mixed) Nash equilibrium of the n-player game $(A_1, \dots, A_n)$ if and only if there exist numbers $p^1, \dots, p^n$ such that $(\mathbf{x}^1, \dots, \mathbf{x}^n, p^1, \dots, p^n)$ is an optimal solution to the problem in MLP1.*

It is easy to see that for two players, MLP1 equals the bilinear formulation in BLP, with $\mathbf{x}^1, \mathbf{x}^2$ instead of $\mathbf{x}, \mathbf{y}$, p^1, p^2 instead of p, q, and A_1, A_2 instead of A, B. Computational experiments on small instances reveal that the solver takes significant time to solve MLP1 to optimality. However, further inspections reveal that it is more the verification of an optimal solution, rather than finding an optimal solution, that is the reason for this. Particularly, the solver finds a solution with objective function value 0 (i.e. a Nash equilibrium) relatively quickly, but spends a lot of time verifying that there is no feasible solution with objective function value larger than zero. However, as there cannot be a feasible solution with strictly positive objective value, it is sufficient for the solver to find a feasible solution with objective function value zero, instead of verifying that the upper bound to the optimisation program is zero. Thus, we reformulate the program into a feasibility program, for which the aim is to find a feasible solution for which the objective function (2a) of program MLP1 is non-negative. As a strictly positive solution is not possible, any feasible solution to MLP2 will have a value of zero, and thus be a Nash equilibrium.

[2] When we refer to (mixed) strategies or a (mixed) Nash equilibrium, this includes pure strategies or a pure Nash equilibrium.

▶ **Corollary 3.** *Every feasible solution of MLP2 is a Nash equilibrium.*

$$\max_{\mathbf{x},\mathbf{p}} \quad 0 \tag{3a}$$

$$\text{s.t.} \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s,\hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} x^j_{s_j} \le p^i \quad \forall i \in [n], s \in S_i \tag{3b}$$

$$\text{(MLP2)} \quad \sum_{i=1}^{n} \left(\sum_{\substack{(s,\hat{\mathbf{s}}) \\ \in S_i \times S_{-i}}} A_i[s,\hat{\mathbf{s}}] x^i_s \prod_{s_j \in \hat{\mathbf{s}}} x^j_{s_j} \right) - \sum_{i=1}^{n} p^i \ge 0 \tag{3c}$$

$$\sum_{s \in S_i} x^i_s = 1 \quad \forall i \in [n] \tag{3d}$$

$$0 \le x^i_s \le 1 \quad \forall i \in [n], s \in S_i \tag{3e}$$

2.2 Mixed-integer formulations

For a two-player game, four mixed-integer formulations whose solutions are equivalent to a Nash equilibrium in a two-player game were given in [16]. We generalize these formulations to multi-player games. The notation we use is similar to the notation introduced in the multilinear formulations. Further, we introduce $U^i = \max_{s^l, s^h \in S_i, \hat{\mathbf{s}}^{\mathbf{l}}, \hat{\mathbf{s}}^{\mathbf{h}} \in S_{-i}} A_i[s^h, \hat{\mathbf{s}}^{\mathbf{h}}] - A_i[s^l, \hat{\mathbf{s}}^{\mathbf{l}}]$ be the maximum difference of any two payoffs of player i for any pure strategies of all players.

We have four mixed-integer multilinear formulations, of which one is a feasibility program and three are optimisation programs. All programs have five sets of variables. $x^i_s, r^i_s, u^i_s, \overline{u}^i$ are real variables, and b^i_s is binary. The MIMLPs have the same interpretation and range of values for variables $x^i_s \ge 0$, $\overline{u}^i, u^i_s, r^i_s \ge 0$, further they also come with constraints (4b), (4c), (4d), (4e) (which are mostly such that variables $x^i_s, \overline{u}^i, u^i_s, r^i_s$ are defined as desired). x^i_s, for all players $i \in [n]$ and all pure strategies $s \in S_i$ of player i, denotes the probability with which player i plays pure strategy s. Hence, variables x^i_s give us the mixed strategy played by each player. In order to be valid strategies, all pure strategies of a player must be played with non-negative probability (Eq. 4h) and sum up to one (Eq. 4b), for all players. $\overline{u}^i$ denotes the highest utility player i can achieve by playing any strategy, given the other players mixed strategies. u^i_s is the expected utility of player i of playing pure strategy s, given the other players play their (mixed) strategies (Eq. 4c). Naturally, $\overline{u}^i \ge u^i_s$ (Eq. 4d). $r^i_s = \overline{u}^i - u^i_s$ (Eq. 4e) is the regret of player i of playing pure strategy s. It is defined as the difference of the highest utility of any strategy for the player to the utility of playing strategy s, given the other players' mixed strategies. By definition, the regret of any pure strategy must be non-negative (4h). Further, in any Nash equilibrium, every pure strategy that is played with strictly positive probability must have zero regret. If there was a pure strategy which the player plays and that has positive regret, the player can increase their payoff by putting more probability on a pure strategy with no regret and putting less probability on the pure strategy with regret. Hence, it would not be a Nash equilibrium.

The meaning of binary variables b^i_s is different in all formulations, with not all constraints of MIMLP 1 regarding this variable (Eq. 4f, 4g) present in MIMLP 2,3,4. In formulation 1, if b^i_s is 1, strategy s of player i is not played, hence $x^i_s = 0$. If $b^i_s = 0$, the probability on strategy s is allowed to be positive, however the regret of the strategy must be zero. (4f) ensures that b^i_s can only be set to 1 if zero probability is on s. Further, (4g) ensures that b^i_s can only be set to zero if the strategy's regret is zero (if $b^i_s = 1$, this constraint does not restrict any variable, as $r^i_s \le U^i$ by definition).

▶ **Proposition 4.** *The set of feasible solutions to MIMLP1 is precisely the set of Nash equilibria for the corresponding multi-player game.*

$$
\text{(MIMLP1)}\quad
\begin{aligned}
\min \quad & 0 && && \text{(4a)}\\
\text{s.t.} \quad & \sum_{s \in S_i} x_s^i = 1 && \forall i \in [n] && \text{(4b)}\\
& u_s^i = \sum_{\hat{\mathbf{s}} \in S_{-i}} \prod_{s_j \in \hat{\mathbf{s}}} x_{s_j}^j A_i[s, \hat{\mathbf{s}}] && \forall i \in [n], \forall s \in S_i && \text{(4c)}\\
& \overline{u}^i \geq u_s^i && \forall i \in [n], \forall s \in S_i && \text{(4d)}\\
& r_s^i = \overline{u}^i - u_s^i && \forall i \in [n], \forall s \in S_i && \text{(4e)}\\
& x_s^i \leq 1 - b_s^i && \forall i \in [n], \forall s \in S_i && \text{(4f)}\\
& r_s^i \leq U^i b_s^i && \forall i \in [n], \forall s \in S_i && \text{(4g)}\\
& x_s^i, r_s^i \geq 0,\ u_s^i, \bar{u}_i \in \mathbb{R} && \forall i \in [n], \forall s \in S_i && \text{(4h)}\\
& b_s^i \in \{0, 1\} && \forall i \in [n], \forall s \in S_i && \text{(4i)}
\end{aligned}
$$

MIMLP1 is a feasibility program, for which only Nash equilibria are feasible solutions. MIMLP2, MIMLP3, MIMLP4 have larger feasible regions, as pure strategies with positive probability are allowed to have positive regret, and pure strategies with positive regret are allowed to be played with positive probability. The formulations minimize a penalty, and it is only Nash equilibria for which the penalty is minimal. Thus, only Nash equilibria are optimal solutions. The advantage of these formulations is that, since finding a Nash equilibrium is assumed to be computationally intractable, these formulations can be used to stop the program before an equilibrium has been calculated, and thus give solutions which are close to an equilibrium, also called approximate equilibria. However, it is more difficult with these formulations to find a specific equilibrium among all equilibria, rather than just an arbitrary equilibrium.

MIMLP2 penalises the regret of a pure strategy that is played with positive probability in the objective function, and thus for optimal solutions, the regret of pure strategies with positive probability is zero. MIMLP3 penalises the probability placed on pure strategies with positive regret, and thus optimal solutions will have zero probability on pure strategies with positive regret. MIMLP4 combines the normalised regret and the probability as a penalty, and the solver can choose whether the regret or the probability should be minimized. As [16] noted, these formulations can be used to find approximate Nash equilibria.

MIMLP2 aims to minimize the regret of pure strategies that are played with positive probabilities. Particularly, the regret of a pure strategy played with positive probability serves as a penalty to the objective function. This is done by introducing variable f_s^i for all $i \in [n], s \in S_i$, which represents a pure strategy's regret if the strategy has positive probability and zero otherwise.

▶ **Proposition 5.** *The set of Nash equilibria minimizes the objective function of MIMLP2.*

$$
\text{(MIMLP2)}\quad
\begin{aligned}
\min \quad & \sum_{i=1}^{n} \sum_{s \in S_i} f_s^i - U^i b_s^i && && \text{(5a)}\\
\text{s.t.} \quad & (4b) - (4f), (4h), (4i) && &&\\
& f_s^i \geq r_s^i && \forall i \in [n], \forall s \in S_i && \text{(5b)}\\
& f_s^i \geq U^i b_s^i && \forall i \in [n], \forall s \in S_i && \text{(5c)}
\end{aligned}
$$

MIMLP3 is similar to MIMLP 2, however instead of minimising the regret of pure strategies played with positive probability, the probabilities of pure strategies with positive regret is minimized. To do so, variables g_s^i are introduced, which are set such that a strategy's penalty in the objective function is zero if the strategy's regret is zero, and x_s^i (the probability with which it is played) otherwise. The set of Nash equilibria minimizes the objective, as strategies with positive regret are not played.

▶ **Proposition 6.** *The set of Nash equilibria minimizes the objective function of MIMLP3.*

$$
(\text{MIMLP3}) \quad
\begin{aligned}
\min \quad & \sum_{i=1}^{n} \sum_{s \in S_i} g_s^i - (1 - b_s^i) && \text{(6a)} \\
\text{s.t.} \quad & (4b) - (4e), (4g) - (4i) && \\
& g_s^i \geq x_s^i \qquad \forall i \in [n], \forall s \in S_i && \text{(6b)} \\
& g_s^i \geq 1 - b_s^i \qquad \forall i \in [n], \forall s \in S_i && \text{(6c)}
\end{aligned}
$$

MIMLP4 combines MIMLP 2 and MIMLP 3. Instead of penalising all pure strategies' regret (MIMLP 2) or penalising all pure strategies' probabilities if they have positive regret (MIMLP 3), this formulation lets the solver decide for each pure strategy whether to penalise the regret or the probability. The penalised regret is expressed with variables f_s^i, the penalised probabilities are expressed with variables g_s^i. When using both the regret and the probabilities, the regret must be normalised, as the probability of a pure strategy is between zero and one, but a pure strategy's regret can generally be larger than one. Hence, f_s^i uses normalised regret r_s^i/U^i, which is between zero and one.

▶ **Proposition 7.** *The set of Nash equilibria minimizes the objective function of MIMLP4.*

$$
(\text{MIMLP4}) \quad
\begin{aligned}
\min \quad & \sum_{i=1}^{n} \sum_{s \in S_i} f_s^i + g_s^i && \text{(7a)} \\
\text{s.t.} \quad & (4b) - (4e), (4h), (4i) && \\
& f_s^i \geq r_s^i / U^i \qquad \forall i \in [n], \forall s \in S_i && \text{(7b)} \\
& f_s^i \geq b_s^i \qquad \forall i \in [n], \forall s \in S_i && \text{(7c)} \\
& g_s^i \geq x_s^i \qquad \forall i \in [n], \forall s \in S_i && \text{(7d)} \\
& g_s^i \geq 1 - b_s^i \qquad \forall i \in [n], \forall s \in S_i && \text{(7e)}
\end{aligned}
$$

2.3 Continuous and feasibility formulations

For potential performance improvements of the mixed-integer multilinear programs, we further give continuous as well as feasibility formulations for the MIMLPs. Particularly, for all MIMLPs, we introduce continuous formulations[3] MIMLP1(C), MIMLP2(C), MIMLP3(C), MIMLP4(C), for which constraint 4i, i.e. constraints ($b_s^i \in \{0, 1\}$) is replaced by $b_s^i = (b_s^i)^2$ (which implies $0 \leq b_s^i \leq 1$ and $b_s^i = 0$ or $b_s^i = 1$). Thus, the continuous formulations are equivalent to the MIMLPs. The continuous formulation MIMLP1(C) for MIMLP1 is given in MIMLP1(C), likewise MIMLP2(C), MIMLP3(C), MIMLP4(C) are simply MIMLP2, MIMLP3, MIMLP4 but constraint (4i) replaced by (8a).

---

[3] We note that due to this, the formulations are no longer mixed-integer, however we will still refer to MIMLP(C), to make clear that they belong to the respective MIMLP

$$
\text{(MIMLP1(C))} \qquad \begin{aligned} \min \quad & (4a) \\ \text{s.t.} \quad & (4b) - (4e), (4h) \\ & b_s^i = (b_s^i)^2 \qquad \forall i \in [n], \forall s \in S_i \end{aligned} \tag{8a}
$$

For MIMLP 2,3,4 we also introduce equivalent feasibility formulations MIMLP2(F), MIMLP3(F), MIMLP4(F), by introducing a constraint which requires the objective function of the respective MIMLP to be equal to the optimal value of the MIMLP. Particularly, MIMLP2 and MIMLP3 have optimal objective function of zero, and thus we introduce constraints (5a) $= 0$, i.e. $\sum_{i=1}^n \sum_{s \in S_i} f_s^i - U^i b_s^i = 0$ (MIMLP2) and (6a) $= 0$, i.e. $\sum_{i=1}^n \sum_{s \in S_i} g_s^i - (1 - b_s^i) = 0$ for MIMLP3. MIMLP4 has optimal value $\sum_{i=1}^n |S_i|$, and thus we introduce constraint (7a) $= \sum_{i=1}^n |S_i|$, i.e. $\sum_{i=1}^n \sum_{s \in S_i} f_s^i + g_s^i = \sum_{i=1}^n |S_i|$. For all feasibility formulations, the objective function is changed to 0. MIMLP3(F) is given in MIMLP3(F).

Further, we introduce MIMLP2(C,F), MIMLP3(C,F), MIMLP4(C,F), which combine the continuous and feasibility formulations of MIMLP2,3,4, and are thus continuous multilinear formulations[4]. MIMLP3(C,F) is given in MIMLP3(C,F).

$$
\text{(MIMLP3(F))} \qquad \begin{aligned} \min \quad & 0 \\ \text{s.t.} \quad & (4b) - (4e), (4g) - (4i), (6b), (6c) \\ & (6a) = 0 \end{aligned}
$$

$$
\text{(MIMLP3(C,F))} \qquad \begin{aligned} \min \quad & 0 \\ \text{s.t.} \quad & (4b) - (4e), (4g) - (4h), (6b), (6c), (8a) \\ & (6a) = 0 \end{aligned}
$$

Table 1 Overview of all mixed-integer multilinear formulations.

MIMLP	1	2	3	4	1(C)	2(C)	3(C)	4(C)	2(F)	3(F)	4(F)	2(C,F)	3(C,F)	4(C,F)
Feasibility program	✓				✓				✓	✓	✓	✓	✓	✓
Optimality program		✓	✓	✓		✓	✓	✓						
Continuous					✓	✓	✓	✓				✓	✓	✓
Mixed-integer	✓	✓	✓	✓					✓	✓	✓			

3 Computational Experiments

All experiments are run on a MacBook Pro with 8GB RAM and Intel i5 CPU. Multilinear and mixed-integer formulations are implemented in AMPL [4]. We use BARON 21.1.13 [15] as the solver, which uses FilterSD and FilterSQP as non-linear subsolvers. As the multilinear formulation MLP2 in Equation (MLP2) is much faster than any of the MIMLPs (see Table 3), we decide to only compare MLP2 against common algorithms for multi-player games. The MIMLPs do not seem to give better performance than existing algorithms, and hence the analysis of those is focused on comparing the MIMLPs to each other, to get an understanding which MIMLP formulation is best. Thus, the experiments consist of two parts:

[4] MIMLP2(C,F), MIMLP3(C,F), MIMLP4(C,F) are thus no longer mixed-integer.

1. a comparison of MLP2 with common algorithms in `Gambit` [10] (results in Table 2),
2. a comparison of the different MIMLPs (results in Table 3).

All games are instanced in GAMUT [12] and have integer payoffs. We focus on *random games* and *covariance games* with negative covariance, as previous work [16, 13] indicates that covariance games with negative covariance are challenging to solve experimentally for a variety of algorithms as they tend to only have few equilibria with small support size. We refer to a covariance game with n players and $|S_i|$ actions per player and covariance ρ as CG(n,$|S_i|$,ρ), and to a random game with n players and $|S_i|$ actions per player as RG(n,$|S_i|$). For all games, we take the average of 10 randomly generated instances of that game, and if a method did not find a Nash equilibrium before the timeout (which, depending on the game, is 300 or 900 seconds), we add the timeout to the average.

Table 2 compares MLP2 to the simplicial subdivision method (SD), the global newton method (GN), and the quantal response equilibrium (QRE) in Gambit. The results can be summarised as follows: The simplicial subdivision algorithm is the slowest, and already small instances are sufficient for the algorithm to not find a Nash equilibrium in less than 15 minutes. The global Newton method, although fast on the instances for which it finds an equilibrium, in many instances terminates without giving an equilibrium back. This issues has been reported in different scenarios, see [18], and in these cases, we put the timeout towards the average. The logit algorithm and the multilinear formulation have similar runtime for smaller instances, but for larger games, our formulations seems to be faster. Thus, to conclude, our algorithm is faster than the algorithms in Gambit we test it with, and can be an alternative.

Table 3 presents the results for the MIMLPs and the reformulations. It should be pointed out than any of the MIMLPs takes much longer to find an equilibrium than MLP2, and thus none of the MIMLPs is suited to find an equilibrium for *large* multi-player games. This is different to the mixed-integer formulations for two-player games, for which [16] showed better performance on some instances than existing algorithms. Therefore, the analysis of the MIMLPs aims more to get an understanding what type of formulation is best to find a Nash equilibrium in a multi-player game, than to compare the MIMLPs to common algorithms.

First, the continuous formulations MIMLP2(C), MIMLP3(C), MIMLP4(C) of MIMLP2, MIMLP3, MIMLP4 don't give much performance improvement compared to MIMLP2,3,4. For MIMLP2 and MIMLP3, both the feasibility formulations MIMLP2(F) and MIMLP3(F) and the combined continuous and feasibility formulations MIMLP2(C,F) and MIMLP3(C,F) give better performance than MIMLP2 and MIMLP3, but whether MIMLP2(C,F) and MIMLP3(C,F) are better than MIMLP2(F) and MIMLP3(C,F) depends very much on the game. For MIMLP4, whether MIMLP4(F) or MIMLP4(C,F) are better than MIMLP4 depends on the game. Further, compared over all games, MIMLP1(C), i.e. the continuous formulation of the feasibility formulation MIMLP1 seems to give the best performance.

4 Future Work

Further questions include using different nonlinear solvers for the multilinear formulation. The solver we use finds a Nash equilibrium faster than any of the other algorithms we compare it to, other solvers should only improve the performance of the multilinear feasibility formulation. We also propose generating hard-to-solve instances. Even though GAMUT [12] offers many different types of games, many of these are easy to solve even for large multi-player games. Covariant games are among the few types of games that are (relatively)

difficult to solve in the game generator GAMUT, and therefore we particularly use these instances. However, due to this, there is not much variety in the hard-to-solve instances we can use. Recent work has focused on hard-to-solve instances for polymatrix games (see [3] and `http://polymatrix-games.github.io`), and so more hard-to-solve instances is a direction to explore.

Table 2 Comparison of multilinear feasibility program to state-of-the-art algorithms.

Instance	MLP2	GN	SD	QRE	
CG(5,5,$\rho = -0.2$)	2.35	810.09	900	1.9	average (in seconds)
	100%	10%	0%	100%	percentage solved
	2.53	0.91	–	1.9	average on solved (in seconds)
CG(3,10,$\rho = -0.2$)	0.57	271.56	518.96	0.36	average (in seconds)
	100%	70%	50%	100%	percentage solved
	0.57	2.22	137.9	0.36	average on solved (in seconds)
RG(5,5)	2.23	540.57	632.98	2.08	average (in seconds)
	100%	40%	40%	100%	percentage solved
	2.23	1.43	232.45	2.08	average on solved (in seconds)
RG(3,10)	0.329	91.325	382.9	0.362	average (in seconds)
	100%	90%	70%	100%	percentage solved
	0.329	1.47	161.287	0.362	average on solved (in seconds)
CG(5,10,$\rho = -0.2$)	250.28	825.52	900	361.46	average (in seconds)
	100%	10%	0%	100%	percentage solved
	250.28	155.21	–	361.46	average on solved (in seconds)
RG(5,10)	208.79	900	900	564.32	average (in seconds)
	100%	0%	0%	90%	percentage solved
	208.79	–	–	527.02	average on solved (in seconds)
CG(5,10,$\rho = -0.1$)	220.72	813.47	900	415.64	average (in seconds)
	100%	10%	0%	90%	percentage solved
	220.72	34.77	–	361.82	average on solved (in seconds)

The time is the average over 10 instances of this game in seconds - if no solution is found after the timeout of 15 minutes, the timeout is evaluated as time for the instance.

Table 3 MIMLP results.

Method	RG(3,5)	RG(3,10)	CG(3,5,-0.2)	CG(5,3,-0.2)	RG(5,3)
	Time	Time	Time	Time	Time
MIMLP1	8.41	229.86	107.4	122.73	112.5
MIMLP1(C)	2.876	231.57	41.4	47.96	66.3
MIMLP2	19.11	202.38	16.16	538.17	660.5
MIMLP2(C)	55.93	272.15	165.5	469.5	453.14
MIMLP2(F)	14.4	150.33	29.72	410.94	345.45
MIMLP2(C,F)	14.3	279.03	68.516	198.7	218.16
MIMLP3	46.79	265.53	161.4	392.45	535.9
MIMLP3(C)	75.93	300	200.59	575.32	430.03
MIMLP3(F)	17.67	225.66	18.8	188.94	115.95
MIMLP3(C,F)	9.16	260.98	76.71	54.56	90.91
MIMLP4	5.84	220.969	79.4	359.54	49.75
MIMLP4(C)	110.3	300	69.6	479.0	462.62
MIMLP4(F)	56.5	221.26	129.78	129.59	56.88
MIMLP4(C,F)	59.19	270.69	65.12	248.85	84.52
MLP 2	0.03	0.36	0.035	0.12	0.09

The time is the average over 10 instances of this game in seconds - if no solution is found after the timeout, the timeout is evaluated as time for the instance
RG(3,5), RG(3,10), CG(3,5,-0.2): Timeout after 300 seconds [5 minutes]
CG(5,3,-0.2), RG(5,3): Timeout after 900 seconds [15 minutes]

References

1. Xi Chen and Xiaotie Deng. Settling the complexity of computing two-player Nash equilibrium. *Journal of the ACM*, 56(3):1–57, 2009. doi:10.1145/1516512.1516516.
2. Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a Nash equilibrium. *SIAM Journal on Computing*, 39(1):195–259, 2009. doi:10.1137/070699652.
3. Argyrios Deligkas, John Fearnley, Tobenna Peter Igwe, and Rahul Savani. An empirical study on computing equilibria in polymatrix games. In *Proceedings of the 16st International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '16, pages 186–195, 2016.
4. Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Cengage Learning, 2nd edition, 2002.
5. Ian Gemp, Rahul Savani, Marc Lanctot, Yoram Bachrach, Thomas Anthony, Richard Everett, Andrea Tacchetti, Tom Eccles, and János Kramár. Sample-based approximation of Nash in large many-player games via gradient descent. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '22, pages 507–515. International Foundation for Autonomous Agents and Multiagent Systems, 2022. arXiv:2106.01285.
6. Srihari Govindan and Robert Wilson. A global newton method to compute Nash equilibria. *Journal of Economic Theory*, 110(1):65–86, 2003. doi:10.1016/S0022-0531(03)00005-X.
7. Srihari Govindan and Robert Wilson. Computing Nash equilibria by iterated polymatrix approximation. *Journal of Economic Dynamics and Control*, 28(7):1229–1241, 2004. doi:10.1016/S0165-1889(03)00108-8.
8. C. E. Lemke and J. T. Howson, Jr. Equilibrium points of bimatrix games. *SIAM Journal on Applied Mathematics*, 12(2):413–423, 1964. doi:10.1137/0112033.
9. O.L. Mangasarian and H. Stone. Two-person nonzero-sum games and quadratic programming. *Journal of Mathematical Analysis and Applications*, 9(3):348–355, 1964. doi:10.1016/0022-247X(64)90021-6.

10 Richard D. McKelvey, Andrew M. McLennan, and Theodore L. Turocy. Gambit: Software tools for game theory, version 16.0.2. URL: `http://www.gambit-project.org`.

11 John Nash. Non-cooperative games. *Annals of Mathematics*, 54(2):286–295, 1951. `doi:10.2307/1969529`.

12 Eugene Nudelman, Jennifer Wortman, Yoav Shoham, and Kevin Leyton-Brown. Run the GAMUT: A comprehensive approach to evaluating game-theoretic algorithms. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '04, pages 880–887, USA, 2004. IEEE Computer Society.

13 Ryan Porter, Eugene Nudelman, and Yoav Shoham. Simple search methods for finding a Nash equilibrium. *Games and Economic Behavior*, 63(2):642–662, 2008. `doi:10.1016/j.geb.2006.03.015`.

14 Joachim Rosenmüller. On a generalization of the lemke–howson algorithm to noncooperative n-person games. *SIAM Journal on Applied Mathematics*, 21(1):73–79, 1971. `doi:10.1137/0121010`.

15 Nick V. Sahinidis. BARON 21.1.13: Global Optimization of Mixed-Integer Nonlinear Programs, *User's Manual*. `http://www.minlp.com/downloads/docs/baron%20manual.pdf`, 2017.

16 Thomas Sandholm, Andrew Gilpin, and Vincent Conitzer. Mixed-integer programming methods for finding Nash equilibria. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2*, AAAI'05, pages 495–501. AAAI Press, 2005.

17 Rahul Savani and Bernhard von Stengel. Hard-to-solve bimatrix games. *Econometrica*, 74(2):397–429, 2006. `doi:10.1111/j.1468-0262.2006.00667.x`.

18 Theodore L. Turocy. Answer to question regarding time limits of algorithms in gambit. URL: `https://github.com/gambitproject/gambit/issues/261#issuecomment-660894391`.

19 Theodore L. Turocy. A dynamic homotopy interpretation of the logistic quantal response equilibrium correspondence. *Games and Economic Behavior*, 51(2):243–263, 2005. Special Issue in Honor of Richard D. McKelvey.

20 G. van der Laan, A. J. J. Talman, and L. van der Heyden. Simplicial variable dimension algorithms for solving the nonlinear complementarity problem on a product of unit simplices using a general labelling. *Mathematics of Operations Research*, 12(3):377–397, August 1987.

21 Robert Wilson. Computing equilibria of n-person games. *SIAM Journal on Applied Mathematics*, 21(1):80–87, 1971. `doi:10.1137/0121011`.

A Correctness of the Proposed Formulations

Here we present proofs for the claims made with regards to the formulations presented in this paper.

Proof of Theorem 2. We first show that if $(\bar{\mathbf{x}}^1, \ldots, \bar{\mathbf{x}}^n)$ is a Nash equilibrium to $(A_1, \ldots, A_m)$, then there exist numbers $\bar{p}^1, \ldots, \bar{p}^n$ such that $(\bar{\mathbf{x}}^1, \ldots, \bar{\mathbf{x}}^n, \bar{p}^1, \ldots, \bar{p}^n)$ is an optimal solution to the program in MLP1. Assume that $(\bar{\mathbf{x}}^1, \ldots, \bar{\mathbf{x}}^n)$ is a Nash equilibrium. For any feasible solution $(\mathbf{x}^1, \ldots, \mathbf{x}^n, p^1, \ldots, p^n)$ of MLP1, constraints (2b), (2c) imply

$$\sum_{s \in S_i} \left(x^i_s \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} x^j_{s_j} \right) \leq p^i$$

for all $i \in [n]$. This implies (2a) ≤ 0 for any feasible solution. Set

$$\bar{p}^i = \sum_{s \in S_i} \left(\bar{x}^i_s \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} \bar{x}^j_{s_j} \right)$$

for every $i \in [n]$. We show that $(\bar{\mathbf{x}}^1, \ldots, \bar{\mathbf{x}}^n, \bar{p}^1, \ldots, \bar{p}^n)$ is feasible and optimal to MLP1.

As $(\bar{\mathbf{x}}^1, \dots, \bar{\mathbf{x}}^n)$ is a Nash equilibrium, we have

$$\sum_{s \in S_i} \left(\bar{x}^i_s \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} \bar{x}^j_{s_j} \right) \geq \sum_{s \in S_i} \left(x^i_s \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} \bar{x}^j_{s_j} \right)$$

for all (mixed) strategies $\mathbf{x}^i \geq 0$ with $\sum_{s \in S_i} x^i_s = 1$. Choosing $\mathbf{x}^i = \mathbf{e}_k$, with $k \in \{1, \dots, |S_i|\}$, hence the unit vector with all zeros except one in the k-th component, we have

$$\bar{p}^i = \sum_{s \in S_i} \left(\bar{x}^i_s \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} \bar{x}^j_{s_j} \right) \geq \sum_{\hat{\mathbf{s}} \in \mathcal{S}_{-i}} A_i[k, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} \bar{x}^j_{s_j} \quad \forall k \in \{1, \dots, |S_i|\},$$

satisfying constraint 2b. As we can apply this for all players $i \in [n]$ (and constraints 2c, 2d hold as $(\bar{\mathbf{x}}^1, \dots, \bar{\mathbf{x}}^n)$ is a Nash equilibrium), it follows that $(\bar{\mathbf{x}}^1, \dots, \bar{\mathbf{x}}^m, \bar{p}^1, \dots, \bar{p}^n)$ is feasible. Further, the objective function value is zero at the point $(\bar{\mathbf{x}}^1, \dots, \bar{\mathbf{x}}^n, \bar{p}^1, \dots, \bar{p}^n)$. As the objective function value is at most zero and $(\bar{\mathbf{x}}^1, \dots, \bar{\mathbf{x}}^n, \bar{p}^1, \dots, \bar{p}^n)$ is feasible, it follows that it is optimal.

To show that if $(\bar{\mathbf{x}}^1, \dots, \bar{\mathbf{x}}^n, \bar{p}^1, \dots, \bar{p}^n)$ is an optimal solution to MLP1, $(\bar{\mathbf{x}}^1, \dots, \bar{\mathbf{x}}^n)$ is a Nash equilibrium, we assume that $(\bar{\mathbf{x}}^1, \dots, \bar{\mathbf{x}}^n, \bar{p}^1, \dots, \bar{p}^n)$ indeed is optimal to MLP1. Since a Nash equilibrium exists in this game and has objective value of zero, and all feasible solutions have non-positive value, it follows that the objective value of $(\bar{\mathbf{x}}^1, \dots, \bar{\mathbf{x}}^n, \bar{p}^1, \dots, \bar{p}^n)$ must be zero. For any $\mathbf{x}^i \geq 0$ with $\sum_{s \in S_i} x^i_s = 1$, for all players $i \in [n]$, constraints 2b, 2c imply

$$\sum_{s \in S_i} \left(x^i_s \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} \bar{x}^j_{s_j} \right) \leq \bar{p}^i \quad \forall i \in [n].$$

Particularly,

$$\sum_{s \in S_i} \left(\bar{x}^i_s \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} \bar{x}^j_{s_j} \right) \leq \bar{p}^i \quad \forall i \in [n].$$

As further objective value of $(\bar{\mathbf{x}}^1, \dots, \bar{\mathbf{x}}^n, \bar{p}^1, \dots, \bar{p}^n)$ is zero, we have

$$\sum_{s \in S_i} \left(\bar{x}^i_s \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} \bar{x}^j_{s_j} \right) = \bar{p}^i \quad \forall i \in [n].$$

Hence,

$$\sum_{s \in S_i} \left(x^i_s \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} \bar{x}^j_{s_j} \right) \leq \sum_{s \in S_i} \left(\bar{x}^i_s \sum_{\hat{\mathbf{s}} \in S_{-i}} A_i[s, \hat{\mathbf{s}}] \prod_{s_j \in \hat{\mathbf{s}}} \bar{x}^j_{s_j} \right) = \bar{p}^i$$

$\forall i \in [n], \forall \mathbf{x}^i \geq 0 : \sum_{s \in S_i} x^i_s = 1$. Therefore, $(\bar{\mathbf{x}}^1, \dots, \bar{\mathbf{x}}^n)$ is a Nash equilibrium, as the constraint states that given the other players mixed strategies $\bar{\mathbf{x}}^j$, no strategy of player i can give higher payoff than strategy $\bar{\mathbf{x}}^i$, for all players. ◀

Proof of Proposition 4. For any player $i \in [n]$ and any pure strategy $s \in S_i$ of player i, x^i_s denotes the probability with which player i plays pure strategy s. Constraint 4b,4h guarantee $\boldsymbol{x}^i$ to be a valid mixed strategy for each player i, as all pure strategies are played with non-negative probability and sum up to one. Constraint 4c defines the expected payoff u^i_s of

player i of playing pure strategy s (given the other players' mixed strategies), and 4d defines the highest possible expected payoff $\overline{u}^i$ of any (mixed) strategy of player i given the other players' (mixed) strategies. Constraint 4e,4h define the regret r_s^i of player i of playing pure strategy $s \in S_i$. The regret of a pure strategy is the difference of player i's highest possible expected payoff $\overline{u}^i$ and i's payoff of playing pure strategy s and is non-negative. Constraint 4i introduces binary variable b_s^i for any pure strategy s of any player i. Constraint 4f requires that b_s^i can only be set to one if player i puts zero probability on pure strategy s. Further, constraint 4g ensures that b_s^i can only be set to zero if the strategy's regret is zero (if $b_s^i = 1$, this constraint does not restrict any variable, as $r_s^i \leq U^i$ by definition). Thus, if b_s^i is 1, strategy s of player i is not played, hence $x_s^i = 0$. If $b_s^i = 0$, the probability on strategy s is allowed to be positive, however the regret of the strategy must be zero. Hence, only pure strategies with zero regret can be played with positive probability, which is precisely the definition of a Nash equilibrium. ◀

Proof of Proposition 5. Constraints 4b,4c,4d,4e,4h,4i guarantee that $\boldsymbol{x}^i, \boldsymbol{r}^i, \boldsymbol{u}^i, \overline{u}^i$ are correctly defined for all players. Due to constraint (4f), b_s^i can only be set to one if the probability on the pure strategy is zero. Then, due to minimising f_s^i in (5a) and Equation (5c), f_s^i must be set to U^i. In that case, f_s^i and $U^i b_s^i$ cancel out in the objective, and hence strategies with zero probability have no penalty. If $b_s^i = 0$, f_s^i equals r_s^i, due to minimising f_s^i and Equation (5b), and as $U^i b_s^i = 0$, the penalty of the pure strategy equals the regret of the strategy, and pure strategies that have no regret do not have a penalty. Thus, due to the objective function, it is encouraged to play pure strategies which have no regret, and to not play strategies with regret. Thus, any pure strategy will only contribute to the objective function if it has positive regret *and* probability. The Nash equilibria minimize the objective function, with optimal objective of zero. As any pure strategy in a Nash equilibrium will either have zero probability (hence no penalty) or zero regret (hence no penalty), the objective function will equal zero. Solutions which do not equal a Nash equilibrium have higher objective value, as for some strategies, $f_s^i > 0$ (as r_s^i and U^i are non-negative). ◀

Proof of Proposition 6. We recall that because of constraint (4g), b_s^i can only be set to zero if the strategy's regret r_s^i is zero. By constraint (6c) and minimising g_s^i, if $b_s^i = 0$, then $g_s^i = 1$. Thus, g_s^i and $1 - b_s^i$ cancel out in the objective function and the penalty of strategy s is zero. If $b_s^i = 1$, due to constraint (6b) and minimising g_s^i, $g_s^i = x_s^i$, and $1 - b_s^i = 0$. Hence, the penalty of strategy s equals x_s^i. Therefore, the probability a pure strategy is played with only contributes to the objective function if the strategy has positive regret. Nash equilibria minimize the objective function, and come with optimal value of zero. Constraint (4f) of MIMLP 1 (namely, $x_s^i \leq 1 - b_s^i$) is no longer in this formulation, and it is possible to set $b_s^i = 1$ even if some probability is placed on s. However, in a Nash equilibrium, b_s^i will only be set to 1 if the probability on s is indeed zero, as pure strategies with positive regret are not played. ◀

Proof of Proposition 7. Constraint (7b) demands that if $b_s^i = 0$, then $f_s^i = r_s^i / U^i$, which is at most 1. Further, due to (7e), $g_s^i = 1$. If $b_s^i = 1$, then $f_s^i = 1$ (constraint (7c)) and $g_s^i = x_s^i$ (constraint (7d)), which is at most 1. Hence, $f_s^i + g_s^i$ is at least 1 for every pure strategy s, and additional penalties (either the normalised regret or the probability of the strategy) contribute to the objective function if a strategy has positive probability and positive regret. Any feasible solution that is not a Nash equilibrium has $f_s^i + g_s^i > 1$ for some strategies, as not all strategies have either no regret or zero probability. Nash equilibria minimize the objective function, with value of $\sum_{i=1}^n |S_i|$, as the normalised regret is zero, or the probability of strategy is zero. ◀

Integer Programming Formulations and Cutting Plane Algorithms for the Maximum Selective Tree Problem

Ömer Burak Onar ✉
Department of Industrial Engineering, Bogazici University, Turkey

Tınaz Ekim ✉
Department of Industrial Engineering, Bogazici University, Turkey

Z. Caner Taşkın ✉
Department of Industrial Engineering, Bogazici University, Turkey

Abstract

This paper considers the Maximum Selective Tree Problem (MSelTP) as a generalization of the Maximum Induced Tree problem. Given an undirected graph with a partition of its vertex set into clusters, MSelTP aims to choose the maximum number of vertices such that at most one vertex per cluster is selected and the graph induced by the selected vertices is a tree. To the best of our knowledge, MSelTP has not been studied before although several related optimization problems have been investigated in the literature. We propose two mixed integer programming formulations for MSelTP; one based on connectivity constraints, the other based on cycle elimination constraints. In addition, we develop two exact cutting plane procedures to solve the problem to optimality. On graphs with up to 25 clusters, up to 250 vertices, and varying densities, we conduct computational experiments to compare the results of two solution procedures with solving a compact integer programming formulation of MSelTP. Our experiments indicate that the algorithm CPAXnY outperforms the other procedures overall except for graphs with low density and large cluster size, and that the algorithm CPAX yields better results in terms of the average time of instances optimally solved and the overall average time.

2012 ACM Subject Classification Theory of computation → Integer programming; Mathematics of computing → Graph theory; Mathematics of computing → Network optimization

Keywords and phrases maximum induced tree, selective tree, cutting plane, separation algorithm, mixed integer programming

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.13

Supplementary Material *Software (source code)*: `https://github.com/omarburk/MSelTP-Paper`
archived at `swh:1:dir:a1cfaed88df61df35c2a6bf140f4a3ba21af145b`

1 Introduction

Given an undirected graph with a partition of its vertex set into clusters, we consider the Maximum Selective Tree Problem (MSelTP) which aims to select at most one vertex per cluster such that the graph induced by the selected vertices is a tree and among all possible vertex selections, the number of vertices of the induced tree is maximized. For problems where alternative decisions are represented by vertices belonging to the same clusters, and selections of vertices (from each cluster) force to consider all edges whose both end vertices are selected, it can be important to obtain a minimally connected graph, which is an induced tree. A water-pipe network, gas-pipe network or a type of circuit having a clustered structure and where the flow uses all pipes or wires among allowed/selected nodes are potential application examples of MSelTP. To the best of our knowledge, MSelTP has not been studied in the literature although various related problems have been considered.

© Ömer Burak Onar, Tınaz Ekim, and Z. Caner Taşkın;
licensed under Creative Commons License CC-BY 4.0

Given a graph, finding a largest vertex subset that induced a tree is called The Maximum Induced Tree problem. This problem is a special case of MSelTP where each cluster in a vertex partition consists of a singleton. Since the decision version of the Maximum Induced Tree problem is NP-complete [11], it follows that MSelTP is also NP-hard. The Maximum Induced Tree problem and several variations of it have been studied from various aspects in the literature [10, 22, 11, 2, 8, 36, 20, 27].

Classical combinatorial optimization problems can often be generalized in various ways. The selective version of each combinatorial optimization problem brings some flexibility into their applications by adding a set of alternatives but also adds in their computational complexity. A selective version consists of taking a graph whose vertices are partitioned into clusters and selecting one vertex per cluster so that the graph induced by the selected vertices admits, among all possible selections, the best solution for the original optimization problem. We note that, in the literature of selective problems, the term cluster is used commonly to mean a set of vertices in a vertex partition. Accordingly, clusters are not complete or dense subgraphs unlike in many other contexts such as community detection. Problems of selective nature have been widely studied in the literature. For instance, the selective graph coloring problem (SelCol) which has been extensively studied [26, 6, 33, 16, 7, 17, 38, 39, 5], takes as input a clustered graph and aims to select exactly one vertex per cluster so that, among all possible such selections, the chromatic number of the graph induced by the selected vertices is minimized. SelCol models the wavelength and routing assignment problem and its selective nature allows us to select a route for each connection from a given set of alternative routes [26].

Generalized network design problems, in general, are obtained by clustered graph instances expressing the feasibility conditions of the classical network design problem in terms of clusters [13]. A selective version of the travelling salesman problem has also been considered in the literature. It aims to select exactly one vertex per cluster so that the selected vertices form a cycle with minimum cost [29, 25, 23].

The generalized spanning tree problem (GMSTP) introduced in [28], takes as input a undirected weighted graph and a partition of its vertex set into clusters, and finds a minimum-cost tree that spans exactly one vertex from each cluster. GMSTP allows us to model local and global networks together for telecommunication networks, where hubs in local networks have to be connected via transmission links such as optical fibers to create a minimum cost global network [28, 12, 14, 4, 19, 30, 15, 21, 34, 18, 32]. The group Steiner tree problem is a more general version of GMSTP [37, 9]. Some other related problems are the prize-collecting generalized minimum spanning tree problem [31] and the generalized traveling salesman problem [3].

Given a graph whose vertex set is partitioned into clusters, another way to generalize classical combinatorial optimization problems is to select representative vertices for each cluster and also edges whose both end vertices are selected and satisfy constraints in the original optimization problem. In this way, the subgraph formed by the selection is not necessarily an induced graph. This notion has been studied in several papers [28, 13, 32, 37, 21, 34, 12, 14, 19, 18, 30, 35, 4, 15, 9, 29, 25, 23], where the number of these representative vertices is required to be at least, at most or exactly one per cluster for different problem types.

Let us now introduce MSelTP in a formal way. MSelTP is defined for an undirected graph $G = (V, E)$ whose vertices are partitioned into m vertex sets called *clusters*. Let $|V| = n$ and $\mathcal{K} = \{V_1, \cdots, V_m\}$ be a clustering of V, that is, $V = V_1 \cup V_2 \cdots \cup V_m$ and $V_l \cap V_k = \emptyset$ for all $V_l, V_k \in \mathcal{K}$ such that $l \neq k$. We can assume without loss of generality that edges are defined only between vertices belonging to different clusters since the intracluster edges are irrelevant when at most one vertex is selected from each cluster.

Given a graph $G = (V, E)$ and $V' \subset V$, a graph whose vertex set is V', and whose edges are all the edges of G that have both ends in V' is called a subgraph of G induced by V', and denoted by $G[V']$ [1]. Moreover, for a $|V|$-dimensional binary vector x, let $G[x]$ represent the subgraph induced by the vertex set $\{i \in V \mid x_i = 1\}$, which is the set of selected vertices. Similarly, let $E[x]$ denote the edge set of $G[x]$.

MSelTP is the problem of selecting a maximum number of vertices such that at most one vertex is selected per cluster, and the graph induced by the selected vertices is a tree. Such a tree is called a *maximum selective tree*. Figure 1 illustrates a solution for MSelTP in an undirected graph with 15 vertices partitioned into 5 clusters.

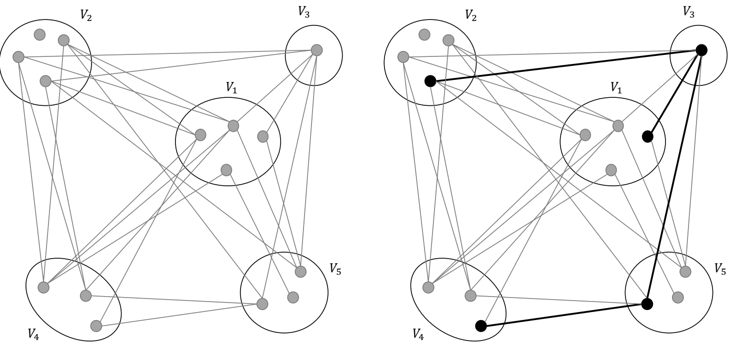


Figure 1 An optimal solution for an instance of MSelTP.

Given a graph $G = (V, E)$ and a subset $S \subset V$, the set of edges in E that have both endpoints in S is denoted by $E(S)$, and the set of edges in E that have only one endpoint in S is denoted by $\delta(S)$:

$$E(S) = \{(i, j) \in E \mid i \in S,\ j \in S\}$$
$$\delta(S) = \{(i, j) \in E \mid i \in S,\ j \notin S\}$$

Given a graph $G = (V, E)$, the directed graph associated with G is denoted by $D = (V, A)$, whose arc set A is composed of arcs $[i, j]$ and $[j, i]$ for each edge $(i, j) \in E$. Similarly, for a directed graph $D = (V, A)$ and a subset $S \subset V$, the set of arcs in A that have both endpoints in S is denoted by $A(S)$, and the set of arcs in A that have only tail or head in S is denoted as follows:

$$A(S) = \{[i, j] \in A \mid i \in S,\ j \in S\}$$
$$\delta^+(S) = \{[i, j] \in A \mid i \in S,\ j \notin S\}$$
$$\delta^-(S) = \{[i, j] \in A \mid i \notin S,\ j \in S\}$$

We use the notations $\delta^+(i)$ and $\delta^-(i)$ instead of $\delta^+(\{i\})$ and $\delta^-(\{i\})$ for brevity.

In Section 2, two integer programming formulations of MSelTP are presented: we present an exact formulation, namely flow based formulation (Model 1) in Subsection 2.1, and a formulation with an exponential number of constraints, namely cycle elimination formulation (Model 2) in Subsection 2.2. In Section 3, we develop two cutting plane algorithms, CPAXnY (Algorithm 1) and CPAX (Algorithm 2), based on the cycle elimination formulation. In Section 4, the computational results of the two cutting plane algorithms (CPAXnY and CPAX) and the flow based formulation (FLOW) are compared.

2 Formulations for MSelTP

2.1 Flow Based Formulation

We propose a formulation for MSelTP based on connectivity constraints. In the formulation, we use flow mechanism introduced by Myung et al. [28] in the directed multicommodity flow model. Model 1 aims to find a connected induced subgraph with maximum number of vertices, which is one more than the number of its edges. Each cluster $V_k \in \mathcal{K}$ corresponds to a commodity. For each commodity k, the flow of commodity k should start from the source cluster, which is uniquely designated (for all commodities) to the destination cluster V_k. Only one source is designated among all clusters from which a vertex is selected. In order to indicate the direction of flow, we use arcs $[i,j]$ and $[j,i]$ in A instead of each edge (i,j) in E, and we create a directed graph $D=(V,A)$ as a directed version of graph $G=(V,E)$. Thus, there may be flow in each direction from i to j and from j to i for each edge $(i,j) \in E$. For each commodity k, we introduce non-integer variables f_{ij}^k to represent the flow of commodity k on arc $[i,j]$ and F_i^k to represent the net flow of commodity k through vertex i. The following binary variables are introduced:
$x_i = 1$ if the vertex i is selected in the solution, 0 otherwise;
$y_{ij} = 1$ if the edge $(i,j) \in E$ is induced by the selected vertices, 0 otherwise;
$w_{ij} = 1$ if there is flow from the vertex $i \in V$ to the vertex $j \in V$, 0 otherwise;
$s_k = 1$ if the cluster $V_k \in \mathcal{K}$ is designated as the source cluster for all the commodities, 0 otherwise;
$t_k = 1$ if no vertex is selected from the cluster $V_k \in \mathcal{K}$, 0 otherwise.

From a flow related point of view, the binary variable x_i indicates whether the vertex i is included in the network so that all flows are sent through, and y_{ij} indicates if the edge (i,j) is able to carry any flow. The variables f_{ij}^k, F_i^k, w_{ij} and s_k are auxiliary flow variables. Our flow based formulation is given in Model 1.

Summing all t-variables, the objective function minimizes the number of clusters having no selected vertex (1). Constraint (2) ensures that the subgraph defined by any solution has at most one vertex from each cluster and that t_k takes value 1 if no vertex is selected from cluster V_k and 0 otherwise. The number of edges induced by the selected vertices, namely $|E[x]|$, should be exactly one less than the number of selected vertices; this is ensured by constraint (3) since the number of selected vertices equals the number of clusters containing a selected vertex. Two selected vertices force the edge between them to be in the (induced) subgraph, with constraint (4). For an edge to be in the induced subgraph, its both end vertices have to be selected, as forced by constraints (5) and (6). Thus, constraints (4), (5) and (6) ensure that the subgraph defined by any solution is the induced subgraph by the selected vertices. It also allows us to relax y-variables as continuous (19) since these constraints force y-variables to be integral.

Flow constraints in general include a root vertex/cluster as a source to send all commodities. In MSelTP, the source cluster has to be designated among those clusters including a selected vertex. If the cluster V_k is designated as the source cluster, only then, the variable s_k should be 1. Constraint (7) forces that only one of the clusters is selected as the source cluster and constraint (8) ensures that only the clusters including a selected vertex can be a source cluster.

Constraints (9) are the net flow equations. Let V_s be the source cluster. The LHS of constraint (9) is the net flow of the commodity k over the vertex i, which should be 1 if i is the selected vertex in the source cluster V_s unless $k = s$ or V_k does not include any selected vertices; -1 if i is the selected vertex in V_k unless $k = s$; and 0 otherwise. These cases are ensured by constraints (10)-(15). Thus, the net flow constraints (9)-(15) suggest that the subgraph defined by any solution has to be connected.

Model 1:

$$\min \quad \sum_{V_k \in \mathcal{K}} t_k \tag{1}$$

s.t.

$$\sum_{i \in V_k} x_i + t_k = 1 \qquad \forall V_k \in \mathcal{K} \tag{2}$$

$$\sum_{(i,j) \in E} y_{ij} = |\mathcal{K}| - 1 - \sum_{V_k \in \mathcal{K}} t_k \tag{3}$$

$$x_i + x_j - 1 \leq y_{ij} \qquad \forall (i,j) \in E \tag{4}$$

$$y_{ij} \leq x_i \qquad \forall (i,j) \in E \tag{5}$$

$$y_{ij} \leq x_j \qquad \forall (i,j) \in E \tag{6}$$

$$\sum_{V_k \in \mathcal{K}} s_k = 1 \tag{7}$$

$$s_k \leq 1 - t_k \qquad \forall V_k \in \mathcal{K} \tag{8}$$

$$\sum_{j:\, [i,j] \in A} f_{ij}^k - \sum_{j:\, [j,i] \in A} f_{ji}^k = F_i^k \qquad \forall i \in V, \quad V_k \in \mathcal{K} \tag{9}$$

$$F_i^k \leq s_k - x_i \qquad \forall i \in V_k, \quad V_k \in \mathcal{K} \tag{10}$$

$$F_i^k \geq s_k - 1 \qquad \forall i \in V_k, \quad V_k \in \mathcal{K} \tag{11}$$

$$F_i^k \leq s_l \qquad \forall i \in V_l, \quad V_l \in \mathcal{K},\ l \neq k \tag{12}$$

$$F_i^k \leq 1 - t_k \qquad \forall i \in V_l, \quad V_l \in \mathcal{K},\ l \neq k \tag{13}$$

$$F_i^k \geq x_i + s_l - t_k - 1 \qquad \forall i \in V_l, \quad V_l \in \mathcal{K},\ l \neq k \tag{14}$$

$$F_i^k \geq 0 \qquad \forall i \in V_l, \quad V_l \in \mathcal{K},\ l \neq k \tag{15}$$

$$f_{ij}^k \leq w_{ij} \qquad \forall [i,j] \in A, \quad V_k \in \mathcal{K} \tag{16}$$

$$w_{ij} + w_{ji} = y_{ij} \qquad \forall (i,j) \in E \tag{17}$$

$$f_{ij}^k \geq 0 \qquad \forall [i,j] \in A, \quad V_k \in \mathcal{K} \tag{18}$$

$$0 \leq y_{ij} \leq 1 \qquad \forall (i,j) \in E \tag{19}$$

$$x_i \in \{0,1\} \qquad \forall i \in V \tag{20}$$

$$w_{ij} \in \{0,1\} \qquad \forall [i,j] \in A \tag{21}$$

$$s_k \in \{0,1\} \qquad \forall V_k \in \mathcal{K} \tag{22}$$

$$t_k \in \{0,1\} \qquad \forall V_k \in \mathcal{K} \tag{23}$$

Considering the edge (i, j) in any selective tree and the flows f_{ij}^k for all $V_k \in \mathcal{K}$ on it, if we hypothetically eliminate this edge to split the selective tree into two trees, for each cluster V_k included in the same tree with the source cluster, the associated commodity k does not flow through (i, j). However, for each cluster V_k included in the other tree, each associated commodity k flows through (i, j) from the tree including the source cluster to the other tree, which indicates that those flows are in the same direction.

When the edge (i, j) is able to carry flow, w_{ij} is hereby used as an indicator of the flow direction for each commodity. Thus, constraints (16) and (17) force that the flow of each commodity on arc $[i, j]$ can be sent only if edge (i, j) is contained in $E[x]$, and if multiple commodities flow on edge (i, j), they all flow in the same direction.

Constraints (5), (6), (9), (16), (17) and (18) ensure that if the vertex i is not selected ($x_i = 0$), related y, w, f and F-variables are forced to be 0 as well. Thus, constraints (12)-(15) allow us to relax F_i^k as $-1 \leq F_i^k \leq 1,\ \forall i \in V,\ V_k \in \mathcal{K}$ since these constraints force F-variables to be integral.

Constraint (3) and the connectivity constraints (9, 16, 17) together ensure that the selection induces also a tree. Therefore, each feasible solution corresponds to a selection of an induced tree with at most one vertex per cluster. If the objective function gives the result of $|\mathcal{K}| - 1$, then, the selection is also a generalized spanning tree.

Model 1 has $O((|V| + |E|) \times |\mathcal{K}|)$ constraints, and $O(|V| + |E| + |\mathcal{K}|)$ binary variables out of $O((|V| + |E|) \times |\mathcal{K}|)$ variables. Since it has a polynomial number of variables and constraints, Model 1 is a compact formulation for MSelTP.

2.2 Cycle Elimination Formulation

We propose another formulation with constraints eliminating cycles for each vertex subset. This formulation has an exponential number of constraints and $O(|V| + |\mathcal{K}|)$ binary variables. Since an acyclic graph with n vertices and $n - 1$ edges is a tree, Model 2 aims to find an acyclic induced graph with maximum number of vertices, which is one more than the number of its edges. The same set of variables x_i, y_{ij} and t_k as in Model 1 are used in Model 2.

Model 2:

$$\min \sum_{V_k \in \mathcal{K}} t_k \tag{24}$$

s.t.

$$\sum_{i \in V_k} x_i + t_k = 1 \qquad \forall V_k \in \mathcal{K} \tag{25}$$

$$\sum_{(i,j) \in E} y_{ij} = |\mathcal{K}| - 1 - \sum_{V_k \in \mathcal{K}} t_k \tag{26}$$

$$x_i + x_j - 1 \leq y_{ij} \qquad \forall (i, j) \in E \tag{27}$$

$$y_{ij} \leq x_i \qquad \forall (i, j) \in E \tag{28}$$

$$y_{ij} \leq x_j \qquad \forall (i, j) \in E \tag{29}$$

$$\sum_{(i,j) \in E(S)} y_{ij} \leq |S| - 1 \qquad \forall S \subset V, \quad 3 \leq |S| \tag{30}$$

$$0 \leq y_{ij} \leq 1 \qquad \forall (i, j) \in E \tag{31}$$

$$x_i \in \{0, 1\} \qquad \forall i \in V \tag{32}$$

$$t_k \in \{0, 1\} \qquad \forall V_k \in \mathcal{K} \tag{33}$$

The flow formulation in Model 1 focuses on the connectivity of the induced subgraph, whereas Model 2 focuses on the fact that the induced graph has to be acyclic. Thus, only the objective function (24), constraint (25) ensuring at most one vertex per cluster, constraint (26) of order-size relation, and constraints (27), (28), (29) forcing the subgraph to be induced by the selected vertices are revisited in this model.

Constraint (30) suggests that the subgraph defined by any solution has to be acylic. For each vertex subset $S \subset V$ with at least 3 vertices, it eliminates all solutions containing a cycle of size $|S|$. Constraints (26) and (30) together ensure that the selection induces a tree. Therefore, each feasible solution yields a selection of an induced tree with at most one vertex per cluster. If the optimal objective function value is $|\mathcal{K}| - 1$, then the selection is also a generalized spanning tree.

3 Cutting Plane Methods

Model 2 has an exponential number of constraints, implying that only relatively small instances can be solved with this formulation. Therefore, we develop cutting plane algorithms that add cycle elimination constraints as needed.

Constraint (30) in Model 2 is of exponential cardinality and it ensures that the subgraph defined by any solution is acylic. Since there is an exponential number of constraints (30), we remove them from the formulation and generate them as needed in a cutting plane fashion. In particular, given a selection of vertices, the feasibility of constraint (30) is checked by running a Depth-First Search algorithm to detect cycles. If there is no cycle, then the selection is feasible. Otherwise, let C denote a detected cycle having vertex set $V(C)$ and edge set $E(C)$. We add the following cut:

$$\sum_{(i,j) \in E(C)} y_{ij} \leq |V(C)| - 1 \tag{34}$$

The cutting plane algorithm CPAXnY summarizes the procedure above as:

Algorithm 1 CPAXnY.

Require: A graph $G = (V, E)$ with clustering $\mathcal{K}$
Ensure: An optimal selection x^* of vertices inducing a maximum selective tree
1: **loop**
2: Solve Model 2 with constraint (30) relaxed. Let $\hat{x}$ be an optimal selection of vertices, and $G[\hat{x}]$ be the subgraph induced by $\hat{x}$.
3: Find cycles in the induced graph, if any, by depth-first search.
4: **if** $G[\hat{x}]$ does not contain any cycles **then**
5: The selection $\hat{x}$ is optimal.
6: $x^* \leftarrow \hat{x}$
7: **stop**
8: **else**
9: For some cycle C, generate cut (34) and add it to Model 2.
10: **end if**
11: **end loop**

Model 2 has $O(|V| + |E| + |\mathcal{K}|)$ variables, which is significantly fewer than in Model 1. Furthermore, we observe that Model 2 can be reformulated in terms of only x and t-variables, hence eliminating y-variables. To do that, all constraints involving y-variables have to be re-expressed. Hence, the related formulation is:

Model 3:

$$\min \sum_{V_k \in \mathcal{K}} t_k$$

s.t.

$$\sum_{i \in V_k} x_i + t_k = 1 \qquad \forall V_k \in \mathcal{K} \tag{35}$$

$$|E[x]| = |\mathcal{K}| - 1 - \sum_{V_k \in \mathcal{K}} t_k \tag{36}$$

$$G[x] \text{ admits no cycle} \tag{37}$$

$$x_i \in \{0,1\} \qquad \forall i \in V \tag{38}$$

$$t_k \in \{0,1\} \qquad \forall V_k \in \mathcal{K} \tag{39}$$

Model 3 has $O(|V|+|\mathcal{K}|)$ variables, which is fewer than in Model 1 and Model 2. Since the sum of y-variables in constraint (26) indicates the number of edges induced by the selected vertices, the size of $E[x]$, which denotes the edge set of $G[x]$, is used in constraint (36). Since $E[x]$ in (36) depends on the selection and since there is an exponential number of (37), we relax constraints (36) and (37). At any point, if constraint (36) is not satisfied, we need to add a cut to eliminate the current selection from the solution set. Laporte and Louveaux define a binary optimality cut to exclude the current solution [24]. Given binary solution v^n with $S = \{i \mid v_i^n = 1\}$ and $S' = \{i \mid v_i^n = 0\}$, the binary optimality cut is defined as:

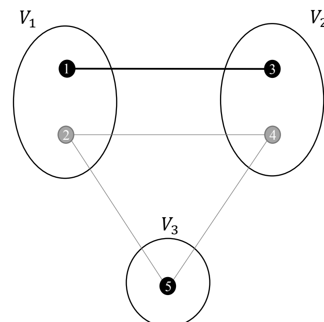
$$\sum_{i \in S}(1 - v_i) + \sum_{i \in S'} v_i \geq 1$$

In our case, the binary variables are x and t-variables. Let $\hat{V}$ be the set of selected vertices $\hat{x}$, and $\hat{\mathcal{K}}$ be $\{V_k \in \mathcal{K} \mid \hat{t}_k = 1\}$, indicating the set consisting of clusters, in which no vertex is selected. Thus, in Model 3, the corresponding binary optimality cut is as follows:

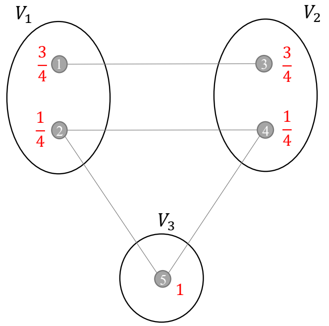
$$\sum_{i \in \hat{V}}(1 - x_i) + \sum_{i \in V \setminus \hat{V}} x_i + \sum_{V_k \in \hat{\mathcal{K}}} (1 - t_k) + \sum_{V_k \in \mathcal{K} \setminus \hat{\mathcal{K}}} t_k \geq 1 \tag{40}$$

By constraint (35), changing a 0-valued x or t variable to 1 forces to change a 1-valued variable to 0, and vice versa. Thus, Equation (40) can be improved as follows:

$$\sum_{i \in \hat{V}}(1 - x_i) + \sum_{V_k \in \hat{\mathcal{K}}} (1 - t_k) \geq 1 \tag{41}$$



(a) A selection that violates (36).



(b) A fractional point satisfying cut (40), and violating cut (42).

Figure 2 A graph example to illustrate that cut (42) is stronger than (40).

Since at most one vertex can be selected from each cluster by constraint (35), the sum of the number of clusters having no selected vertex and the number of selected vertices is equal to the total number of clusters, which leads $|\hat{\mathcal{K}}| + |\hat{V}| = |\mathcal{K}|$. By that equation, an equivalent of Equation (41) is as follows:

$$\sum_{i \in \hat{V}} x_i + \sum_{V_k \in \hat{\mathcal{K}}} t_k \leq |\mathcal{K}| - 1 \tag{42}$$

▶ **Proposition 1.** *Cut* (42) *is stronger than cut* (40).

Proof. Any pair (x, t) with $x \geq 0,\ t \geq 0$ satisfying (41) and hence (42) also satisfies (40).

To show that cut (42) is stronger than cut (40), we consider a graph $G = (V, E)$ with clustering $\mathcal{K} = \{V_1, V_2, V_3\}$ where $V = \{1, 2, 3, 4, 5\}$, $V_1 = \{1, 2\}$, $V_2 = \{3, 4\}$, $V_3 = \{5\}$, and $E = \{\{1, 3\}, \{2, 4\}, \{2, 5\}, \{4, 5\}\}$. One possible selection for this graph is $\hat{x} = \{1, 0, 1, 0, 1\}$, hence $\hat{t} = \{0, 0, 0\}$ as in Figure 2a. The induced graph $G[\hat{x}]$ is, then, composed of vertices $\{1, 3, 5\}$ and edge $(1, 3)$, and constraint (36) is not satisfied. This is a case where a binary optimality cut is generated. Cut (40) associated with selection $\hat{x}$ is $(1 - x_1) + (1 - x_3) + (1 - x_5) + x_2 + x_4 + t_1 + t_2 + t_3 \geq 1$, whereas cut (42) associated with selection $\hat{x}$ is $x_1 + x_3 + x_5 \leq 2$. The point (x, t) with $x = (3/4, 1/4, 3/4, 1/4, 1)$ and $t = (0, 0, 0)$, as in Figure 2b, satisfies constraint (35) and cut (40); however, cut (42) is violated. Hence, cut (42) is stronger than cut (40). ◀

Thus, when the number of edges does not satisfy the condition (36), cut (42) is generated. The feasibility of constraint (37) is checked by running a Depth-First Search algorithm to detect cycles. If there is no cycle, then the selection (x, t) is optimum. Otherwise, let C denote a detected cycle having vertex set $V(C)$. We add the following cut:

$$\sum_{i \in V(C)} x_i \leq |V(C)| - 1 \tag{43}$$

The cutting plane algorithm CPAX summarizes the procedure above.

Algorithm 2 CPAX.

Require: A graph $G = (V, E)$ with clustering $\mathcal{K}$
Ensure: An optimal selection x^* of vertices inducing a maximum selective tree
1: **loop**
2: Solve Model 3 with constraints (36) and (37) relaxed. Let $\hat{x}$ be an optimal selection of vertices, and $G[\hat{x}]$ be the subgraph induced by $\hat{x}$.
3: Find the subgraph $G[\hat{x}]$ induced by $\hat{x}$.
4: Find cycles in $G[\hat{x}]$, if any, by depth-first search.
5: **if** $G[\hat{x}]$ does not contain any cycles **then**
6: **if** $|E[\hat{x}]| = |\mathcal{K}| - 1 - \sum_{V_k \in \mathcal{K}} \hat{t}_k$ **then**
7: The selection is optimal.
8: $x^* \leftarrow \hat{x}$
9: **stop**
10: **else**
11: Generate cut (42) for $(\hat{x}, \hat{t})$ and add it to Model 3.
12: **end if**
13: **else**
14: For some cycle C, generate cut (43) and add it to Model 3.
15: **end if**
16: **end loop**

4 Experimental Results

We conduct computational experiments to compare Algorithm 1 and 2 with solving the compact integer programming formulation Model 1 of MSelTP, called FLOW thereafter. We implement the algorithms in C++ and conduct experiments on a computer with an Intel Core i5-7200U @ 2.50 GHz processor and 8 GB RAM with a time limit of 1500 seconds for each experiment. We use CPLEX 20.1 as a solver. We use lazy callback mechanism in CPLEX to apply the cutting plane algorithms. Source code of our implementation is available at `https://github.com/omarburk/MSelTP-Paper`.

We randomly generate test instances as graphs with clusters varying from 5 to 25. We call the number of vertices in a cluster the *cluster size*, and we use three different average cluster size values, which are 3, 6 and 10. As an average density value, we use 0.1, 0.3, 0.5, and 0.7. For each number of clusters, average cluster size and edge density, 10 random instances are generated.

The graphs are randomly generated so that a vertex can be in any cluster with the same probability and for each pair of vertices, the probability of forming an edge between them is the same, which is the edge density value. However, we assume that there is no empty cluster and edges are defined only between vertices belonging to different clusters since the intracluster edges are irrelevant when at most one vertex is selected from each cluster. Additionally, we only choose connected graphs as test instances. In CPAXnY and CPAX, in iterations where multiple cycles are found, we limit the number of cuts to be added in an iteration to 75.

We conduct experiments on 600 test instances of MSelTP to compare the three methods. The number of clusters in these graphs are 5, 10, 15, 20, and 25. Each of them has 3 different sizes as “small”, “medium”, and “large”, corresponding to the average number of vertices per cluster, which are 3, 6, and 10. There are 4 different density options for each case as 0.1 and 0.3 being “low” densities and 0.5 and 0.7 being “high” densities. For each tuple of types,

Table 1 Summary table for all graph instances.

cluster size	density	FLOW # opt	FLOW avg gap nonopt	FLOW avg time	CPAXnY # opt	CPAXnY avg gap nonopt	CPAXnY avg time	CPAX # opt	CPAX avg gap nonopt	CPAX avg time
small	low	80	83.46	331.88	100		5.20	98	61.90	63.33
	high	42	76.71	902.35	100		31.96	81	31.48	372.40
	all	**122**	**78.44**	**617.11**	**200**		**18.58**	**179**	**34.38**	**217.86**
medium	low	57	99.82	718.64	92	63.35	147.82	89	80.28	190.67
	high	31	97.63	1088.51	76	61.59	485.60	58	74.35	709.72
	all	**88**	**98.47**	**903.57**	**168**	**62.03**	**316.71**	**147**	**75.58**	**450.19**
large	low	46	100.00	903.01	90	91.64	189.26	90	92.33	178.32
	high	23	99.93	1187.03	57	85.63	736.31	42	88.41	896.16
	all	**69**	**99.96**	**1045.02**	**147**	**86.77**	**462.78**	**132**	**88.99**	**537.24**
overall		**279**	**94.21**	**855.24**	**515**	**77.45**	**266.02**	**458**	**75.91**	**401.77**

10 random cases are generated. We first present the general analysis of the results in Table 1. The cases are grouped by their cluster sizes and densities, in which each combination represents 100 instances. The first two columns in this table identify these combinations. Next columns are split into groups for each method (FLOW, CPAXnY, CPAX). In each column group, the first column shows the number of optimally solved instances out of 100. The second column shows the average of the relative optimality gap for nonoptimal cases in percent within the given time limit of 1500 seconds. The relative optimality gap is calculated as $\left(\frac{UB-LB}{UB} \times 100\right)$, where LB is the best known lower bound on the optimal solution value and UB is the upper bound, which is the best integer objective found. The third column shows the average overall time in seconds within the time limit. For each cluster size group, the last row corresponds to aggregate values for all density groups.

The values at the last row of each group and the final row indicate that CPAXnY outperforms the other two methods in terms of each criteria in the table, except the total average relative gap. Moreover, CPAX yields better results than FLOW. CPAXnY and CPAX optimally solves 86% and 76% of the instances, respectively. FLOW solves only 47% of the instances optimally. In terms of the average relative gap and the average time, the ranking between the methods is the same. For only large-size low-density group of instances, CPAX method seems better than CPAXnY in terms of the average time, despite yielding the average relative gap a bit worse, while their number of optimally solved instances are the same. Moreover, CPAX method seems a bit better than CPAXnY in terms of the total average relative gap although CPAXnY method yields better average relative gaps for each group. The reason is that CPAXnY solves all small-size group optimally, therefore it has no nonoptimal case, which affects the total average relative gap.

We observe that the performance of each method deteriorates as the average cluster size increases. Similarly, as the graph gets denser, the performance of each method becomes worse. While CPAXnY method spends a few seconds on average for small-size low-density group and it solves each one of them to optimality, it rises to 736 seconds for large-size high-density group, and it solves only 57% of the instances optimally.

In order to analyze the results better, we compare the methods separately for different cluster sizes and densities. In our next set of experiments, we compare the results of three methods on graphs with "small", "medium" and "large" cluster sizes. The average number of vertices per cluster is 3 in graphs with "small" cluster size, 6 in graphs with "medium" cluster size, whereas it is 10 in the "large" one. The results of the comparison for graphs with small, medium and large cluster sizes are presented in Tables 2, 3 and 4, respectively in Appendix A.

In each table hereafter, the first three columns in the table identify the number of clusters, the number of vertices, and average edge density across ten random instances. Next columns are split into groups for each method (FLOW, CPAXnY, CPAX). In each column group, the first column shows the number of optimally solved instances out of 10. The second column shows the average of the relative optimality gap in percent of the instances that are not solved to optimality within the given time limit of 1500 seconds, while the third column shows that of all the instances. The relative optimality gap is calculated as $\left(\frac{UB-LB}{UB} \times 100\right)$, where LB is the best known lower bound on the optimal solution value and UB is the upper bound, which is the best integer objective found. The fourth and the fifth columns show the average time in seconds of the instances that are not solved to optimality within the time limit and that of all the instances, respectively. The procedures CPAXnY and CPAX have one last column that shows average time spent in their subproblem in seconds of all the instances. As we can see from Tables 2, 3 and 4, in each row, the average time in subproblems is negligible. All the values in each row are the average result of the 10 instances for that case.

As we can see the results in Table 2, the algorithm CPAXnY outperforms the other procedures in terms of the number of optimally solved instances, the average of the relative optimality gap for both nonoptimal cases and overall, and the average time of instances optimally solved and the overall average time. For graphs with lower density and a smaller number of cluster, its results are similar with that of CPAX in terms of the average time of instances optimally solved and the overall average time, yet, as the number of clusters increases, it gave better results. Furthermore, for graphs with higher density, CPAXnY yields significantly better results than others. It still optimally solves all the cases whereas CPAX is not able to solve to optimality except one instance for graphs with 25 clusters. Moreover, for graphs with 20 and 25 clusters, the overall average time of CPAX is longer more than 10 times that of CPAXnY. Both outperform FLOW significantly, which is able to solve very few instances optimally as the number of clusters increases to 15 for graphs with higher density.

As we can see the results in Table 3, the algorithm CPAXnY still outperforms the other procedures in terms of each criteria. However, the performance of each method deteriorates as the cluster size increases. For graphs with lower density and fewer number of clusters, the results of CPAXnY are similar with that of CPAX in terms of the average time of instances optimally solved and the overall average time, yet, as the number of clusters increases, it gave better results. This time, instead of all cases, CPAXnY and CPAX optimally solve 92% and 89% of the instances, respectively. Furthermore, for graphs with higher density, CPAXnY yields much better results than others. However, it optimally solves 76% of the instances this time and 80% of the instances for graphs with 20 cluster, whereas CPAX is not able to solve any instances to optimality for graphs with 20 and 25 clusters. Moreover, CPAXnY has better average of the relative optimality gap for both nonoptimal cases and overall. Both outperform FLOW significantly. Among graphs with higher density, it is able to solve the instances optimally for only those with 5 clusters and half of those with 10 clusters.

As we can see the results in Table 4, for graphs with lower density, the algorithm CPAXnY outperforms the other procedures in terms of the average of the relative optimality gap for both nonoptimal cases and overall; however, the algorithm CPAX yields better results than the other procedures in terms of the average time of instances optimally solved and the overall average time. CPAXnY and CPAX solve the same number of instances to optimality. For graphs with lower density, both CPAXnY and CPAX optimally solve 90% of the instances. Nevertheless, for graphs with higher density, CPAXnY yields much better results than others in terms of each criteria. It optimally solves 57% of the instances this time and 85% of the instances for graphs with 15 cluster, whereas CPAX optimally solves 42% of the instances

and it is not able to solve any instances to optimality for graphs with 15, 20 and 25 clusters except 3 instances for those with 15. Their average times are closer than before. Moreover, CPAXnY has better average of the relative optimality gap for both nonoptimal cases and overall. Both outperform FLOW significantly. Among graphs with higher density, it is able to solve the instances optimally for only those with 5 clusters and 15% of those with 10 clusters. As a limitation, for the large-size high-density graph instances, CPAXnY is not able to solve after those with 20 clusters in 1500 seconds, whereas CPAX is not almost able to solve after those with 15 clusters in 1500 seconds and FLOW is not after those with 10 clusters. As the number of clusters or density increases, the performance of all methods deteriorates in general, especially FLOW method worsens faster. Intuitively, one can think that finding a tree gets easier as density increases; however, finding an induced tree gets harder since the possibility of including a cycle in an induced subgraph increases. Comparing Tables 2, 3 and 4, we observe also that for graphs with high density, the performance of all three methods gets worse in general as the cluster size increases. However, for graphs with low density, the deterioration in performance is less than high density graphs.

As a future research direction, MSelTP can be examined for different kinds of graph classes in terms of algorithms, formulations, and complexity. Moreover, a variant of MSelTP can be studied for forests instead of trees; this would correspond to the selective version of the well-known feedback vertex set problem, which has not been studied to the best of our knowledge. Additionally, vertex-weighted and/or edge-weighted versions of MSelTP can be studied.

References

1. John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph Theory with Applications*. Elsevier Science Publishing Co., Inc., 1976.
2. M. Chudnovsky and Paul D. Seymour. The three-in-a-tree problem. *Combinatorica*, 30:387–417, 2010.
3. Ovidiu Cosma, Petrică C. Pop, and Laura Cosma. An effective hybrid genetic algorithm for solving the generalized traveling salesman problem. In Hugo Sanjurjo González, Iker Pastor López, Pablo García Bringas, Héctor Quintián, and Emilio Corchado, editors, *Hybrid Artificial Intelligent Systems*, pages 113–123, Cham, 2021. Springer International Publishing.
4. Ernando Gomes de Sousa, Rafael Castro de Andrade, and Andréa C. Santos. A multigraph formulation for the generalized minimum spanning tree problem. In *ISCO 2018: Combinatorial Optimization*, volume 10856 of *Lecture Notes in Computer Science*, pages 133–143. Springer, July 2018. doi:10.1007/978-3-319-96151-4_12.
5. Marc Demange, Tınaz Ekim, and Bernard Ries. On the minimum and maximum selective graph coloring problems in some graph classes. *Discrete Applied Mathematics*, 204:77–89, 2016. doi:10.1016/j.dam.2015.10.005.
6. Marc Demange, Tınaz Ekim, Bernard Ries, and Cerasela Tanasescu. On some applications of the selective graph coloring problem. *European Journal of Operational Research*, 240(2):307–314, 2015. doi:10.1016/j.ejor.2014.05.011.
7. Marc Demange, Jérôme Monnot, Petrica Pop, and Bernard Ries. On the complexity of the selective graph coloring problem in some special classes of graphs. *Theoretical Computer Science*, 540-541:89–102, 2014. Combinatorial Optimization: Theory of algorithms and Complexity. doi:10.1016/j.tcs.2013.04.018.
8. Nicolas Derhy and Christophe Picouleau. Finding induced trees. *Discrete Applied Mathematics*, 157:3552–3557, October 2009. doi:10.1016/j.dam.2009.02.009.
9. C.W Duin, A Volgenant, and S Voß. Solving group Steiner problems as Steiner problems. *European Journal of Operational Research*, 154(1):323–329, 2004. doi:10.1016/S0377-2217(02)00707-5.

10 Paul Erdös and Zbigniew Palka. Trees in random graphs. *Discret. Math.*, 46:145–150, 1983.

11 Paul Erdös, Michael Saks, and Vera T Sós. Maximum induced trees in graphs. *Journal of Combinatorial Theory, Series B*, 41(1):61–79, 1986. doi:10.1016/0095-8956(86)90028-6.

12 Corinne Feremans, Martine Labbé, and Gilbert Laporte. A comparative analysis of several formulations for the generalized minimum spanning tree problem. *Networks*, 39:29–34, January 2002. doi:10.1002/net.10009.

13 Corinne Feremans, Martine Labbé, and Gilbert Laporte. Generalized network design problems. *European Journal of Operational Research*, 148(1):1–13, 2003. doi:10.1016/S0377-2217(02)00404-6.

14 Corinne Feremans, Martine Labbé, Gilbert Laporte, and Etudes Commerciales. The generalized minimum spanning tree problem: Polyhedral analysis and branch-and-cut algorithm. *Networks*, 43, November 2002. doi:10.1002/net.10105.

15 Cristiane S. Ferreira, Luis Satoru Ochi, Víctor Parada, and Eduardo Uchoa. A GRASP-based approach to the generalized minimum spanning tree problem. *Expert Systems with Applications*, 39(3):3526–3536, 2012. doi:10.1016/j.eswa.2011.09.043.

16 Yuri Frota, Nelson Maculan, Thiago F Noronha, and Celso C Ribeiro. A branch-and-cut algorithm for partition coloring. *Networks: An International Journal*, 55(3):194–204, 2010.

17 Fabio Furini, Enrico Malaguti, and Alberto Santini. An exact algorithm for the partition coloring problem. *Computers & Operations Research*, 92:170–181, 2018. doi:10.1016/j.cor.2017.12.019.

18 Bruce Golden, Saahitya Raghavan, and Daliborka Stanojevic. The prize-collecting generalized minimum spanning tree problem. *Journal of Heuristics*, 14:69–93, February 2008. doi:10.1007/s10732-007-9027-1.

19 Mohamed Haouari and Jouhaina Siala. Upper and lower bounding strategies for the generalized minimum spanning tree problem. *European Journal of Operational Research*, 171:632–647, February 2006. doi:10.1016/j.ejor.2004.07.072.

20 Alain Hertz, Odile Marcotte, and David Schindl. On the maximum orders of an induced forest, an induced tree, and a stable set. *Yugoslav Journal of Operations Research*, 24:199–215, January 2014. doi:10.2298/YJOR130402037H.

21 Edmund Ihler, Gabriele Reich, and Peter Widmayer. Class Steiner trees and VLSI-design. *Discrete Applied Mathematics*, 90(1):173–194, 1999. doi:10.1016/S0166-218X(98)00090-0.

22 Michal Karonski and Zbigniew Palka. On the size of a maximal induced tree in a random graph. *Math. Slovaca*, 30:151–155, 1980.

23 Gilbert Laporte, Ardavan Asef-Vaziri, and Chelliah Sriskandarajah. Some applications of the generalized travelling salesman problem. *The Journal of the Operational Research Society*, 47(12):1461–1467, 1996.

24 Gilbert Laporte and François V. Louveaux. The integer L-shaped method for stochastic integer programs with complete recourse. *Operations Research Letters*, 13(3):133–142, 1993. doi:10.1016/0167-6377(93)90002-X.

25 Gilbert Laporte and Yves Nobert. Generalized travelling salesman problem through n sets of nodes: An integer programming approach. *INFOR: Information Systems and Operational Research*, 21(1):61–75, 1983. doi:10.1080/03155986.1983.11731885.

26 Guangzhi Li and Rahul Simha. The partition coloring problem and its application to wavelength routing and assignment. In *In Proceedings of the First Workshop on Optical Networks*, 2000.

27 Rafael A. Melo and Celso C. Ribeiro. Maximum weighted induced forests and trees: New formulations and a computational comparative review. *International Transactions in Operational Research*, 2021. doi:10.1111/itor.13066.

28 Young Soo Myung, Chang Ho Lee, and Dong Wan Tcha. On the generalized minimum spanning tree problem. *Networks*, 26(4):231–241, 1995. doi:10.1002/net.3230260407.

29 Charles E. Noon and James C. Bean. A lagrangian based approach for the asymmetric generalized traveling salesman problem. *Operations Research*, 39(4):623–632, 1991.

30 Temel Oncan, Jean-François Cordeau, and Gilbert Laporte. A tabu search heuristic for the generalized minimum spanning tree problem. *European Journal of Operational Research*, 191:306–319, December 2008. doi:10.1016/j.ejor.2007.08.021.

31 Petrica C. Pop. On the prize-collecting generalized minimum spanning tree problem. *Annals of Operations Research*, 150:193–204, March 2007. doi:10.1007/s10479-006-0153-1.

32 Petrica C. Pop. The generalized minimum spanning tree problem: An overview of formulations, solution procedures and latest advances. *European Journal of Operational Research*, 283(1):1–15, 2020. doi:10.1016/j.ejor.2019.05.017.

33 Petrica C. Pop, Bin Hu, and Günther Raidl. A memetic algorithm with two distinct solution representations for the partition graph coloring problem. In *Revised Selected Papers of the 14th International Conference on Computer Aided Systems Theory - EUROCAST 2013 - Volume 8111*, volume 8111, pages 219–226. Springer-Verlag, February 2013. doi:10.1007/978-3-642-53856-8_28.

34 Petrica C. Pop, Walter Kern, and Georg J. Still. *The Generalized Minimum Spanning Tree Problem*. University of Twente, Department of Applied Mathematics, 2000.

35 Petrica C. Pop, Oliviu Matei, Cosmin Sabo, and Adrian Petrovan. A two-level solution approach for solving the generalized minimum spanning tree problem. *European Journal of Operational Research*, August 2017. doi:10.1016/j.ejor.2017.08.015.

36 Dieter Rautenbach. Dominating and large induced trees in regular graphs. *Discrete Mathematics*, 307(24):3177–3186, 2007. doi:10.1016/j.disc.2007.03.043.

37 Gabriele Reich and Peter Widmayer. Beyond Steiner's problem: A VLSI oriented generalization. In *Graph-Theoretic Concepts in Computer Science*, volume 411 of *Lecture Notes in Computer Science*, pages 196–210. Springer, Berlin, Heidelberg, 1990. doi:10.1007/3-540-52292-1_14.

38 Oylum Şeker, Tınaz Ekim, and Z. Caner Taşkın. A decomposition approach to solve the selective graph coloring problem in some perfect graph families. *Networks*, 73(2):145–169, 2019. doi:10.1002/net.21850.

39 Oylum Şeker, Tınaz Ekim, and Z. Caner Taşkın. An exact cutting plane algorithm to solve the selective graph coloring problem in perfect graphs. *European Journal of Operational Research*, 291(1):67–83, 2021. doi:10.1016/j.ejor.2020.09.017.

A Computational Results for graphs with small/medium/large cluster sizes

Table 2 Computational results for graphs with small cluster size.

# clust	# vert	avg density	FLOW # opt	FLOW avg % gap in nonopt	FLOW avg % gap overall	FLOW avg time in opt	FLOW avg time overall	CPAXnY # opt	CPAXnY avg % gap in nonopt	CPAXnY avg % gap overall	CPAXnY avg time in opt	CPAXnY avg time overall	CPAXnY avg time in subpr	CPAX # opt	CPAX avg % gap in nonopt	CPAX avg % gap overall	CPAX avg time in opt	CPAX avg time overall	CPAX avg time in subpr
5	15	0.1	10		0.00	0.22	0.22	10		0.00	0.02	0.02	0.00	10		0.00	0.11	0.11	0.01
5	15	0.3	10		0.00	0.25	0.25	10		0.00	0.03	0.03	0.00	10		0.00	0.05	0.05	0.00
10	30	0.1	10		0.00	0.71	0.71	10		0.00	0.03	0.03	0.00	10		0.00	3.95	3.95	0.09
10	30	0.3	10		0.00	2.94	2.94	10		0.00	0.05	0.05	0.00	10		0.00	0.02	0.02	0.00
15	45	0.1	10		0.00	1.74	1.74	10		0.00	0.05	0.05	0.00	10		0.00	26.60	26.60	0.40
15	45	0.3	10		0.00	134.85	134.85	10		0.00	0.52	0.52	0.00	10		0.00	0.06	0.06	0.01
20	60	0.1	10		0.00	31.11	31.11	10		0.00	0.26	0.26	0.01	10		0.00	36.27	36.27	0.44
20	60	0.3	0	77.03	77.03		1500.00	10		0.00	3.93	3.93	0.01	10		0.00	12.59	12.59	0.14
25	75	0.1	10		0.00	146.96	146.96	10		0.00	6.52	6.52	0.04	9	100.00	10.00	6.31	155.68	0.26
25	75	0.3	0	89.90	89.90		1500.00	10		0.00	40.64	40.64	0.02	9	23.81	2.38	275.48	397.93	0.52
			80	**83.46**	**16.69**	**39.85**	**331.88**	**100**	**0.00**	**0.00**	**5.20**	**5.20**	**0.01**	**98**	**61.90**	**1.24**	**34.01**	**63.33**	**0.18**
5	15	0.5	10		0.00	0.21	0.21	10		0.00	0.03	0.03	0.00	10		0.00	0.02	0.02	0.00
5	15	0.7	10		0.00	0.26	0.26	10		0.00	0.03	0.03	0.00	10		0.00	0.02	0.02	0.00
10	30	0.5	10		0.00	39.64	39.64	10		0.00	0.40	0.40	0.00	10		0.00	0.05	0.05	0.01
10	30	0.7	10		0.00	53.75	53.75	10		0.00	0.86	0.86	0.00	10		0.00	0.24	0.24	0.01
15	45	0.5	2	52.31	41.84	1148.08	1429.62	10		0.00	2.65	2.65	0.00	10		0.00	5.65	5.65	0.07
15	45	0.7	0	64.88	64.88		1500.00	10		0.00	5.21	5.21	0.00	10		0.00	19.54	19.54	0.12
20	60	0.5	0	80.89	80.89		1500.00	10		0.00	27.18	27.18	0.00	10		0.00	322.85	322.85	0.43
20	60	0.7	0	79.16	79.16		1500.00	10		0.00	24.42	24.42	0.00	10		0.00	428.15	428.15	0.57
25	75	0.5	0	88.40	88.40		1500.00	10		0.00	154.78	154.78	0.00	1	32.97	29.67	974.91	1447.49	0.74
25	75	0.7	0	89.75	89.75		1500.00	10		0.00	104.01	104.01	0.00	0	30.14	30.14		1500.00	0.00
			42	**76.71**	**44.49**	**77.02**	**902.35**	**100**	**0.00**	**0.00**	**31.96**	**31.96**	**0.00**	**81**	**31.48**	**5.98**	**107.90**	**372.40**	**0.16**

Table 3 Computational results for graphs with medium cluster size.

			FLOW					CPAXnY						CPAX					
# clust	# vert	avg density	# opt	avg % gap in nonopt	avg % gap overall	avg time in opt	avg time overall	# opt	avg % gap in nonopt	avg % gap overall	avg time in opt	avg time overall	avg time in subpr	# opt	avg % gap in nonopt	avg % gap overall	avg time in opt	avg time overall	avg time in subpr
5	30	0.1	10		0.00	0.20	0.20	10		0.00	0.03	0.03	0.00	10		0.00	0.09	0.09	0.01
5	30	0.3	10		0.00	0.44	0.44	10		0.00	0.05	0.05	0.00	10		0.00	0.02	0.02	0.00
10	60	0.1	10		0.00	2.53	2.53	10		0.00	0.07	0.07	0.00	10		0.00	49.45	49.45	0.56
10	60	0.3	10		0.00	35.46	35.46	10		0.00	0.13	0.13	0.00	10		0.00	0.02	0.02	0.00
15	90	0.1	10		0.00	52.15	52.15	10		0.00	0.22	0.22	0.00	9	100.00	10.00	8.66	157.80	0.20
15	90	0.3	2	99.46	79.57	875.46	1375.09	10		0.00	1.72	1.72	0.00	10		0.00	0.25	0.25	0.03
20	120	0.1	5	100.00	50.00	941.10	1220.55	10		0.00	0.67	0.67	0.01	10		0.00	6.37	6.37	0.20
20	120	0.3	0	99.75	99.75		1500.00	10		0.00	99.67	99.67	0.03	10		0.00	192.18	192.18	0.89
25	150	0.1	0	100.00	100.00		1500.00	10		0.00	2.81	2.81	0.03	10		0.00	0.48	0.48	0.09
25	150	0.3	0	99.89	99.89		1500.00	2	63.35	50.68	864.13	1372.83	0.04	0	78.30	78.30		1500.00	0
			57	**99.82**	**42.92**	**129.19**	**718.64**	**92**	**63.35**	**5.07**	**30.24**	**147.82**	**0.01**	**89**	**80.28**	**8.83**	**28.84**	**190.67**	**0.22**
5	30	0.5	10		0.00	0.74	0.74	10		0.00	0.03	0.03	0.00	10		0.00	0.02	0.02	0.00
5	30	0.7	10		0.00	1.85	1.85	10		0.00	0.05	0.05	0.00	10		0.00	0.02	0.02	0.00
10	60	0.5	10		0.00	506.90	506.90	10		0.00	0.98	0.98	0.00	10		0.00	0.35	0.35	0.02
10	60	0.7	1	90.26	81.23	256.02	1375.60	10		0.00	4.19	4.19	0.00	10		0.00	7.12	7.12	0.11
15	90	0.5	0	97.41	97.41		1500.00	10		0.00	34.52	34.52	0.00	10		0.00	222.16	222.16	0.60
15	90	0.7	0	97.73	97.73		1500.00	10		0.00	66.62	66.62	0.00	8	45.00	9.00	709.40	867.52	1.30
20	120	0.5	0	98.95	98.95		1500.00	8	37.18	7.44	699.24	832.71	0.01	0	64.00	64.00		1500.00	0.00
20	120	0.7	0	98.33	98.33		1500.00	8	24.62	4.92	771.07	916.86	0.01	0	63.57	63.57		1500.00	0.00
25	150	0.5	0	100.00	100.00		1500.00	0	73.36	73.36		1500.00	0.00	0	87.22	87.22		1500.00	0.00
25	150	0.7	0	100.00	100.00		1500.00	0	62.08	62.08		1500.00	0.00	0	88.47	88.47		1500.00	0.00
			31	**97.63**	**67.37**	**172.61**	**1088.51**	**76**	**61.59**	**14.78**	**168.77**	**485.60**	**0.00**	**58**	**74.35**	**31.23**	**137.45**	**709.72**	**0.31**

Table 4 Computational results for graphs with large cluster size.

			FLOW					CPAXnY						CPAX					
# clust	# vert	avg density	# opt	avg % gap in nonopt	avg % gap overall	avg time in opt	avg time overall	# opt	avg % gap in nonopt	avg % gap overall	avg time in opt	avg time overall	avg time in subpr	# opt	avg % gap in nonopt	avg % gap overall	avg time in opt	avg time overall	avg time in subpr
5	50	0.1	10		0.00	0.78	0.78	10		0.00	0.03	0.03	0.00	10		0.00	0.39	0.39	0.03
5	50	0.3	10		0.00	1.81	1.81	10		0.00	0.05	0.05	0.00	10		0.00	0.02	0.02	0.00
10	100	0.1	10		0.00	8.10	8.10	10		0.00	0.17	0.17	0.00	10		0.00	107.83	107.83	0.82
10	100	0.3	10		0.00	356.09	356.09	10		0.00	0.51	0.51	0.00	10		0.00	0.02	0.02	0.00
15	150	0.1	6	100.00	40.00	938.88	1163.33	10		0.00	0.86	0.86	0.00	10		0.00	4.99	4.99	0.15
15	150	0.3	0	100.00	100.00		1500.00	10		0.00	10.90	10.90	0.01	10		0.00	0.16	0.16	0.03
20	200	0.1	0	100.00	100.00		1500.00	10		0.00	5.59	5.59	0.01	10		0.00	1.73	1.73	0.15
20	200	0.3	0	100.00	100.00		1500.00	10		0.00	345.68	345.68	0.06	10		0.00	164.77	164.77	1.32
25	250	0.1	0	100.00	100.00		1500.00	10		0.00	28.77	28.77	0.03	10		0.00	3.31	3.31	0.26
25	250	0.3	0	100.00	100.00		1500.00	0	91.64	91.64		1500.00	0.00	0	92.33	92.33		1500.00	0.00
			46	**100.00**	**54.00**	**202.20**	**903.01**	**90**	**91.64**	**9.16**	**43.62**	**189.26**	**0.01**	**90**	**92.33**	**9.23**	**31.47**	**178.32**	**0.31**
5	50	0.5	10		0.00	2.84	2.84	10		0.00	0.06	0.06	0.00	10		0.00	0.02	0.02	0.00
5	50	0.7	10		0.00	6.65	6.65	10		0.00	0.07	0.07	0.00	10		0.00	0.03	0.03	0.00
10	100	0.5	3	100.00	70.00	1036.14	1360.84	10		0.00	2.53	2.53	0.00	10		0.00	0.27	0.27	0.02
10	100	0.7	0	99.93	99.93		1500.00	10		0.00	26.70	26.70	0.00	9	83.33	8.33	159.36	293.43	0.58
15	150	0.5	0	100.00	100.00		1500.00	10		0.00	465.66	465.66	0.01	3	72.86	51.00	392.90	1167.87	1.47
15	150	0.7	0	99.52	99.52		1500.00	7	44.44	13.33	701.94	868.05	0.00	0	73.24	73.24		1500.00	0.00
20	200	0.5	0	100.00	100.00		1500.00	0	87.30	87.30		1500.00	0.00	0	90.57	90.57		1500.00	0.00
20	200	0.7	0	100.00	100.00		1500.00	0	81.55	81.55		1500.00	0.00	0	92.46	92.46		1500.00	0.00
25	250	0.5	0	100.00	100.00		1500.00	0	94.44	94.44		1500.00	0.00	0	99.82	99.82		1500.00	0.00
25	250	0.7	0	100.00	100.00		1500.00	0	91.60	91.60		1500.00	0.00	0	97.35	97.35		1500.00	0.00
			23	**99.93**	**76.94**	**139.28**	**1187.03**	**57**	**85.63**	**36.82**	**173.05**	**736.31**	**0.00**	**42**	**88.41**	**51.28**	**62.29**	**896.16**	**0.23**

A Graph-Theoretic Formulation of Exploratory Blockmodeling*

Alexander Bille
Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Germany

Niels Grüttemeier
System Technologies and Image Exploitation,
Fraunhofer IOSB, Lemgo, Fraunhofer Institute of Optronics, Germany

Christian Komusiewicz
Institute of Computer Science, Friedrich Schiller Universität Jena, Germany

Nils Morawietz
Institute of Computer Science, Friedrich Schiller Universität Jena, Germany

Abstract

We present a new simple graph-theoretic formulation of the exploratory blockmodeling problem on undirected and unweighted one-mode networks. Our formulation takes as input the network G and the maximum number t of blocks for the solution model. The task is to find a minimum-size set of edge insertions and deletions that transform the input graph G into a graph G' with at most t neighborhood classes. Herein, a neighborhood class is a maximal set of vertices with the same neighborhood. The neighborhood classes of G' directly give the blocks and block interactions of the computed blockmodel.

We analyze the classic and parameterized complexity of the exploratory blockmodeling problem, provide a branch-and-bound algorithm, an ILP formulation and several heuristics. Finally, we compare our exact algorithms to previous ILP-based approaches and show that the new algorithms are faster for $t \geq 4$.

2012 ACM Subject Classification Theory of computation → Social networks; Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Branch-and-bound

Keywords and phrases Clustering, Exact Algorithms, ILP-Formulation, Branch-and-Bound, Social Networks

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.14

Supplementary Material *Software (Source Code and Results)*: `https://git.uni-jena.de/algo-engineering/blockmodeling`
archived at `swh:1:dir:1271064707a641448704346b9ff9dc43cdc799b3`

Funding *Nils Morawietz*: Supported by the Deutsche Forschungsgemeinschaft (DFG), project OPERAH, KO 3669/5-1.

Acknowledgements We would like to thank the anonymous reviewers of SEA for their helpful comments which have improved the presentation of our results.

1 Introduction

In social network analysis, a standard task is to determine which vertices have the same role in the network. One approach for this role assignment problem, called structural equivalence [3], is to assign the same role to vertices if and only if they have the *same* neighborhood. This

* Most of the results of the presented work are also contained in the first author's Master's thesis `https://www.uni-marburg.de/de/fb12/arbeitsgruppen/algorith/paper/master-alex-bille.pdf`

© Alexander Bille, Niels Grüttemeier, Christian Komusiewicz, and Nils Morawietz;
licensed under Creative Commons License CC-BY 4.0

demand is often too strict. In this case, one may instead use blockmodeling. Here, one wants to partition the vertex set of the network into blocks in such a way that vertices with *similar* neighborhoods end up in the same block [3, 25]. The partition of the vertices gives a blockmodel of the input network, which is essentially a graph whose vertices are the blocks and where an edge between two blocks a and b indicates that vertices of block a are likely to interact with vertices of block b. Usually this graph is represented via its adjacency matrix, also called blockmatrix or imagematrix. For an example of a vertex partition of a graph and the corresponding blockmatrix, see Figure 1.

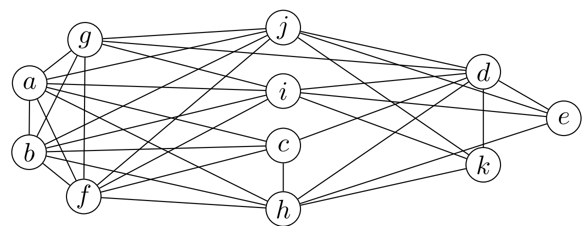
A critical distinction in blockmodeling is whether the blockmatrix is fixed in advance or not [5, 8, 21]. For fixed blockmatrices, the blockmodeling problem essentially asks to confirm whether this blockmatrix is a good explanation for the network; when the blockmatrix is not fixed, then we have an *exploratory* blockmodeling problem, where we aim to identify an unknown model from the given input. In this work, we study a simple graph-theoretic formulation of the exploratory blockmodeling problem. We adopt a graph modification point of view which has already been used for graph clustering [11, 23] and special cases of blockmodeling such as the identification of core/periphery structures [2, 4]. In our formulation, the only prespecified aspect of the blockmodel is the number t of blocks. A starting point is to consider the situation that the graph admits a perfect blockmodel with t (possibly empty) blocks. Informally speaking, this is the case when the number of different neighborhoods of the graph is at most t: In that case, we may simply assign all vertices that have the same neighborhood to the same block. In the resulting blockmodel, each block is either a clique or an independent set in the graph and between two different blocks, either all edges are present or all edges are missing in the input graph.

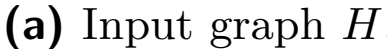
Naturally, for small values of t, most networks will not admit a perfect blockmodel in this sense. Our aim is to find a new network that can be obtained from the original network by a minimum number of edge additions or deletions and which admits a perfect blockmodel with t blocks. In other words, we aim for a blockmodel with t blocks such that the number of disagreements between input network and blockmodel is minimum. This objective function measures globally how different the neighborhoods inside a block are. Thus, minimizing the objective leads to a blockmodel in which vertices with similar neighborhoods indeed end up in the same block.

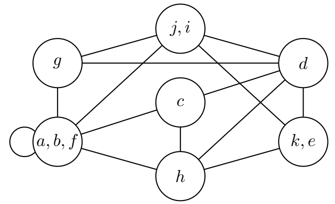
1.1 Related Work

Batagelj et al. [1] also consider the problem of finding the blockmodel with the minimum number of disagreements. There are two main differences: First, in contrast to Batagelj et al. [1], we take a graph-theoretic view. Second, we consider exact algorithms whereas Batagelj et al. [1] use a local search heuristic called TEA. The most closely related exact approaches were used by Brusco and Steinley [5] and Dabkowski et al. [8]. One difference is that these works consider directed networks. A further, algorithmic, difference is that both works solve the exploratory blockmodeling by considering all possible $t \times t$ blockmatrices and selecting one that gives a solution with a minimum number of disagreements. Brusco and Steinley [5] note that their approach is limited to $t \leq 3$ due to the rapidly growing number of possible blockmatrices. Dabkowski et al. [8] show that by considering for example isomorphism classes of blockmatrices, this approach can be extended to $t = 4$. Both works use integer linear programming (ILP) formulations to compute optimal solutions for small networks with up to 20 vertices [5] and 13 vertices [8], respectively.

There are further less closely related formulations. Reichardt and White [21] use a cost function that additionally introduces corrections for the degree distribution of the network. They consider the case with and without prespecified blockmatrix and use local search



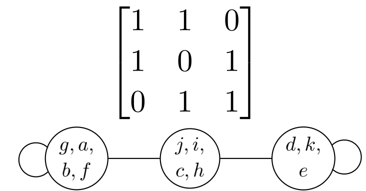

(a) Input graph H.



(b) Neighborhood graph of H. For the sake of clarity, loops are omitted when the corresponding neighborhood class consists of one vertex.

	g	a	b	f	j	i	c	h	d	k	e
g	-	1	1	1	1	1	0	0	1	0	0
a	1	-	1	1	1	1	1	1	0	0	0
b	1	1	-	1	1	1	1	1	0	0	0
f	1	1	1	-	1	1	1	1	0	0	0
j	1	1	1	1	-	0	0	0	1	1	1
i	1	1	1	1	0	-	0	0	1	1	1
c	0	1	1	1	0	0	-	1	1	0	0
h	0	1	1	1	0	0	1	-	1	1	1
b	1	0	0	0	1	1	1	1	-	1	1
k	0	0	0	0	1	1	0	1	1	-	0
e	0	0	0	0	1	1	0	1	1	0	-

(c) Partitioned adjacency matrix of H where red entries are the vertex pairs contained in an optimal set of edge modifications S.



(d) The blockmatrix for H with $t = 3$ obtained by applying S, and the neighborhood graph of $G \triangle S$.

Figure 1 Example instance $(H, t = 3)$ of Blockmodeling with one optimal solution S.

to compute heuristic solutions. Chan et al. [6] use nonnegative matrix factorization to compute blockmodels without prespecified blockmatrix, the factorization problem is solved via a (heuristic) gradient descent method. Jessop [14] and Proll [19] use a model where the objective function rewards large blocks and every block must have a certain minimum density in the input network, both models are solved via ILP formulations.

From the more graph-algorithmic perspective, the most closely related work is due to Damaschke and Mogren [9] who consider graph modification formulations of blockmodeling problems where the blockmatrix is prespecified and each block must become a clique. This problem was shown to be NP-hard for a number of different blockmatrix types [9, 15]; the problem can be solved efficiently when the number of necessary modifications and t are small [9, 13]. Core/periphery problems can be seen as blockmodeling problems for $t = 2$ and a certain fixed blockmatrix [2]. The special case when t equals the number of vertices, each block is fixed to be a clique, and the blockmatrix is fixed to have no edges between different blocks is known as Cluster Editing [23].

A further distinction in blockmodeling approaches is whether they are direct or indirect [3]. Direct approaches compute the blockmodel using the adjacency information of the graph itself. In contrast, indirect approaches follow a two-step procedure where the first step is to compute a distance function for the network vertices based on their neighborhoods and the second step is to compute a clustering of the vertex set with respect to this distance function. In this terminology, our approach and the related ones mentioned above are direct methods.

1.2 Our Results

We first formally define Blockmodeling, a simple edge-modification-based formulation for exploratory blockmodeling with t (possibly empty) blocks that avoids blockmatrices in the problem definition (Section 2). We then show that Blockmodeling is NP-hard for all fixed values of $t \geq 2$ and that it can be solved efficiently when t and the necessary number k of edge modifications are small (Section 3). We develop a branch-and-bound algorithm, an ILP formulation, and several heuristics for Blockmodeling (Section 4). For the branch-and-bound algorithm, we present several speedups based on reduction rules and upper and lower bounds. We evaluate our algorithms experimentally on standard benchmark data sets (Section 5). A comparison with an adaption of the approach of Dabkowski et al. [8] to undirected networks shows that the new algorithm is much faster for $t \geq 4$. For example, for $t = 5$ our new algorithm can find optimal solutions on networks with up to 32 vertices whereas previous approaches can solve only networks with up to 18 vertices. Summarizing, our new approach is competitive with state-of-the-art approaches for exact blockmodeling and paves the way for exact algorithms for larger values of t. In this regard, our approach is a substantial improvement over previous approaches as trying all $t \times t$ blockmatrices becomes clearly infeasible for $t \geq 6$.

Moreover, we find that the heuristics give almost-optimal solutions on the considered instances. Finally, we show that for $t = 4$ our approach finds a reasonable solution for the Karate Club network. Proofs of statements marked with (*) are deferred to the appendix.

2 Preliminaries and Problem Definition

We now introduce some relevant graph-theoretic notation and formally define the exploratory blockmodeling problem.

Notation. For a set S and an integer x, we define $\binom{S}{x} := \{T \subseteq S \mid x = |T|\}$. The symmetrical difference of two sets S and T is denoted by $S \triangle T := (S \cup T) \setminus (S \cap T)$. A collection of sets $(T_1, \ldots, T_x)$ is a *partition* of S if and only if $\bigcup_{i=1}^{x} T_i = S$ and $T_i \cap T_j = \emptyset$ for all $i \neq j$.

A simple undirected graph $G = (V, E)$ consists of a set of *vertices* V and a set of *edges* $E \subseteq \binom{V}{2}$. We set $n := |V|$ and $m := |E|$. Let S be a set of vertices of G, then we denote the deletion of S in G by $G - S := (V \setminus S, \{e \in E \mid S \cap e = \emptyset\})$. The set of edges of G with one endpoint in S and the other in T is denoted by $E_G(S, T) := \{\{s, t\} \in E \mid s \in S, t \in T\}$. Furthermore, let $E_G(S) := E_G(S, S)$ denote the set of edges with both endpoints in S.

The *neighborhood* of a vertex v is defined as $N_G(v) := \{u \mid \{u, v\} \in E\}$. We call the vertices of $N_G(v)$ the *neighbors* of v. If $\{u, v\} \in E$, then we say u and v are *adjacent*. If a vertex v is adjacent to every other vertex of V, then we call v *universal*. A vertex with no neighbors is called *isolated*. Two disjoint vertex sets $S_1 \subseteq V$ and $S_2 \subseteq V$ are *adjacent* if each vertex of S_1 is adjacent to every vertex of S_2. Similarly, S_1 and S_2 are *non-adjacent* if $E_G(S_1, S_2) = \emptyset$. We say the vertices u and v have the same *adjacency* to another vertex w if $\{u, w\} \in E$ if and only if $\{v, w\} \in E$. We say that u and v have the same *adjacency* to a set W of vertices if for each vertex $w \in W$, u and v have the same adjacency to w.

A set of vertices $S \subseteq V$ is a *clique* in G if $\binom{S}{2} \subseteq E$ and an *independent set* in G if $\binom{S}{2} \cap E = \emptyset$. If G is clear from the context, we may omit the subscript.

Problem Definition. Perfect structural equivalence is defined using the following equivalence relation over vertices [17].

▶ **Definition 1.** *Let $G = (V, E)$ be a graph. We let $\sim_G$ denote the relation over V such that $u \sim_G v$ if and only if $N(u) \setminus \{v\} = N(v) \setminus \{u\}$.*

▶ **Definition 2.** *Let $G = (V, E)$ be a graph. The* neighborhood partition *$\mathcal{W}$ of G is the collection of equivalence classes of $\sim_G$. We say that G has a* neighborhood diversity *of $|\mathcal{W}|$.*

The neighborhood partition is unique for each graph and it can be computed in linear time via computing a modular decomposition of the graph [18, 12]. Each set of this partition is called *neighborhood class*. Every neighborhood class is a clique or an independent set. A neighborhood class C is called *positive* if C is a clique and *negative* if C is an independent set. Note that a neighborhood class containing only one element is both positive and negative.

▶ **Definition 3.** *The* neighborhood graph *of a graph with neighborhood partition $\mathcal{W}$ is the graph $(\mathcal{W}, E')$ with $\{W_i, W_j\} \in E'$ if and only if for all $u \in W_i$ and all $v \in W_j$, $\{u, v\} \in E$ or if $W_i = W_j$ and W_i is positive.*

Neighborhood graphs are undirected but not necessarily simple since they may contain loops.

A transformation of a graph G is done by a set of *edits* $S \subseteq \binom{V}{2}$ and results in the graph $G \triangle S := (V, E \triangle S)$. We may now formulate blockmodeling as follows.

Blockmodeling
Input: An undirected graph $G = (V, E)$ and integers k and t.
Question: Is there a set of edits $S \subseteq \binom{V}{2}$ of size at most k such that $G \triangle S$ has neighborhood diversity at most t?

A set of edits that fulfills the above requirements is a *solution* of an instance I, a solution S for I is *optimal* if there is no solution S' for I with $|S'| < |S|$. Given a solution S to an instance of Blockmodeling, we can obtain the blockmodel from $G \triangle S$ as follows: the equivalence classes of $G \triangle S$ are the blocks and the blockmatrix is the adjacency matrix of the neighborhood graph. See Figure 1 for an example instance of Blockmodeling and an optimal solution.

3 NP-Hardness and Kernelization

We now study the complexity of the problem. For the hardness proof, it will be interesting to distinguish vertices whose neighborhoods are changed by the solution from the remaining vertices. Accordingly, a vertex v is called *affected* by a set of edits S if at least one element of S contains v. Otherwise, v is *unaffected* by S.

▶ **Lemma 4 (*).** *If (G, k, t) is a yes-instance of Blockmodeling, then there is a solution such that every vertex of each neighborhood class of size larger than $2k$ in G is unaffected.*

3.1 NP-Hardness

In this section, we show NP-hardness for Blockmodeling for each fixed $t \geq 2$. We first show the NP-hardness for $t = 2$ and afterwards, we extend this result to each fixed $t \geq 2$.

▶ **Lemma 5.** *Blockmodeling is* NP*-hard for $t = 2$.*

Proof. We reduce from Sparse Split Graph Editing[1] which is NP-hard [15].

[1] Note that Sparse Split Graph Editing is a confirmatory formulation of Blockmodeling for $t = 2$ with the fixed block matrix $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$. Hence, the problems are closely related but the NP-hardness of Sparse Split Graph Editing does not directly imply the NP-hardness of Blockmodeling.

Sparse Split Graph Editing
Input: A graph $G = (V, E)$ and an integer k.
Question: Is there an edge set S of size at most k such that $G \triangle S$ is a sparse split-graph, that is, a graph consisting of a clique C and set of isolated vertices P?

Let $I = (G = (V, E), k)$ be an instance of Sparse Split Graph Editing. Moreover, let $G' = (V \cup K, E)$ be the graph obtained by adding an independent set K of size $2k + 1$ to G. We set $I' = (G', k, t = 2)$ and show that I is a yes-instance of Sparse Split Graph Editing if and only if I' is a yes-instance of Blockmodeling.

($\Rightarrow$) Let S be a solution of I, that is, an edge set $S \subseteq \binom{V}{2}$ of size at most k such that $G \triangle S$ is a sparse split-graph. We show that S is a solution for I'. Let $G'_{\text{res}} = G' \triangle S$. Note that all vertices of K are unaffected by S. Hence, all vertices of K are isolated in G'_{res}. Since $G \triangle S$ consists of a clique C and an isolated set P, all vertices of $P \cup K$ are isolated in G'_{res}, and thus in the same neighborhood class. Moreover, the vertices of C are a clique in G'_{res} and no vertex of C has any neighbors in $P \cup K$. Thus, C is a neighborhood class in G'_{res}. Consequently, G'_{res} has a neighborhood diversity of at most 2.

($\Leftarrow$) Let S be an optimal solution for I'. Due to Lemma 4, we can assume that each vertex of K is unaffected by S, that is, $S \subseteq \binom{V}{2}$. We show that $G_S := G \triangle S$ is a sparse split-graph. Since S is a solution for I', S has size at most k. Let C and P be the two (potentially empty) neighborhood classes of $G'_{\text{res}} := (V \cup K, E \triangle S)$. Assume without loss of generality that $K \subseteq P$. Note that this implies that P is an independent set in G'_{res}. Moreover, since each vertex of K is unaffected, each vertex of $P \supseteq K$ is isolated in G'_{res}.

Consequently, C is either an empty set or a clique with no neighbors outside of C in G'_{res}. Thus, G'_{res} is a sparse split-graph. Since G_S is equivalent to $G'_{\text{res}} - K$, G_S is also a sparse split-graph. ◀

With the NP-hardness of Blockmodeling for $t = 2$ at hand, we can now easily obtain NP-hardness of Blockmodeling for each fixed $t \geq 2$.

▶ **Theorem 6 (*).** *Blockmodeling is* NP*-hard for every fixed* $t \geq 2$.

3.2 Kernelization

In this section, we provide a problem kernel for the parameter $k + t$ for Blockmodeling. Informally, this is a data reduction algorithm that, in polynomial time, transforms every instance into an equivalent one whose size is bounded by a function of $t + k$; for a formal definition, we refer to the standard monographs on parameterized algorithms [7, 10]. First, note that Lemma 4 implies the correctness of the following.

▶ **Reduction Rule 1.** *Let $I = (G = (V, E), k, t)$ be an instance of Blockmodeling and let C be a neighborhood class of size at least $2k + 2$ in G. Then, remove $|C| - (2k + 1)$ arbitrary vertices of C from G.*

Before we give the kernel, we make another observation regarding the difference of the neighborhood diversities by an application of one edit.

▶ **Lemma 7 (*).** *Let $G = (V, E)$ be a graph with neighborhood diversity t and let $e = \{u, v\}$ be a vertex pair. Then, the neighborhood diversity of $G \triangle \{e\}$ differs from t by at most 2.*

Based on Lemma 4 and Lemma 7, we are now able to obtain a polynomial kernel as shown in the following theorem.

▶ **Theorem 8.** *Blockmodeling admits a kernel with $\mathcal{O}(k^2 + kt)$ vertices that can be computed in $\mathcal{O}(n + m)$ time.*

Proof. By Lemma 7, a single edit reduces the neighborhood diversity by at most 2. Hence, a graph with more than $2k+t$ neighborhood classes cannot be solved with k edits. Consequently, our algorithm returns false if the neighborhood diversity of the input graph is larger than $2k+t$. Next, we reduce the size of each neighborhood class to at most $2k + 1$ by applications of Reduction Rule 1. After an exhaustive application of Reduction Rule 1, any instance has at most $(2k + t) \cdot (2k + 1) = 4k^2 + 2k + 2kt + t \in \mathcal{O}(k^2 + kt)$ vertices. Since the neighborhood partition can be computed in linear time [18], and each application of Reduction Rule 1 can be done simultaneously, this whole algorithm runs in linear time. ◀

Informally, this means that large instances where both k and t are small can be replaced by relatively small instances (which can then be solved faster by an exact algorithm of choice).

4 Algorithms

4.1 Branch-and-Bound

Basic Search Tree Algorithm. Our branch-and-bound algorithm considers $t + 1$ many vertices from distinct neighborhood classes in G. At least two of these vertices have to be in the same neighborhood class in the resulting graph. Recall that two vertices are in different neighborhood classes if and only if there is some vertex to which they have a different adjacency.

▶ **Definition 9.** *Let u and v be vertices. A vertex w is a* witness *of a vertex pair $\{u, v\}$ if and only if w is adjacent to exactly one of u and v.*

Let $\text{wit}(\{u, v\})$ denote the set of witnesses of $\{u, v\}$. To bring u and v in the same neighborhood class, all witnesses need to be *resolved*. Consider a witness w of $\{u, v\}$ with $\{w, u\} \in E$ and $\{w, v\} \notin E$. To resolve w, we have to either add the missing edge $\{w, v\}$ or delete the present edge $\{w, u\}$. This decision has to be done for each witness independently. Hence, when $r = |\text{wit}(\{u, v\})|$ is the number of witnesses of $\{u, v\}$, there are 2^r different possibilities to achieve that u and v are in the same neighborhood class. We call an edit set S a *resolve set* of $\{u, v\}$ if the edits of S resolve every witness of $\{u, v\}$ and S is minimal under this property.

▶ **Observation 10.** *Let $G := (V, E)$ and $G' := (V, E')$ be graphs on the same vertex set V and let u and v be distinct vertices of V. If u and v are in the same neighborhood class in G', then there is some resolve set S of $\{u, v\}$ such that $S \subseteq E \triangle E'$. More specific, for each witness w of $\{u, v\}$, $E \triangle E'$ contains either $\{u, w\}$ or $\{v, w\}$.*

With this observation at hand, we can write a basic version of a branch-and-bound algorithm shown in Algorithm 1.

Note that the algorithm is correct, since if I is a yes-instance of Blockmodeling, there is an optimal solution S for I, such that for some pair $\{u, v\} \in \binom{T}{2}$, u and v are in the same neighborhood class in $G \triangle S$. Due to Observation 10, for each witness w of $\{u, v\}$, S contains either $\{u, w\}$ or $\{v, w\}$. In particular, this holds for the vertex w chosen at Line 7. Hence, I is a yes-instance of Blockmodeling if and only if for some instance I' defined in Line 10, I' is a yes-instance of Blockmodeling.

Algorithm 1 `solveBaB`.

1: **Input** Instance $I = (G, k, t)$ of BLOCKMODELING
2: **Output** *True* if and only if I is a yes-instance of BLOCKMODELING
3: **if** $k < 0$ **then return** *False*
4: **if** G has neighborhood diversity of at most t **then return** *True*
5: $T \leftarrow$ a set of $t + 1$ vertices of distinct neighborhood classes in G
6: **for all** $\{u, v\} \in \binom{T}{2}$ **do**
7: $\quad w \leftarrow$ a witness of $\{u, v\}$
8: $\quad$ **for all** $e \in \{\{u, w\}, \{v, w\}\}$ **do**
9: $\quad\quad G' \leftarrow G \triangle \{e\}$
10: $\quad\quad I' \leftarrow (G', k - 1, t)$
11: $\quad\quad$ **if** `solveBaB`(I') **then return** *True*
12: **return** *False*

In the following, we bound the running time of Algorithm 1. Since T contains exactly $t+1$ vertices, Algorithm 1 reaches Line 10 exactly $\binom{t+1}{2} \cdot 2 = t^2 + t$ times. Each instance defined in Line 10 reduces k by exactly one. Hence, the search tree has at most $(\binom{t+1}{2} \cdot 2)^k = (t^2 + t)^k$ leaves.

The running time of the other computations depend on n. The neighborhood classes of G can be computed in $\mathcal{O}(n + m) \subseteq \mathcal{O}(n^2)$ time [18]. Moreover, when given the adjacency matrix, for each pair $\{u, v\} \in \binom{T}{2}$, a witness of $\{u, v\}$ can be found in $\mathcal{O}(n)$ time. Since the adjacency matrix can be computed in $\mathcal{O}(n^2)$ time, Algorithm 1 runs in $\mathcal{O}((t^2 + t)^k \cdot n^2)$ time. By initially applying the kernelization algorithm presented in Theorem 8, we obtain the following.

▶ **Theorem 11.** *BLOCKMODELING can be solved in $\mathcal{O}((t^2 + t)^k \cdot (k^2 + kt)^2 + n + m)$ time.*

Thus, the problem can be solved efficiently when t and k are small.

Heuristic Speedups. To reduce the running time of the branch-and-bound algorithm we develop several speedups. In the following we describe those that have the highest impact.

The first speedup is an improved branching: Once we have branched into the case that two vertices u and v from a set T need to be merged, we consider the witnesses of u and v one by one. That is, after branching on witness w, we check in the recursive calls whether u and v still have some witness w', and if this is the case, we directly branch into the two ways to resolve w'. As a consequence, often the number of created branches is 2 instead of $t^2 + t$. Furthermore, we store all witnesses for each vertex pair and update them whenever an edit is performed. We use the stored witnesses to update the neighborhood partition after an edit is done.

This branch-and-bound algorithm uses the solution size k^* of any of our heuristics (described in Section 4.3) as an initial upper bound for any optimal solution. The algorithm then searches for a solution S of size at most $k^* - 1$. If such a solution is found, the algorithm decreases the value of k^* by 1 and continues to search for a solution of size at most $k^* - 1$. This is done until no solution of size $k^* - 1$ can be found, that is, if the size of each optimal solution is k^*.

When an edge e is added or removed, we label the vertex pair e as *permanent* in the corresponding child branches. The algorithm forbids that a permanent vertex pair can be edited again. Whenever all witnesses of vertex pair $\{u, v\}$ of the for-loop in Line 6 are resolved, we label this vertex pair as *merged* for its recursive calls. Each two vertices of a merged vertex pair should be in the same neighborhood class in the resulting graph. If each recursive call with u and v merged returns false, we label $\{u, v\}$ as *apart* for the remaining recursive calls. Based on this labeling, we introduce the following reduction rule.

▶ **Reduction Rule 2.** *Let $\{u, v\}$ be a merged vertex pair and let w be a witness of $\{u, v\}$. a) If both vertex pairs $\{u, w\}$ and $\{v, w\}$ are permanent, then return false. b) If the vertex pair $\{u, w\}$ is permanent, then edit $\{v, w\}$.*

The labeling can also be used as follows: any vertex pair that is labeled apart can be skipped in the for-loop of Line 6. Another improvement is the use of a lower bound algorithm (LBA) which computes a number that underestimates the optimal solution size. Our LBA computes disjoint sets of $t + 1$ many vertices of distinct neighborhood classes in the current graph. We call such a set *pack* and a collection of packs is called a *packing*. We define the cost of a pack P as $\min_{p \in \binom{P}{2} \setminus A} |\text{wit}(p)|$ where A are the vertex pairs labeled as apart. The cost is the minimum number of edits that are incident with vertices of P. For a packing $\mathcal{P}$, the sum over the cost of each pack in $\mathcal{P}$ divided by 2 is a lower bound for an instance; division by 2 is necessary since one edit may resolve witnesses in two packs. Furthermore, we try to increase the lower bound by a local search that swaps vertices of a pack with vertices of another pack or with vertices that are in no pack. The packing is updated after each edit; after a fixed time the packing is deleted and recomputed.

4.2 ILP Formulation

The idea of our ILP is derived from Observation 10. Let $G = (V, E)$ be the input graph. The editing of one vertex pair $\{u, v\}$ is represented by the *edit variable* $e_{\{u,v\}} \in \{0, 1\}$. The vertex pair $\{u, v\}$ is in the solution if and only if $e_{\{u,v\}} = 1$. Hence, the objective is to minimize $\sum_{\{u,v\} \in \binom{V}{2}} e_{\{u,v\}}$.

We introduce a *merge variable* $m_{\{u,v\}} \in \{0, 1\}$ for each vertex pair $\{u, v\} \in \binom{V}{2}$. If a merge variable $m_{\{u,v\}}$ equals 1, all witnesses of $\{u, v\}$ must be resolved. Let w be a witness of $\{u, v\}$. The constraints $m_{\{u,v\}} \leq e_{\{u,w\}} + e_{\{v,w\}}$ and $m_{\{u,v\}} \leq 2 - e_{\{u,w\}} - e_{\{v,w\}}$ guarantee that exactly one edit variable equals 1 and thus, in the solution w is resolved for $\{u, v\}$. Next, we introduce constraints for ensuring that there will be at most t neighborhood classes: For every vertex set V' of size $t + 1$, at least two vertices have to be in the same neighborhood class in the resulting graph. Thus, there is at least one vertex pair $\{x, y\}$ with $x \in V'$ and $y \in V'$ such that $m_{\{x,y\}} = 1$ for every solution. To model the transitivity of $\sim_G$, we add further constraints. Consider the vertices u, v, and w. If $m_{\{u,w\}} = m_{\{v,w\}} = 1$, then $m_{\{u,v\}}$ has to be 1 as well. This is equivalent to the constraint $m_{\{u,w\}} + m_{\{v,w\}} - m_{\{u,v\}} \leq 1$. The ILP is given by

$$\min \quad \sum_{\{u,v\} \in \binom{V}{2}} e_{\{u,v\}},$$

$$\text{s.t.} \quad m_{\{u,v\}} \leq e_{\{u,w\}} + e_{\{v,w\}} \qquad \forall \{u, v\} \in \binom{V}{2}, \forall w \in V, \tag{a}$$

$$m_{\{u,v\}} \leq 2 - e_{\{u,w\}} - e_{\{v,w\}} \qquad \forall \{u, v\} \in \binom{V}{2}, \forall w \in V, \tag{b}$$

$$1 \leq \sum_{\{u,v\} \in \binom{X}{2}} m_{\{u,v\}} \qquad \forall X \in \binom{V}{t+1}, \tag{c}$$

$$1 \geq m_{\{u,w\}} + m_{\{v,w\}} - m_{\{u,v\}} \qquad \forall \{u, v, w\} \in \binom{V}{3}, \tag{d}$$

$$e_{\{u,v\}} \in \{0, 1\}, \ m_{\{u,v\}} \in \{0, 1\} \qquad \forall \{u, v\} \in \binom{V}{2}.$$

Now, we analyze the number of variables and constraints. For each vertex pair, the ILP has an edit variable and a merge variable. In total, there are $2 \cdot \binom{n}{2} \in \mathcal{O}(n^2)$ variables. Two constraints are constructed for each witness of a vertex pair. A vertex pair can have up to $n-2$ witnesses. Thus, the ILP has $\binom{n}{2} \cdot (n-2) \in \mathcal{O}(n^3)$ *witness constraints* (a) and (b). There are $\mathcal{O}(n^3)$ many *transitivity constraints* (d) as well, one for each vertex triple. The largest number of constraints is taken by the *merge constraints* (c). There are $\binom{n}{t+1} \in \Theta(n^{t+1})$ such constraints. To improve the running time of this algorithm, we add the transitivity and merge constraints in a lazy way. That is, initially, the ILP contains only the witness constraints. Whenever a solution is found for the current constraints, then we construct the graph $G' = G \triangle S$ where S is the set of vertex pairs p with $e_p = 1$ in the current solution. If the neighborhood diversity of G' is larger than t, we find a vertex set X of size $t+1$ of pairwise distinct neighborhood classes in G'. Then, we add a merge constraint for X to the ILP together with the transitivity constraints for all vertex triples of $\binom{X}{3}$. Finally, a current solution may create a new witness w for some vertex pair $\{u, v\}$, for example by editing the edge $\{u, w\}$ and not the edge $\{v, w\}$. To model that this new witness either needs to be resolved by an additional edit or that the current solution must be changed, we add the constraint $m_{\{u,v\}} \leq (1 - e_{\{u,w\}}) + e_{\{v,w\}}$ in this case.

4.3 Heuristics

We now present two greedy heuristic algorithms and a local search approach.

Block-Framework. Let $G = (V, E)$ be the input graph. All heuristics maintain some partition $\mathcal{B} = \{B_0, B_1, \ldots, B_b\}$ of V during the computation, each set of this partition is called a *block*. In each step, a greedy heuristic computes a new partition with a decreased number of blocks. This is repeated until the partition consists of t blocks. To choose the next partition, we need to compute the cost of a partition as follows. For each block, we compute the minimal cost for putting all vertices of this block into the same neighborhood class by considering two aspects: First, a block has to be a clique or independent set in the resulting graph G'. Second, two blocks have to be adjacent or non-adjacent in G'. We define the cost functions ω_1 and ω_2 that compute the minimal cost for both aspects. The cost functions are defined as

$$\omega_1(B_i) := \min(\binom{|B_i|}{2} - |E(B_i)|, |E(B_i)|),$$
$$\omega_2(B_i, B_j) := \min(|B_i| \cdot |B_j| - |E(B_i, B_j)|, |E(B_i, B_j)|).$$

Then, the cost of a partition is the sum of the cost of each block and of each pair of blocks:

$$\omega(\mathcal{B}) := \sum_{B_i \in \mathcal{B}} \omega_1(B_i) + \sum_{\{B_i, B_j\} \in \binom{\mathcal{B}}{2}} \omega_2(B_i, B_j).$$

Merge-Heuristic. The initial partition $\mathcal{B}$ is the neighborhood partition of G. In each step, the *Merge-Heuristic* searches the best partition that can be obtained from the current one by merging two blocks. Algorithm 2 shows the pseudocode of the Merge-Heuristic, herein the function $\text{merge}_{\mathcal{B}}(B_i, B_j)$ returns the partition where the two blocks B_i and B_j of $\mathcal{B}$ are merged, that is, $\text{merge}_{\mathcal{B}}(B_i, B_j) := (\mathcal{B} \setminus \{B_i, B_j\}) \cup \{B_i \cup B_j\}$. This process is repeated until the partition consists of t blocks and the cost ω of the final partition is returned.

Algorithm 2 Method `Merge-Heuristic`.

1: **Input** Instance $I = (G, t)$
2: **Output** upper bound for the cost of a solution of I
3: $\mathcal{B} \leftarrow$ neighborhood partition of G
4: **while** $|\mathcal{B}| > t$ **do**
5: find $B_i \in \mathcal{B}$ and $B_j \in \mathcal{B}$ with $i \neq j$ such that $\omega(\text{merge}_{\mathcal{B}}(B_i, B_j))$ is minimal
6: $\mathcal{B} \leftarrow \text{merge}_{\mathcal{B}}(B_i, B_j)$
7: **return** $\omega(\mathcal{B})$

Now we discuss the running time. We use a matrix where the number of edges between two blocks are stored. The matrix for the initial partition can be computed in $\mathcal{O}(m + |\mathcal{B}|^2)$ time. In each iteration of the while-loop the algorithm computes the cost increase for each block pair. In total, we compute the cost increase for $\sum_{i=t}^{|\mathcal{B}|} \binom{i}{2} \in \mathcal{O}(|\mathcal{B}|^3)$ merges. Since several summands of $\omega(\mathcal{B})$ are unaffected by a merge, we only have to update costs for pairs that contain the merged block. Since the merged block is involved in $\mathcal{O}(|\mathcal{B}|)$ pairs, we can compute the cost increase of a merge and update the matrix after performing a merge in $\mathcal{O}(|\mathcal{B}|)$ time. Therefore, the total running time of the while loop is $\mathcal{O}(|\mathcal{B}|^4)$. Note that the cost of the current partition can be stored in a variable and can be updated according to the cost increase of a merge. Overall, the Merge-Heuristic runs in $\mathcal{O}(|\mathcal{B}|^4 + m)$ time.

Split-Heuristic. This heuristic also starts with the neighborhood partition and decreases the number of blocks by greedily choosing some block which *splits* and whose vertices are then added to other blocks. To estimate the cost increase incurred by splitting a block, we define a function $\tau_{\mathcal{B}}$ which describes the cost increase when a vertex is added to a block. For example, let $\mathcal{B}$ be the current block partition and we want to compute the cost increase for a vertex $v \in B_v$ by putting v in B_i. For a naive approach, let $\mathcal{B}' := \mathcal{B} \setminus \{B_v\}$ be the partition without B_v and let $\mathcal{B}'' := (\mathcal{B}' \setminus \{B_i\}) \cup \{B_i \cup \{v\}\}$ be the partition without B_v where B_i is augmented by v. The cost difference is computed by $\omega(\mathcal{B}'') - \omega(\mathcal{B}')$. Due to canceling of terms where the addition does not affect ω_1 or ω_2, we can simplify the term to

$$\tau_{\mathcal{B}'}(B_i, v) := \omega_1(B_i \cup \{v\}) - \omega_1(B_i) + \sum_{B_i \neq B_j \in \mathcal{B}'} \omega_2(B_i \cup \{v\}, B_j) - \omega_2(B_i, B_j).$$

The minimal cost increase for a vertex $v \in B_v$ is computed by $\min_{B_i \in \mathcal{B}'} \tau_{\mathcal{B}'}(B_i, v)$.

To determine the block which should be split, the algorithm searches a block B^* such that the sum of the minimum cost increase over all vertices of B^* is minimal.

During the split process, the chosen block B^* will first be removed from the partition. Then, each vertex of B^* will be put sequentially in another block such that the cost increase is minimal. After the distribution of the vertices, one step is complete. The Split-Heuristic is shown in Algorithm 3.

Note that the order of the vertices during the splitting process can affect the distribution. Furthermore, the precomputed cost for such a block B^* of the block partition $\mathcal{B}$ is not always the actual cost after the split. This is due to the fact that the cost increase for each vertex v is computed with the partition $\mathcal{B} \setminus \{B^*\}$. The other vertices of $B^* \setminus \{v\}$ are not considered in $\tau_{\mathcal{B} \setminus \{B^*\}}(B_i, v)$ for some $B_i \in \mathcal{B} \setminus \{B^*\}$. Note that the actual cost of a split is never smaller than the estimation.

Now we discuss the running time. As in the Merge-Heuristic, we use the adjacency matrix and an additional matrix for the number of edges from each vertex to each block. With these matrices, we can compute $\tau_{\mathcal{B}}$ in $\mathcal{O}(|\mathcal{B}|)$ time. Therefore, we need $\mathcal{O}(|\mathcal{B}|^2)$ time to find the

Algorithm 3 Method `Split-Heuristic`.

1: **Input** Instance $I = (G, t)$
2: **Output** upper bound for the cost of a solution of I
3: $\mathcal{B} \leftarrow$ neighborhood partition of G
4: **while** $|\mathcal{B}| > t$ **do**
5: $\quad B_v \leftarrow \underset{B \in \mathcal{B}}{\text{argmin}}(\sum_{v \in B} \min_{B' \in \mathcal{B} \setminus B} \tau_{\mathcal{B} \setminus B}(B', v))$
6: $\quad \mathcal{B} \leftarrow \mathcal{B} \setminus B_v$
7: $\quad$ **for all** $v \in B_v$ **do**
8: $\quad\quad B^* \leftarrow \underset{B' \in \mathcal{B}}{\text{argmin}}\, \tau_{\mathcal{B} \setminus \{B'\} \cup \{B' \cup \{v\}\}}(B', v)$
9: $\quad\quad \mathcal{B} \leftarrow \mathcal{B} \setminus \{B^*\} \cup \{B^* \cup \{v\}\}$
10: **return** $\omega(\mathcal{B})$

block with the minimal cost increase for one vertex. In Line 5 the algorithm iterates over every block and computes its cost increase. Since every vertex is considered exactly once, this takes $\mathcal{O}(n|\mathcal{B}|^2)$ time. Afterwards, the split process starts. The determination of the best block is repeated for each vertex of this block and the vertex will be moved to another block. The latter part demands an update for both matrices. This update requires $\mathcal{O}(n)$ time since only the entries of adjacent vertices and their blocks need to be updated. Therefore, Line 7 can be computed in $\mathcal{O}(n(|\mathcal{B}|^2 + n))$. Hence, for the while-loop the running time is $\mathcal{O}(|\mathcal{B}|(n|\mathcal{B}|^2 + n(|\mathcal{B}|^2 + n))) = \mathcal{O}(|\mathcal{B}|^3 n + |\mathcal{B}|n^2)$.

Local Search. Our local search algorithm receives the graph and a block partition with t blocks as input. The algorithm tries to improve the solution by small changes until a locally optimal solution is found. The following three kinds of changes are considered. The first kind of change allows to move a vertex from one block and to another block. The second kind of change exchanges the blocks of two vertices. The third kind of change removes all vertices of the same neighborhood class from one block and puts these vertices to another block. Each of these changes is applied only if it reduces the total cost. The first two kinds of changes are the same changes local search approach TEA by Batagelj et al. [1]. We use our local search to improve the partitions obtained by the greedy heuristics.

5 Experimental Evaluation

In this section, we compare our algorithms with each other and with an adapted ILP-formulation of Dabkowski et al. [8] which is also described in Section 5.1. Furthermore, we discuss the quality of our heuristics. Finally, we analyze the structure of an optimal Blockmodeling solution with $t = 4$ for Zachary's Karate Club graph [26].

5.1 Matrix-Based ILP

In this section, we recall the ILP of Dabkowski et al. [8]. This ILP calculates for a specific blockmatrix B how well the input graph $G = (V, E)$ fits B. Naively, to find the optimal blockmatrix we need to solve an integer linear program for every quadratic matrix with at most t columns. Fortunately, it is possible to exploit isomorphism properties to reduce the number of matrices to consider.

Now we describe the formulation in detail. For each vertex pair $\{i, j\} \in \binom{V}{2}$, the constant $s_{i,j}$ equals 1 if $\{i, j\} \in E$ and 0, otherwise. Furthermore, the constant $b_{p,\ell}$ corresponds to the entry in the pth row and ℓth column of B. The binary variable $x_{i,p}$ indicates whether vertex i is in block p. Obviously, every vertex should be assigned to exactly one block and each block should have at least one vertex, otherwise another blockmatrix fits this graph better. This formulation models the error for the adjacency of two vertices i and j that are assigned to the blocks p and ℓ, respectively, (that is, $x_{i,p} = x_{j,\ell} = 1$) using the term $b_{p,\ell} + s_{i,j} - 2b_{p,\ell}s_{i,j}$. Altogether, the ILP is given by:

$$
\begin{aligned}
\min \sum_{p\in\{1,\ldots,t\}} \sum_{\ell\in\{1,\ldots,t\}} \sum_{i\in V} \sum_{j\in V\setminus\{i\}} & x_{i,p}x_{j,\ell}(b_{p,\ell} + s_{i,j} - 2b_{p,\ell}s_{i,j}) & \\
\text{s.t.} \sum_{p\in\{1,\ldots,t\}} & x_{i,p} = 1 & \forall i \in V, \\
\sum_{i\in\{1,\ldots,n\}} & x_{i,p} \geq 1 & \forall p \leq t, \\
& x_{i,p} \in \{0, 1\} & \forall i \in V, \forall p \leq t.
\end{aligned}
$$

Since the objective function is not linear, they introduced new variable $y_{i,j,p,\ell}$ to replace $x_{i,p}x_{j,\ell}$ for every $1 \leq p, \ell \leq t$, $i \in V$, and $j \in V \setminus \{i\}$ and added several constraints ensuring $y_{i,j,p,\ell} = x_{i,p}x_{j,\ell}$ [8].

Since Blockmodeling is defined only on undirected graphs and the ILP of Dabkowski et al. [8] is designed to work with directed graphs, we adapt their formulation slightly by omitting some variables that are redundant in the case of undirected graphs. This improves the running time and thus allows for a more fair comparison to our methods.

Let *MB-ILP* be the algorithm that searches a best blockmatrix for a given graph G and an integer t using the above ILP. MB-ILP applies the adapted ILP formulation for symmetric block matrices of increasing size and stops when all symmetric block matrices with up to t columns have been considered.

MB-ILP can be improved by adding an upper bound. Then, MB-ILP has to find a solution where the objective function is smaller than the upper bound. We use the minimum of the heuristic upper bound (as described in Section 5.2) and the best solution of any previously solved blockmatrix as the upper bound.

5.2 Running Time

The algorithms[2] are implemented in Java with OpenJDK 14.0.1. To solve the ILPs we used the Gurobi Optimizer in version 10.0.0. Each experiment was run on a single thread of an Intel(R) Xeon(R) Silver 4116 CPU 2.10GHz machine with 128GB RAM under Debian GNU/Linux 11 operating system.

For the experiments we used social networks obtained from KONECT [16]. For each graph of Table 1, we constructed four instances, one for each $t \in \{2, 3, 4, 5\}$ giving a total of 48 instances. Each algorithm had a time limit of 3 hours for each instance. The three graphs whose names include "pos" were obtained from directed and weighted graphs. In the appendix, we describe in detail how this was done.

All algorithms used the best solution of 10 runs of the Merge-Heuristic with local search improvement and 10 runs of the Split-Heuristic with local search improvement as initial upper bound. Table 1 shows for each graph G, for which values of t at least one of our algorithms

[2] `https://git.uni-jena.de/algo-engineering/blockmodeling`

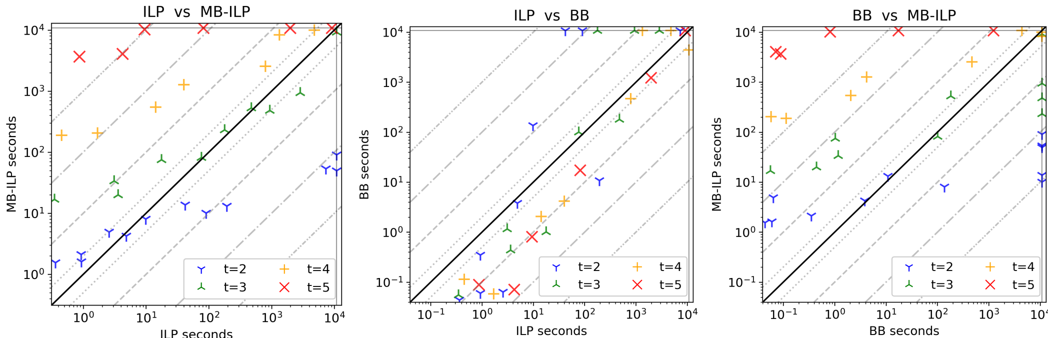


Figure 2 Each data point represents an instance, the color indicates the value of t. The gray lines mark a relative running time difference of 2, 10, 100, and 1000, respectively. The thin black lines mark the time limit. Points on these lines are not solved by the corresponding algorithm.

Table 1 Overview of the graphs used for the running time evaluation. Numbers in the middle columns denote the discovery of the optimal solution size for $t \in \{2, 3, 4, 5\}$ for each graph. Numbers in the last columns denote the maximal t the corresponding algorithm solved within the time limit.

Name	n	m	k_2	k_3	k_4	k_5	BB	ILP	MB-ILP
Highland Tribes pos	16	29	18	12	9	8	5	5	5
Kangaroo	17	91	20	16	9	8	5	5	5
Crisis in a Cloister pos	18	26	22	19	16	13	5	5	5
Taro Exchange	22	39	35	31	27	24	5	5	4
Zebra	27	111	32	26	20	17	5	5	4
Dutch College pos	32	87	61	51	45	41	2	5	4
Karate Club	34	78	65	57	44	–	–	4	4
Chesapeake Bay	39	170	118	104	–	–	–	3	3
HIV	40	41	38	33	30	–	4	3	3
Dolphins	62	159	146	–	–	–	–	–	2
Train Bomb	64	243	185	152	–	–	–	2	3
Iceland	75	114	105	–	–	–	–	–	2

found an optimal solution, and for each algorithm this table includes the maximal t-value for which the algorithm solved the instance (G, t). The branch-and-bound algorithm (BB) solved 24 instances and both ILPs solved 32. Every algorithm solved at least one instance which could not be solved by any other algorithm. For example, BB solved (HIV, $t = 4$), our ILP solved (Dutch College, 5) and MB-ILP solved (Iceland, 2). Altogether, for 36 out of the 48 instances at least one of these algorithms found an optimal solution.

Figure 2 shows a running time comparison between BB, our ILP, and MB-ILP and Figure 5 shows how many instances each algorithm solved depending on the time. While BB solved many instances faster than both ILPs, the ILPs solved more instances in total. As expected, the running time of MB-ILP depends very strongly on t; compared to our algorithms, MB-ILP solved more instances with $t \in \{2, 3\}$ and less with $t \in \{4, 5\}$, even if the optimal solution size is small. For larger t, our ILP is faster than MB-ILP by a factor up to more than 1000. In our opinion, the worse running time for $t = 2$ is not critical, since this special case can be solved much faster with tailored algorithms.

5.3 Heuristics

Now we evaluate the quality of Merge-Heuristic and Split-Heuristic with subsequent local search compared to optimal solutions. In particular, we examine the dependence of the relative error of the best found heuristic solution to the number of repetitions of these

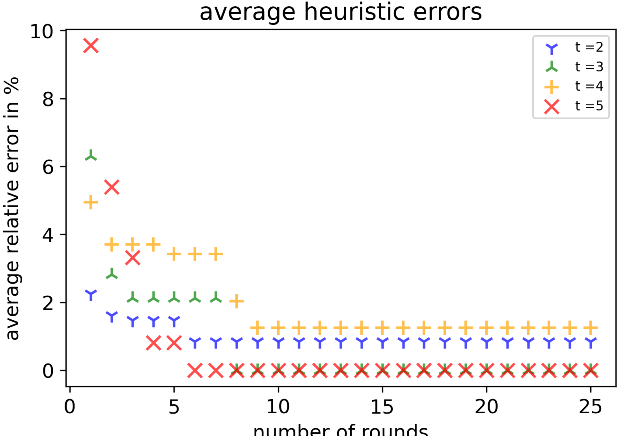


Figure 3 Relative difference between optimal solutions and the heuristics after r repetitions. One repetition includes a run of Merge-Heuristic and Split-Heuristic, each with subsequent local search.

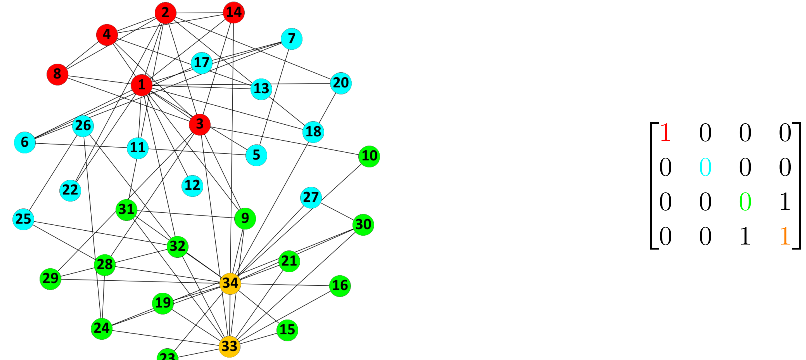


Figure 4 The Karate Club graph and its blockmatrix of an optimal clustering for $t = 4$; the vertex colors indicate the block in the optimal computed blockmodel.

heuristics. The maximum number of repetitions was set to 25. The optimal solution size was found for 32 of 36 instances. In the four cases where the optimal solution was not found, the relative difference was 1.37 % (Dolphins, $t = 2$), 7.89 % (HIV, $t = 2$), 10% (HIV, $t = 4$), and 0.95 % (Iceland, $t = 2$). The relative errors, averaged over all instances, are shown in Figure 3. After 9 rounds no progress is made. In preliminary experiments we could not deduce that one heuristic is in general better than the other. Therefore, we used both for the computation of the upper bound.

5.4 Karate Club

We now discuss one optimal blockmodel of the well-known Karate Club [26] for $t = 4$. This graph consists of members of a karate club. An edge represents an interaction between two actors outside the karate club. The club split into two new clubs, headed by the members 1 and 34, respectively. The clustering and the blockmatrix of an optimal solution for $t = 4$ is shown in Figure 4. The four blocks of the computed blockmodel correspond approximately to the split into two clubs: The red and cyan blocks correspond to the new club headed by vertex 1, the orange and green blocks correspond to the new club headed by 34. The orange

and red blocks (which contain the new leaders 1 and 34, respectively) are the two clique blocks and can be interpreted as the two cores of the corresponding new clubs. The orange and green blocks form together a dense split graph, the red and cyan clusters form together a sparse split graph. Hence, the blockmodel recovers a core/periphery structure within the clubs. In these terms, the vertices which are in a block that does not correspond to their new club (9, 25, 26, and 27) are periphery vertices. In other words, the cores are correctly separated. Finally, note that there is another optimal solution that adds 27 to the green cluster. This solution deletes fewer edges; it could be advantageous to favor such optimal solutions, to avoid producing too many isolated vertices.

6 Conclusion

We presented a new formulation of exact exploratory blockmodeling in undirected networks as graph-modification problem and developed exact and heuristic algorithms for this approach. Our algorithms are competitive with previous state-of-the-art exact approaches in terms of running times. More crucially, our algorithms enable solving the blockmodeling problem for larger values of t for which previous approaches based on enumerating all candidate blockmatrices become prohibitively slow.

There are many opportunities for future work. First, further improvements of our algorithms are desirable and likely possible. The most promising direction here seems to be the development of better lower bounds. Further extending the range of tractable instances could then allow for an empirical study of the quality of optimal solutions, beyond the anecdotical evidence discussed here. Moreover, an adaption to directed networks seems promising. Finally, one could extend our formulation also to blockmodeling with more complicated objective functions such as the one of Reichardt and White [21].

References

1. Vladimir Batagelj, Anuška Ferligoj, and Patrick Doreian. Direct and indirect methods for structural equivalence. *Social Networks*, 14(1):63–90, 1992. Special Issue on Blockmodels.
2. Stephen P. Borgatti and Martin G. Everett. Models of core/periphery structures. *Social Networks*, 21(4):375–395, 2000.
3. Stephen P. Borgatti, Martin G. Everett, and Jeffrey C. Johnson. *Analyzing social networks*. SAGE, 2013.
4. Sharon Bruckner, Falk Hüffner, and Christian Komusiewicz. A graph modification approach for finding core-periphery structures in protein interaction networks. *Algorithms for Molecular Biology*, 10:16, 2015.
5. Michael J. Brusco and Douglas Steinley. Integer programs for one- and two-mode blockmodeling based on prespecified image matrices for structural and regular equivalence. *Journal of Mathematical Psychology*, 53(6):577–585, 2009.
6. Jeffrey Chan, Wei Liu, Andrey Kan, Christopher Leckie, James Bailey, and Kotagiri Ramamohanarao. Discovering latent blockmodels in sparse and noisy graphs using non-negative matrix factorisation. In *22nd ACM International Conference on Information and Knowledge Management (CIKM '13)*, pages 811–816. ACM, 2013.
7. Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.
8. Matthew Dabkowski, Neng Fan, and Ronald L. Breiger. Exploratory blockmodeling for one-mode, unsigned, deterministic networks using integer programming and structural equivalence. *Social Networks*, 47:93–106, 2016.
9. Peter Damaschke and Olof Mogren. Editing simple graphs. *Journal of Graph Algorithms and Applications*, 18(4):557–576, 2014.

10 Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013.

11 Michael R. Fellows, Jiong Guo, Christian Komusiewicz, Rolf Niedermeier, and Johannes Uhlmann. Graph-based data clustering with overlaps. *Discrete Optimization*, 8(1):2–17, 2011.

12 Michel Habib and Christophe Paul. A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, 4(1):41–59, 2010.

13 Falk Hüffner, Christian Komusiewicz, and André Nichterlein. Editing graphs into few cliques: Complexity, approximation, and kernelization schemes. In *Proceedings of the 14th International Symposium on Algorithms and Data Structures (WADS '15)*, volume 9214 of *Lecture Notes in Computer Science*, pages 410–421. Springer, 2015.

14 Alan Jessop. Blockmodels with maximum concentration. *European Journal of Operational Research*, 148(1):56–64, 2003.

15 Ivan Kovác, Ivana Selecéniová, and Monika Steinová. On the clique editing problem. In *Proceedings of the 39th International Symposium on Mathematical Foundations of Computer Science (MFCS '14)*, volume 8635 of *Lecture Notes in Computer Science*, pages 469–480. Springer, 2014.

16 Jérôme Kunegis. KONECT: the koblenz network collection. In *Proceedings of the 22nd International World Wide Web Conference (WWW '13)*, pages 1343–1350. International World Wide Web Conferences Steering Committee / ACM, 2013.

17 Michael Lampis. Algorithmic meta-theorems for restrictions of treewidth. *Algorithmica*, 64(1):19–37, 2012.

18 Ross M. McConnell and Jeremy P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3):189–241, 1999.

19 Les G. Proll. ILP approaches to the blockmodel problem. *European Journal of Operational Research*, 177(2):840–850, 2007.

20 K. E. Read. Cultures of the central highlands, new guinea. *Southwestern Journal of Anthropology*, 10(1):1–43, 1954.

21 Jörg Reichardt and Douglas R White. Role models for complex networks. *The European Physical Journal B*, 60(2):217–224, 2007.

22 Samuel F Sampson. *Crisis in a cloister.* PhD thesis, Ph. D. Thesis. Cornell University, Ithaca, 1969.

23 Ron Shamir, Roded Sharan, and Dekel Tsur. Cluster graph modification problems. *Discrete Applied Mathematics*, 144(1-2):173–182, 2004.

24 Gerhard G Van de Bunt, Marijtje AJ Van Duijn, and Tom AB Snijders. Friendship networks through time: An actor-oriented dynamic statistical network model. *Computational & Mathematical Organization Theory*, 5(2):167–192, 1999.

25 Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

26 Wayne W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33(4):452–473, 1977.

A Supplementary Material

A.1 Deferred Proofs

A.1.1 Proof of Lemma 4

Proof. Let $I := (G = (V, E), t, k)$ be a yes-instance of BLOCKMODELING. Assume towards a contradiction, that for each optimal solution S, there is a at least one neighborhood class C of G with $|C| > 2k$ where some vertices of C is affected by S.

Let S be an optimal solution for I and let C be a neighborhood class in G with $|C| > 2k$ such that at least one vertex of C is affected. We can assume without loss of generality, that C is an independent set in G, as otherwise, we can simply consider the complement

graph $G' = (V, \binom{V}{2} \setminus E)$. Since S is a solution for I, $|S| \leq k$. Hence, there is at least one vertex $c \in C$ which is unaffected by S, that is, c has the same neighbors in G and in $G_{\text{res}} = (V, E \triangle S)$.

Let $P \subseteq V$ be the neighborhood class of c in G_{res}. We show that P is an independent set in G_{res}. Note that this is the case if $P = \{c\}$. If P contains a second vertex of C, then P is an independent set in G_{res} since c is unaffected. Otherwise, c is the only vertex of C in P and there is at least one vertex p in $P \setminus \{c\}$. Hence, if P is a clique, $\{c, p\} \in E$ which implies that for each $c' \in C \setminus \{c\}$, $\{c', p\} \in E$ since c is unaffected and C is a neighborhood class in G. Since P is a neighborhood class in G_{res}, C is an independent set in G, and c is unaffected, for each $c' \in C \setminus \{c\}$, some edge $\{p, v\}$ is in S. This would imply that $|S| > k$. Since by assumption $|S| \leq k$, P is an independent set in G_{res}. Note that this also implies that there is no edge between any vertex of C and any vertex of $P \setminus C$ in E since P is an independent set in G_{res} and c is unaffected.

Let $S' := \{\{u, v\} \in S \mid u \notin C, v \notin C\}$. We show that S' is a solution for I. Note that each vertex of C is unaffected by S'. Since by the above, P is an independent set in G_{res} with $N(p) = N(c)$ for each $p \in P \setminus C$, and there is no edge between any vertex of $P \setminus C$ and C in E, $P \cup C$ is a neighborhood class in $G_{\text{alt}} := (V, E \triangle S')$. It remains to show that $G_{\text{alt}} - (P \cup C)$ has neighborhood diversity at most $t - 1$.

Let u and v be two distinct vertices of $V \setminus (C \cup P)$ of the same neighborhood class in G_{res}. We show that u and v are in the same neighborhood class in G_{alt}. Since u and v are in the same neighborhood class in G_{res} and $|S| \leq k$, both u and v are either adjacent to each vertex of C in G or non-adjacent to each vertex of C in G. Hence, both u and v are either adjacent to each vertex of $C \cup P$ in G_{alt} or non-adjacent to each vertex of $C \cup P$ in G_{alt}. By the fact that S' contains all vertex pairs of S that do not contain any vertex of C, u and v are in the same neighborhood class in G_{alt}.

Since S is a solution for I, $G_{\text{res}} \setminus P$ has neighborhood diversity at most $t - 1$. By the above, this implies that $G_{\text{alt}} \setminus (C \cup P)$ has neighborhood diversity at most $t - 1$. Hence S' is a solution for I. Since S' is a proper subset of S, S is not an optimal solution for I, a contradiction. ◀

A.1.2 Proof of Theorem 6

Proof. For $t = 2$ the NP-hardness is shown by Lemma 5. For $t > 2$, we reduce from Blockmodeling. Let $I = (G = (V, E), k, 2)$ be an instance of Blockmodeling. Moreover, let $G' = (V', E')$ be the graph obtained from G by adding for each $i \in \{1, \ldots, t - 2\}$ a clique C_i of size $2k + 1$ to G such that each vertex of C_i has no neighbors outside of C_i in G'. Finally, we set $I' = (G', k, t)$ and show that I is a yes-instance of Blockmodeling if and only if I' is a yes-instance of Blockmodeling.

Note that for each $i \in \{1, \ldots, t - 2\}$, the clique C_i is a neighborhood class in G'. Let $S \subseteq \binom{V'}{2}$ be an optimal solution for I'. Hence, due to Lemma 4, every vertex of $V' \setminus V$ is unaffected by S. That is, every optimal solution for I' is a subset of $\binom{V}{2}$ and no vertex of V is in any neighborhood class some vertex of $V' \setminus V$.

Let $S \subseteq \binom{V}{2}$ be a set of size at most k. Hence, for each $i \in \{1, \ldots, t-2\}$, the clique C_i is a neighborhood class in $G'_{\text{res}} := (V', E' \triangle S)$. As a consequence G'_{res} has neighborhood diversity at most t if and only if $G_{\text{res}} = (V, E \triangle S)$ has neighborhood diversity at most 2. Hence, I is a yes-instance of Blockmodeling if and only if I' is a yes-instance of Blockmodeling. ◀

Table 2 Overview of the graphs and their links.

Graph	Link
Highland Tribes	http://www.konect.cc/networks/ucidata-gama/
Kangaroos	http://www.konect.cc/networks/moreno_kangaroo/
Cloister	http://www.konect.cc/networks/moreno_sampson/
Taro Exchange	http://www.konect.cc/networks/moreno_taro/
Zebra	http://www.konect.cc/networks/moreno_zebra/
Dutch College	http://www.konect.cc/networks/moreno_vdb/
Karate Club	http://www.konect.cc/networks/ucidata-zachary/
Chesapeake Bay	http://www.konect.cc/networks/dimacs10-chesapeake/
HIV	http://www.konect.cc/networks/hiv/
Dolphins	http://www.konect.cc/networks/dolphins/
Train Bomb	http://www.konect.cc/networks/moreno_train/
Iceland	http://www.konect.cc/networks/iceland/

A.1.3 Proof of Lemma 7

Proof. Let C_u and C_v be the neighborhood classes of u and v respectively, let $\mathcal{C}$ be the neighborhood partition of G and let $\mathcal{C}_{\text{rest}} := \mathcal{C} \setminus \{C_u, C_v\} = \{C_1, \dots, C_{t-2}\}$. For two vertices x and y of $V \setminus \{u, v\}$ the relation $\sim_G$ is the same as $\sim_{G'}$ because x and y have the same neighbors in G as in G'. Therefore, the vertices $\{v_1, \dots, v_{t-2}\}$ with $v_i \in C_i$ are in $t-2$ distinct neighborhood classes in G' since $v_i \nsim_G v_j$ for $1 \leq i \neq j \leq t-2$. This implies that G' has at least $t-2$ neighborhood classes.

On the other hand, both u and v might be in single-sized neighborhood classes in G'. Hence, G' has neighborhood diversity at most $t+2$. ◀

A.2 Data acquisition

We describe how we obtained undirected and unweighted graphs out of the given data sets for the three "pos"-graphs. The weight of an edge in these data sets represents how much one actor likes/dislikes the other. The undirected graph Highland Tribes [20] consists of edges of weights 1 and -1. We removed the edges with a negative weight. The remaining two graphs are Crisis in a Cloister [22] and Dutch College [24]. Both are directed and have weights $\{-1, 0, 1\}$ and $\{-1, 0, 1, 2, 3\}$ respectively. In the undirected graphs, there is an undirected

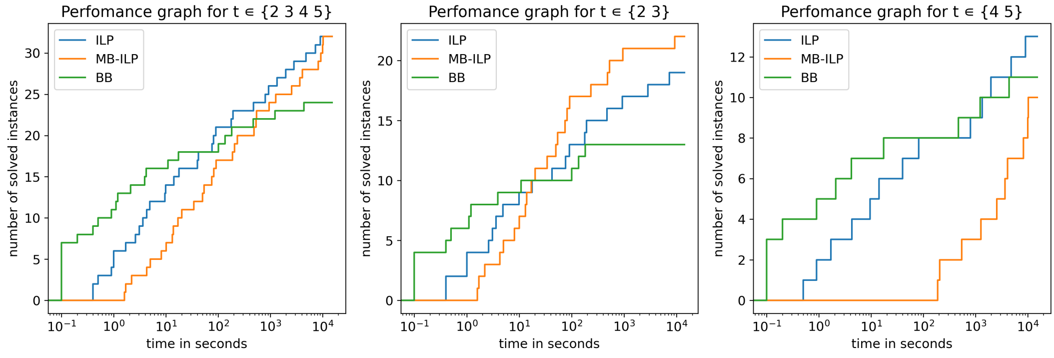


Figure 5 These graphs indicates how many instances with selected values for t each algorithm solved depending on the time.

edge $\{u, v\}$ if and only if there is an edge form u to v and an edge from v to u, the sum of weights of these edges is at least 2, and none of the weights of these edges is negative. The Dutch College data consists of 7 graphs among the same actors. For our experiment we used the graph (timestamp: 924217200) with the most edges obtained by the above-mentioned method.

FREIGHT: Fast Streaming Hypergraph Partitioning

Kamal Eyubov
Universität Heidelberg, Germany

Marcelo Fonseca Faraj
Universität Heidelberg, Germany

Christian Schulz
Universität Heidelberg, Germany

Abstract

Partitioning the vertices of a (hyper)graph into k roughly balanced blocks such that few (hyper)edges run between blocks is a key problem for large-scale distributed processing. A current trend for partitioning huge (hyper)graphs using low computational resources are streaming algorithms. In this work, we propose FREIGHT: a Fast stREamInG Hypergraph parTitioning algorithm which is an adaptation of the widely-known graph-based algorithm Fennel. By using an efficient data structure, we make the overall running of FREIGHT linearly dependent on the pin-count of the hypergraph and the memory consumption linearly dependent on the numbers of nets and blocks. The results of our extensive experimentation showcase the promising performance of FREIGHT as a highly efficient and effective solution for streaming hypergraph partitioning. Our algorithm demonstrates competitive running time with the Hashing algorithm, with a difference of a maximum factor of four observed on three fourths of the instances. Significantly, our findings highlight the superiority of FREIGHT over all existing (buffered) streaming algorithms and even the in-memory algorithm HYPE, with respect to both cut-net and connectivity measures. This indicates that our proposed algorithm is a promising hypergraph partitioning tool to tackle the challenge posed by large-scale and dynamic data processing.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases Hypergraph partitioning, graph partitioning, edge partitioning, streaming

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.15

Related Version *Full Version*: https://arxiv.org/pdf/2302.06259.pdf

Funding We acknowledge support by DFG grant SCHU 2567/5-1.

1 Introduction

Graphs are ubiquitous in nature and can be used to represent a wide variety of phenomena such as road networks, dependencies in databases, communications in distributed algorithms, interactions in social networks, and so forth. Nevertheless, phenomena where interactions between entities are not necessarily pairwise are more adequately modeled by hypergraphs, which can capture higher-order interactions [23]. With the massive proliferation of data, processing large-scale (hyper)graphs on distributed systems and databases becomes a necessity for a wide range of applications. When processing a (hyper)graph in parallel, k processors operate on distinct portions of the (hyper)graph while communicating to one another through message-passing. To make the parallel processing efficient, an important preprocessing step consists of partitioning the vertices of the (hyper)graph into k roughly balanced blocks such that few (hyper)edges run between blocks. (Hyper)graph partitioning is NP-hard [16] and there can be no approximation algorithm with a constant ratio for general (hyper)graphs [8]. Thus, heuristics are used in practice. A current trend for partitioning huge (hyper)graphs quickly and using low computational resources are streaming algorithms [36, 5, 20, 13, 14, 25, 19, 3, 35].

© Kamal Eyubov, Marcelo Fonseca Faraj, and Christian Schulz;
licensed under Creative Commons License CC-BY 4.0

The most popular streaming approach in literature is the one-pass model [1], where vertices arrive one at a time including their (hyper)edges and then have to be permanently assigned to blocks. In the domain of graphs, most algorithms are either very fast but do not care for solution quality at all (such as `Hashing` [34]), or are still fast, but much slower and capable of computing significantly better solutions than just random assignments (such as such `Fennel` [36]). Recently, the gap between these groups of algorithms has been closed by a streaming multi-section algorithm [14] which is up to two orders of magnitude faster than `Fennel` while cutting only 5% more edges than it on average. In the domain of hypergraphs, there is a similar gap that has not yet been closed. In particular, there is the same trivial `Hashing` -based algorithm on one side, and more sophisticated and expensive algorithms [3, 35] on the other side.

In this work, we propose `FREIGHT`: a Fast stREamInG Hypergraph parTitioning algorithm that can optimize for the cut-net as well as the connectivity metric. By using an efficient data structure, we make the overall running time of `FREIGHT` linearly dependent on the pin-count of the hypergraph and the memory consumption linearly dependent on the numbers of nets and blocks. Our proposed algorithm demonstrates remarkable efficiency, with a running time comparable to the `Hashing` algorithm and a maximum discrepancy of only four in three quarters of the instances. Importantly, our study establishes the superiority of `FREIGHT` over all current (buffered) streaming algorithms and even the in-memory algorithm `HYPE`, in both cut-net and connectivity measures. This shows the potential of our algorithm as a valuable tool for partitioning hypergraphs in the context of large and constantly changing data processing environments.

2 Preliminaries

2.1 Basic Concepts

Hypergraphs and Graphs. Let $H = (V = \{0, \ldots, n-1\}, E)$ be an *undirected hypergraph* with no multiple or self hyperedges, with $n = |V|$ vertices and $m = |E|$ hyperedges (or *nets*). A net is defined as a subset of V. The vertices that compose a net are called *pins*. A vertex $v \in V$ is *incident* to a net $e \in E$ if $v \in e$. Let $c : V \to \mathbb{R}_{\geq 0}$ be a vertex-weight function, and let $\omega : E \to \mathbb{R}_{>0}$ be a net-weight function. 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 $I(v)$ be the set of incident nets of v, let $d(v) := |I(v)|$ be the *degree* of v, let $d_w(v) := w(I(v))$ be the *weighted degree* of v, and let Δ be the maximum degree of H. We generalize the notations $d(.)$ and $d_w(.)$ to sets, such that $d(V') = \sum_{v \in V'} d(v)$ and $d_w(V') = \sum_{v \in V'} d_w(v)$. Two vertices are *adjacent* if both are incident to the same net. Let the number of pins $|e|$ in a net e be the *size* of e, let $\xi = \max_{e \in E}\{|e|\}$ be the maximum size of a net in H.

Let $G = (V = \{0, \ldots, n-1\}, E)$ be an *undirected graph* with no multiple or self edges, such that $n = |V|$, $m = |E|$. Let $c : V \to \mathbb{R}_{\geq 0}$ be a vertex-weight function, and let $\omega : E \to \mathbb{R}_{>0}$ be an edge-weight function. 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 $N(v) = \{u : \{v, u\} \in E\}$ denote the neighbors of 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. Let $d(v)$ be the degree of vertex v and Δ be the maximum degree of G.

Partitioning. The *(hyper)graph partitioning* problem consists of assigning each vertex of a (hyper)graph to exactly one of k distinct *blocks* respecting a balancing constraint in order to minimize the weight of the (hyper)edges running between the blocks, i.e., the edge-cut

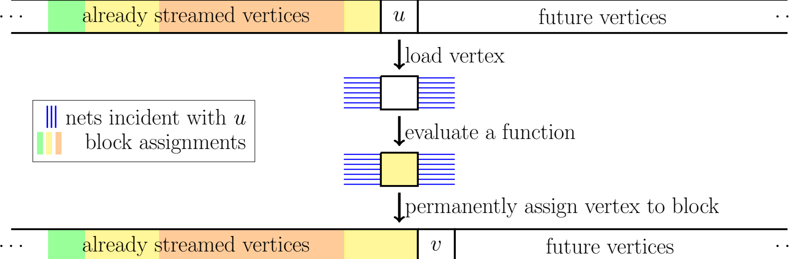


Figure 1 Typical layout of streaming algorithm for hypergraph partitioning.

(resp. cut-net). More precisely, it partitions V into k blocks $V_1,\ldots,V_k$ (i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$), which is called a *k-partition* of the (hyper)graph. The *edge-cut* (resp. *cut-net*) of a k-partition consists of the total weight of the *cut edges* (resp. *cut nets*), i.e., edges (resp. nets) crossing blocks. More formally, let the edge-cut (resp. cut-net) be $\sum_{i<j} \omega(E')$, in which $E' := \big\{e \in E, \exists\{u,v\} \subseteq e : u \in V_i, v \in V_j, i \neq j\big\}$ is the *cut-set* (i.e., the set of all cut nets). The *balancing constraint* demands that the sum of vertex weights in each block does not exceed a threshold associated with some allowed *imbalance* ϵ. More specifically, $\forall i \in \{1,\ldots,k\} : c(V_i) \leq L_{\max} := \big\lceil(1+\epsilon)\frac{c(V)}{k}\big\rceil$. For each net e of a hypergraph, $\Lambda(e) := \{V_i \mid V_i \cap e \neq \emptyset\}$ denotes the *connectivity set* of e. The *connectivity* $\lambda(e)$ of a net e is the cardinality of its connectivity set, i.e., $\lambda(e) := |\Lambda(e)|$. The so-called *connectivity* metric (λ-1) is computed as $\sum_{e \in E'}(\lambda(e) - 1)\ \omega(e)$, where E' is the cut-set.

Streaming. Streaming algorithms usually follow an iterative load-compute-store logic. Our focus and the most used streaming model is the *one-pass* model. In this model, vertices of a (hyper)graph are loaded one at a time alongside with their (hyper)edges, then some logic is applied to permanently assign them to blocks, as illustrated in Figure 1. A similar sequence of operations is used to partition a stream of edges of a graph on the fly. In this case, edges of a graph are loaded one at a time alongside with their end-points, then some logic is applied to permanently assign them to blocks. This logic can be as simple as a `Hashing` function or as complex as scoring all blocks based on some objective and then assigning the vertex to the block with highest score. There are other, more sophisticated, streaming models such as the sliding window [27] and the buffered streaming [20, 13], but are beyond the scope of this work.

2.2 Related Work

There is a huge body of research on (hyper)graph partitioning. The most prominent tools to partition (hyper)graphs in memory include `PaToH` [10], `Metis` [21], `hMetis` [22], `Scotch` [28], `HYPE` [24], `KaHIP` [30], `KaMinPar` [18], `KaHyPar` [31], `Mt-KaHyPar` [17], and `mt-KaHIP` [2]. The readers are referred to [11, 9, 33] for extensive material and references. Here, we focus on the results specifically related to the scope of this paper. In particular, we provide a detailed review for the following problems based on the one-pass streaming model: hypergraph partitioning and graph vertex partitioning.

Streaming Hypergraph Partitioning. Alistarh et al. [3] propose `Min-Max`, a one-pass streaming algorithm to assign the vertices of a hypergraph to blocks. For each block, this algorithm keeps track of nets which contain pins in it. This implies a memory consumption

of $O(mk)$. When a vertex is loaded, `Min-Max` allocates it to the block containing the largest intersection with its nets while respecting a hard constraint for load balance. The authors theoretically prove that their algorithm is able to recover a hidden *co-clustering* with high probability, where a co-clustering is defined as a simultaneous clustering of vertices and hyperedges. In the experimental evaluation, `Min-Max` outperforms five intuitive streaming approaches with respect to load imbalance, while producing solutions up to five times more imbalanced than internal-memory algorithms such as `hMetis`.

Taşyaran et al. [35] propose improved versions of the algorithm `Min-Max` [3]. The authors present `Min-Max-N2P`, a modified version of `Min-Max` that stores blocks containing each net's pins instead of storing nets per block, as done in `Min-Max`. In their experiments, `Min-Max-N2P` is three orders of magnitude faster than `Min-Max` while keeping the same cut-net. The authors also introduce three algorithms with reduced memory usage compared to `Min-Max`: Min-Max-Lℓ, a modification of `Min-Max-N2P` that employs an upper-bound ℓ to limit memory consumption per net, `Min-Max-BF` which utilizes Bloom filters for membership queries, and `Min-Max-MH` that uses hashing functions to replace the connectivity information between blocks and nets. In their experiments, their three algorithms reduce the running time in comparison to `Min-Max`, especially Min-Max-Lℓ and `Min-Max-MH`, which are up to four orders of magnitude faster. On the other hand, the three algorithms generate solutions with worse cut-net than `Min-Max`, especially `Min-Max-MH`, which increases the cut-net by up to an order of magnitude. Moreover, the authors propose a technique to improve the partitioning decision in the streaming setting by including a buffer to store some vertices and their net sets. This approach operates similarly to `Min-Max-N2P`, but with the added ability to revisit buffered vertices and adjust their partition assignment based on the connectivity metric. The authors propose three algorithms using this buffered approach: `REF` that buffers every incoming vertex but only reassigns those that may improve connectivity, `REF_RLX` that buffers all vertices and reassigns all vertices in the buffer, and `REF_RLX_SV` that only buffers vertices with small net sets and reassigns all vertices in the buffer. Their experimental results show that the use of buffered approaches leads to a 5-20% improvement in partitioning quality compared to non-buffered approaches, but with a trade-off of increased runtime.

Streaming Graph Vertex Partitioning. Stanton and Kliot [34] introduced graph partitioning in the streaming model and proposed some heuristics to solve it. Their most prominent heuristic include the one-pass methods `Hashing` and *linear deterministic greedy* (`LDG`). In their experiments, `LDG` had the best overall edge-cut. In this algorithm, vertex assignments prioritize blocks containing more neighbors and use a penalty multiplier to control imbalance. Particularly, a vertex v is assigned to the block V_i that maximizes $|V_i \cap N(v)| * \lambda(i)$ with $\lambda(i)$ being a multiplicative penalty defined as $(1 - \frac{|V_i|}{L_{\max}})$. The intuition is that the penalty avoids to overload blocks that are already very heavy. In case of ties on the objective function, `LDG` assigns the vertex to the block with fewer vertices. Overall, `LDG` partitions a graph in $O(m+nk)$ time. On the other hand, `Hashing` has running time $O(n)$ but produces a poor edge-cut.

Tsourakakis et al. [36] proposed `Fennel`, a one-pass partitioning heuristic based on the widely-known clustering objective *modularity* [7]. `Fennel` assigns a vertex v to a block V_i, respecting a balancing threshold, in order to maximize an expression of type $|V_i \cap N(v)| - f(|V_i|)$, i.e., with an additive penalty. This expression is an interpolation of two properties: attraction to blocks with many neighbors and repulsion from blocks with many non-neighbors. When $f(|V_i|)$ is a constant, the expression coincides with the first property. If $f(|V_i|) = |V_i|$, the expression coincides with the second property. In particular, the authors

defined the Fennel objective with $f(|V_i|) = \alpha * \gamma * |V_i|^{\gamma-1}$, in which γ is a free parameter and $\alpha = m\frac{k^{\gamma-1}}{n^{\gamma}}$. After a parameter tuning made by the authors, Fennel uses $\gamma = \frac{3}{2}$, which provides $\alpha = \sqrt{k}\frac{m}{n^{3/2}}$. As LDG, Fennel partitions a graph in $O(m + nk)$ time.

Faraj and Schulz [14] propose a shared-memory streaming algorithm for vertex partitioning which performs recursive multi-sections on the fly. As a preliminary phase, their algorithm decomposes a k-way partitioning problem into a hierarchy containing $\lceil \log_b k \rceil$ layers of b-way partitioning subproblems. This hierarchy can either reflect the topology of a high performance system to solve a process mapping [15, 29] or be computed for an arbitrary k to solve a regular vertex partitioning. Then, an adapted version of Fennel is used to solve each of the subproblems in such a way that the whole k-partition is computed on the fly during a single pass over the graph. While producing an edge-cut around 5% lower than Fennel, their algorithm has theoretical complexity $O((m + nb) \log_b k)$ and experimentally ran up to two orders of magnitude faster than Fennel.

Besides the one-pass model, other streaming models have also been used to solve vertex partitioning. Restreaming graph partitioning has been introduced by Nishimura and Ugander [26]. In this model, multiple passes through the entire input are allowed, which enables iterative improvements. The authors proposed easily implementable restreaming versions of LDG and Fennel: ReLDG and ReFennel, respectively. Awadelkarim and Ugander [5] studied the effect of vertex ordering for streaming graph partitioning. The authors introduced the notion of *prioritized streaming*, in which (re)streamed vertices are statically or dynamically reordered based on some priority. The authors proposed a prioritized version of ReLDG Patwary et al. [27] proposed WStream, a greedy stream algorithm that keeps a sliding stream window. Jafari et al. [20] proposed a shared-memory multilevel algorithm based on a buffered streaming model. Their algorithm uses the one-pass algorithm LDG as the coarsening, initial partitioning, and the local search steps of their multilevel scheme. Faraj and Schulz [13] proposed HeiStream, a multilevel algorithm also based on a buffered streaming model. Their algorithm loads a chunk of vertices, builds a model, and then partitions this model with a traditional multilevel algorithm coupled with an extended version of the Fennel objective.

3 FREIGHT: Fast Streaming Hypergraph Partitioning

In this section, we provide a detailed explanation of our algorithmic contribution. First, we define our algorithm named FREIGHT. Next, we present the advantages and disadvantages of using two different formats for streaming hypergraphs and partitioning them using FREIGHT. Additionally, we explain how we have removed the dependency on k from the complexity of FREIGHT by implementing an efficient data structure for block sorting.

3.1 Mathematical Definition

In this section, we provide a mathematical definition for FREIGHT by expanding the idea of Fennel to the domain of hypergraphs. Recall that, assuming the vertices of a graph being streamed one-by-one, the Fennel algorithm assigns an incoming vertex v to a block V_d where d is computed as follows:

$$d = \underset{i,\ |V_i| < L_{\max}}{\operatorname{argmax}} \left\{ |V_i \cap N(v)| - \alpha * \gamma * |V_i|^{\gamma-1} \right\} \quad (1)$$

The term $-\alpha * \gamma * |V_i|^{\gamma-1}$, which penalizes block imbalance in Fennel, is directly used in FREIGHT without modification and with the same meaning. The term $|V_i \cap N(v)|$, which minimizes edge-cut in Fennel, needs to be adapted in FREIGHT to minimize the intended

metric, i.e., either cut-net or connectivity. Before explaining how this is adapted, recall that, in contrast to graph partitioning, in hypergraph partitioning the incident nets $I(v)$ of an incoming vertex v might contain nets that are already cut, i.e., with pins assigned to multiple blocks. The version of FREIGHT designed to optimize for *connectivity* accounts for already cut nets by keeping track of the block d_e to which the most recently streamed pin of each net e has been assigned. More formally, the connectivity version of FREIGHT assigns an incoming vertex v of a hypergraph to a block V_d with d given by Equation (2), where $I^i_{obj}(v) = I^i_{con}(v) = \{e \in I(v) : d_e = i\}$. On the other hand, the version of FREIGHT designed to optimize for *cut-net* ignores already cut nets, since their contribution to the overall cut-net of the hypergraph k-partition is fixed and cannot be changed anymore. More formally, the cut-net version of FREIGHT assigns an incoming vertex v of a hypergraph to a block V_d with d given by Equation (2), where $I^i_{obj}(v) = I^i_{cut}(v) = I^i_{con}(v) \setminus E'$ and E' is the set of already cut nets.

$$d = \underset{i,\ |V_i| < L_{\max}}{\operatorname{argmax}} \left\{ |I^i_{obj}(v)| - \alpha * \gamma * |V_i|^{\gamma-1} \right\} \tag{2}$$

Both configurations of FREIGHT interpolate two objectives: favoring blocks with many incident (uncut) nets and penalizing blocks with large cardinality. We briefly highlight that FREIGHT can be adapted for weighted hypergraphs. In particular, when dealing with weighted nets, the term $|I^i_{obj}(v)|$ is substituted by $\omega(I^i_{obj}(v))$. Likewise when dealing with weighted vertices, the term $-\alpha * \gamma * |V_i|^{\gamma-1}$ is substituted by $-c(v) * \alpha * \gamma * c(V_i)^{\gamma-1}$, where the weight $c(v)$ of v is used as a multiplicative factor in the penalty term.

3.2 Streaming Hypergraphs

In this section, we present and discuss the streaming model used by FREIGHT. Recall in the streaming model for graphs vertices are loaded one at a time alongside with their adjacency lists. Thus, just streaming the graph (without doing additional compuations, implies a time cost $O(m+n)$. In our model, the vertices of a hypergraph are loaded one at a time alongside with their incident nets, as illustrated in Figure 1. Our streaming model implies a time cost $O(\sum_{e \in E} |e| + n)$ just to stream the hypergraph, where $O(\sum_{e \in E} |e|)$ is the cost to stream each net e exactly $|e|$ times. FREIGHT uses $O(m+k)$ memory, with $O(m)$ being used to keep track, for each net e, of its cut/uncut status as well as the block d_e to which its most recently streamed pin was assigned. This net-tracking information, which substitutes the need to keep track of vertex assignments, is necessary for executing FREIGHT. Although FREIGHT consumes more memory than required by graph-based streaming algorithms which often use $O(n+k)$ memory, it is still far better than the $O(mk)$ worst-case memory required by the state-of-the-art algorithms for streaming hypergraph partitioning [3, 35], all of which are also based on a computational model that implies a time cost $O(\sum_{e \in E} |e| + n)$ just to stream the hypergraph.

3.3 Efficient Implementation

In this section, we describe an efficient implementation for FREIGHT. Recall that, for every vertex v that is loaded, FREIGHT uses Equation (2) to find the block with the highest score among up to k options. A simple method to accomplish this task consists of explicitly evaluating the score for each block and identifying the one with the highest score. This results in a total of $O(nk)$ evaluations, leading to an overall complexity of $O(\sum_{e \in E} |e| + nk)$. We propose an implementation that is significantly more efficient than this approach.

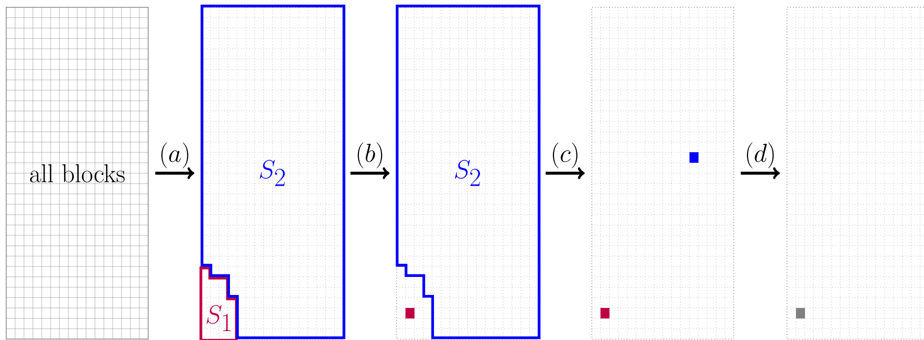


Figure 2 Illustration of the process to solve Equation (2) for an incoming vertex u with $k = 512$ blocks. (a) The k blocks are decomposed into S_1 and S_2, with $|S_1| = O(|I(u)|)$. (b) Equation (3) is explicitly solved at cost $O(|I(u)|)$. (c) Equation (4) is implicitly solved at cost $O(1)$. (d) Both solutions are then evaluated using their `FREIGHT` scores to determine the final solution for Equation (2).

For each loaded vertex v, our implementation separates the blocks V_i for which $|V_i| < L_{\max}$ into two disjoint sets, S_1 and S_2. In particular, the set S_1 comprises blocks V_i where $|I^i_{obj}(v)| > 0$, while the set S_2 comprises the remaining blocks, i.e., blocks V_i for which $|I^i_{obj}(v)| = 0$. Using the sets provided, we break down Equation (2) into Equation (3) and Equation (4), which are solved separately. The resulting solutions are compared based on their `FREIGHT` scores to ultimately find the solution for Equation (2). The overall process is illustrated in Figure 2.

$$d = \underset{i \in S_1}{\operatorname{argmax}} \left\{ |I^i_{obj}(v)| - \alpha * \gamma * |V_i|^{\gamma - 1} \right\} \tag{3}$$

$$d = \underset{i \in S_2}{\operatorname{argmax}} \left\{ |I^i_{obj}(v)| - \alpha * \gamma * |V_i|^{\gamma - 1} \right\} = \underset{i \in S_2}{\operatorname{argmin}} |V_i| \tag{4}$$

Now we explain how we solve Equation (3) and Equation (4). To solve Equation (3), we use the theoretical complexity outlined in Theorem 1 and solve it explicitly. In contrast, Equation (4) is implicitly solved by identifying the block with minimal cardinality. We use an efficient data structure to keep all blocks sorted by cardinality throughout the entire execution, which enables us to solve Equation (4) in constant time.

▶ **Theorem 1.** *Equation (3) can be solved in time $O(|I(v)|)$.*

Proof. The terms $|I^i_{obj}(v)|$ in Equation (3) can be computed by iterating through the nets of v at a cost of $O(|I(v)|)$ and determining their status as cut, unassigned, or assigned to a block. The calculation of the factors $-\alpha * \gamma * |V_i|^{\gamma-1}$ in Equation (3) can be done in time $O(|S_1|) = O(|I(v)|)$, thus completing the proof. ◀

Now we explain our data structure to keep the blocks sorted by cardinality during the whole algorithm execution. The data structure is implemented with two arrays A and B, both with k elements, and a list L. The array A stores all k blocks always in ascending order. The array B maps the index i of a block V_i to its position in A. Each element in the list L represents a bucket. Each bucket is associated with a unique block cardinality and contains the leftmost and the rightmost positions ℓ and r of the range of blocks in A which currently

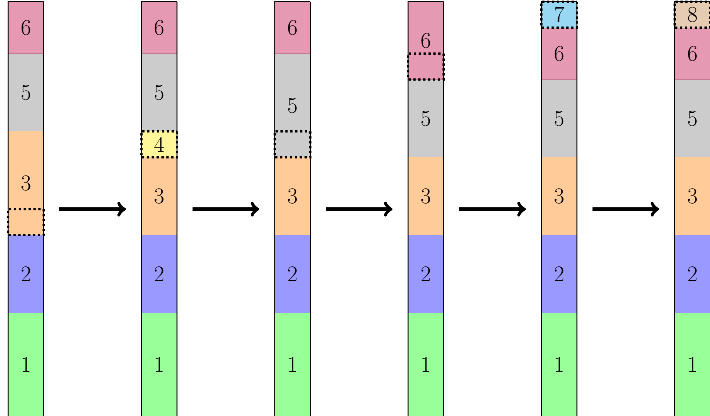


Figure 3 Illustration of our data structure used to keep the blocks sorted by cardinality throughout the execution of FREIGHT. The array A is represented as a vertical rectangle. Each region of A is covered by a unique bucket, which is represented by a unique color filling the corresponding region in A. The cardinality associated with each bucket is written in the middle of the region of A covered by it. Here we represent the behavior of the data structure when assigning vertices to the block surrounded by a dotted rectangle five times consecutively.

have this cardinality. Reciprocally, each block in A has a pointer to the unique bucket in L corresponding to its cardinality. To begin the algorithm, L is set up with a single bucket for cardinality 0 which covers the k positions of A, i.e., its paramenters ℓ and r are 1 and k, respectively. The blocks in A are sorted in any order initially, however, as each block starts with a cardinality of 0, they will be ordered by their cardinalities.

When a vertex is assigned to a block V_d, we update our data structure as detailed in Algorithm 1 and exemplified in Figure 3. We describe Algorithm 1 in detail now. In line 1, we find the position p of V_d in A and find the bucket C associated with it. In line 2, we exchange the content of two positions in A: the position where V_d is located and the position identified by the variable r in C, which marks the rightmost block in A covered by C. This variable r is afterwards decremented in line 3 since V_d is now not covered anymore by the bucket C. In lines 4 and 5, we check if the new (increased) cardinality of V_d matches the cardinality of the block located right after it in A. If so, we associate V_d to the same bucket as it and decrement this bucket's leftmost position ℓ in line 6; Otherwise, we push a new bucket to L and match it to V_d adequately in lines 8 and 9. Finally, in line 10, we delete C in case its range $[\ell, r]$ is empty. Figure 3 shows our data structure through five consecutive executions of Algorithm 1. Theorem 2 proves the correctness of our data structure. Theorem 3 shows that, using our proposed data structure, we need time $O(1)$ to either solve Equation (4) or prove that the solution for Equation (3) solves Equation (2). Note that our data structure can only handle unweighted vertices. In case of weighted vertices, a bucket queue can be used instead of our data structure, resulting in the same overall complexity and requiring $O(k + L_{\max})$ memory, while our data structure only requires $O(k)$ memory. The overall complexity of FREIGHT, which directly follows from Theorem 1 and Theorem 3, is expressed in Corollary 4.

▶ **Theorem 2.** *Our proposed data structure keeps the blocks within array A consistently sorted in ascending order of cardinality.*

Proof. We inductively prove two claims at the same time: (a) the variables ℓ and r contained in each bucket from L respectively store the leftmost and the rightmost positions of the unique range of blocks in A which currently have this cardinality; (b) the array A contains

Algorithm 1 Increment cardinality of block V_d in the proposed data structure.

1: $p \leftarrow B_d$; $C \leftarrow A_p.bucket$;
2: $q \leftarrow C.r$; $c \leftarrow A_q.id$; $Swap(A_p, A_q)$; $Swap(B_c, B_d)$;
3: $C.r \leftarrow C.r - 1$;
4: $C' \leftarrow A_{q+1}.bucket$;
5: **if** $C.cardinality + 1 = C'.cardinality$ **then**
6: $\quad A_q.bucket \leftarrow C'$; $C'.\ell \leftarrow C'.\ell - 1$;
7: **else**
8: $\quad C'' \leftarrow NewBucket()$; $A_q.bucket \leftarrow C''$; $L \leftarrow L \cup \{C''\}$;
9: $\quad C''.cardinality \leftarrow C.cardinality + 1$; $C''.\ell \leftarrow q$; $C''.r \leftarrow q$;
10: **if** $C.r = C.\ell$ **then** $L \leftarrow L \setminus \{C\}$;

the blocks sorted in ascending order of cardinality. Both claims are trivially true at the beginning, since all blocks have cardinality 0 and L is initialized with a single bucket with $\ell = 1$ and $r = k$. Now assuming that (a) and (b) are true at some point, we show that they keep being true after Algorithm 1 is executed. Note that line 2 performs the only position exchange in A throughout the whole algorithm. As (a) is assumed, it is the case that V_d swaps positions with the rightmost block in A containing the same cardinality of V_d. Since the cardinality of V_d will be incremented by one and all blocks have integer cardinalities, this concludes the proof of (b). To prove that (a) remains true, note that the only buckets in L that are modified are C (line 3), C' (line 6), and C'' (line 9). Claim (a) remains true for C because V_d, whose cardinality will be incremented, is the only block removed from its range. Claim (a) remains true for C' because line 6 is only executed if the new cardinality of V_d equals the cardinality of C', whose current range starts right after the new position of V_d in A. Bucket C'' is only created if the new cardinality of V_d is respectively larger and smaller than the cardinalities of C and C'. Since (b) is true, then this condition only happens if there is no block in A with the same cardinality as the new cardinality of V_d. Hence, claim (a) remains true for C'', which is created covering only the position of V_d in A. ◀

▶ **Theorem 3.** *By utilizing our proposed data structure, solving Equation (4) or demonstrating that any solution for Equation (3) is also a solution for Equation (2) can be accomplished in $O(1)$ time.*

Proof. Algorithm 1 contains no loops and each command in it has a complexity of $O(1)$, thus the total cost of the algorithm is $O(1)$. Our data structure executes Algorithm 1 once for each assigned vertex, hence it costs $O(1)$ per vertex. Say we are evaluating an incoming vertex v. According to Theorem 2, the block V_d with minimum cardinality is stored in the first position of the array A, hence it can be accessed in time $O(1)$. In case $V_d \in S_2$, then d is a solution for Equation (4). On the other hand, if V_d is in S_1, the `FREIGHT` score of V_d will be larger than the `FREIGHT` score of the solution for Equation (4) by at least $|I_d(v)| > 0$. In this case, it follows that any solution for Equation (3) solves Equation (2). ◀

▶ **Corollary 4.** *The overall complexity of `FREIGHT` is $O\big(\sum_{e \in E} |e| + n\big)$.*

4 Experimental Evaluation

Setup. We performed our implementations in C++ and compiled them using gcc 11.2 with full optimization turned on (-O3 flag). Unless mentioned otherwise, all experiments are performed on a single core of a machine consisting of a sixteen-core Intel Xeon Silver 4216

processor running at 2.1 GHz, 100 GB of main memory, 16 MB of L2-Cache, and 22 MB of L3-Cache running Ubuntu 20.04.1. The machine can handle 32 threads with hyperthreading. Unless otherwise mentioned we stream (hyper)graphs directly from the internal memory to obtain clear running time comparisons. However, note that `FREIGHT` as well as most of the other used algorithms can also be run streaming the hypergraphs from hard disk.

Baselines. We compare `FREIGHT` against various state-of-the-art algorithms. In this section we will list these algorithms and explain our criteria for algorithm selection. We have implemented `Hashing` in C++, since it is a simple algorithm. It basically consists of hashing the IDs of incoming vertices into $\{1, \ldots, k\}$. The remaining algorithms were obtained either from official repositories or privately from the authors, with the exception of `Min-Max`, for which there is no official implementation available. Here, we use the `Min-Max` implementations by Taşyaran et al. [35]. All algorithms were compiled with gcc 11.2.

We run `Hashing`, `Min-Max` [3] and all its improved versions proposed by Taşyaran et al. [35]: `Min-Max-BF`, `Min-Max-N2P`, `Min-Max-L`ℓ, `Min-Max-MH`, `REF`, `REF_RLX`, and `REF_RLX_SV`. (see Section 2.2 for details on the different `Min-Max` versions), `HYPE` [24], and `PaToH` v3.3 [10]. `Hashing` is relevant because it is the simplest and fastest streaming algorithm, which gives us a lower bound for partitioning time. `Min-Max` is a current state-of-the-art for streaming hypergraph partitioning in terms of cut-net and connectivity. The improved and buffered versions of `Min-Max` proposed in [35] are relevant because some of them are orders of magnitude faster than `Min-Max` while others produce improved partitions in comparison to it. `HYPE` and `PaToH` are in-memory algorithms for hypergraph partitioning, hence they are not suitable for the streaming setting. However, we compare against them because `HYPE` is among the fastest in-memory algorithms while `PaToH` is very fast and also computes partitions with very good cut-net and connectivity. Note that KaHyPar [31] is the leading tool with respect to solution quality, however it is also much slower than `PaToH`.

Instances. We selected hypergraphs from various sources to test our algorithm. The considered hypergraphs were used for benchmark in previous works on hypergraph partitioning. Prior to each experiment, we converted all hypergraphs to the appropriate streaming formats required by each algorithm. We removed parallel and empty hyperedges and self loops, and assigned unitary weight to all vertices and hyperedges. In all experiments with streaming algorithms, we stream the hypergraphs with the natural given order of the vertices. We use a number of blocks $k \in \{512, 1024, 1536, 2048, 2560\}$ unless mentioned otherwise. We allow a fixed imbalance of 3% for all experiments (and all algorithms) since this is a frequently used value in the partitioning literature. All algorithms always generated balanced partitions, except for `HYPE` which generated highly unbalanced partitions in around 5% of its experiments.

We use the same benchmark as in [31]. This consists of 310 hypergraphs from three benchmark sets: 18 hypergraphs from the ISPD98 Circuit Benchmark Suite [4], 192 hypergraphs based on the University of Florida Sparse Matrix Collection [12], and 100 instances from the international SAT Competition 2014 [6]. The SAT instances were converted into hypergraphs by mapping each boolean variable and its complement to a vertex and each clause to a net. From the Sparse Matrix Collection, one matrix was selected for each application area that had between 10 000 and 10 000 000 columns. The matrices were converted into hypergraphs using the row-net model, in which each row is treated as a net and each column as a vertex.

Methodology. Depending on the focus of the experiment, we measure running time, cut-net, and-or connectivity. We perform 5 repetitions per algorithm and instance using random seeds for non-deterministic algorithms, and calculate the arithmetic average

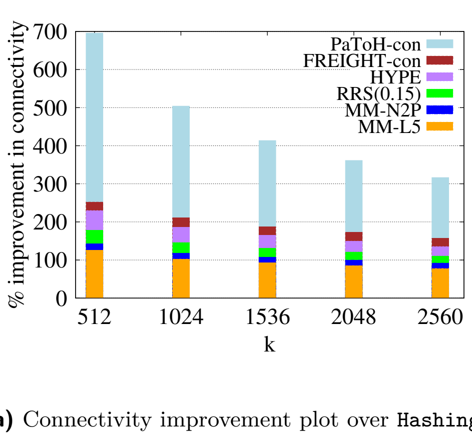
of the computed objective function and running time per instance. When further averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the *final score*.

Given a result of an algorithm A, we express its value σ_A (which can be objective or running time) as *improvement* over an algorithm B, computed as $\left(\frac{\sigma_B}{\sigma_A} - 1\right) * 100\%$; We also use *performance profiles* to represent results. They relate the running time (quality) of a group of algorithms to the fastest (best) one on a per-instance basis (rather than grouped by k). The x-axis shows a factor τ while the y-axis shows the percentage of instances for which A has up to τ times the running time (quality) of the fastest (best) algorithm. Bar charts and boxplots are also employed to represent our findings. We use bar charts to visualize the average value of an objective function in relation to k, where each algorithm is represented by vertical bars of a given color with origin on the x-axis. The bars for every value of k have a common origin and are arranged in terms of their height, allowing all heights to be visible. We use boxplots to give a clear picture of the dataset distribution by displaying the minimum, maximum, median, first and third quartiles, while disregarding outliers.

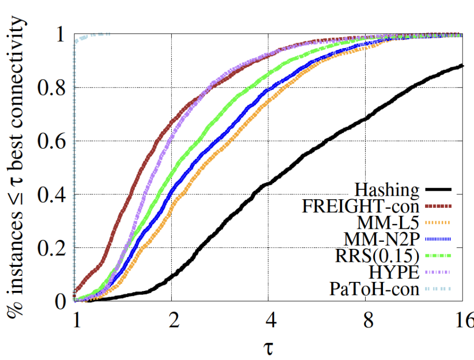
4.1 Results

In this section, we show experiments in which we compare `FREIGHT` against the current state-of-the-art of streaming hypergraph partitioning. As already mentioned, we also use two internal-memory algorithms [24, 10] as more general baselines for comparison. We focus our experimental evaluation on the comparison of solution quality and running time. Observe that `PaToH` and `FREIGHT` have distinct versions designed to optimize for each quality metric (i.e., connectivity and cut-net). For a meaningful comparison, we only take into account the relevant version when dealing with each quality metric, however, both versions are still considered for running time comparisons. To differentiate between the versions, suffixes `-con` and `-cut` are added to represent the connectivity-optimized and cut-net versions respectively. For clarity, we refrain from discussing state-of-the-art streaming algorithms that are *dominated* by another algorithm. We define a dominated algorithm as one that has worse running time compared to another without offering a superior solution quality in return, or vice-versa. In particular, we leave out `Min-Max` and `Min-Max-BF` since they are dominated by `Min-Max-N2P`, which is referred to as `MM-N2P` hereafter. Similarly, we omit `Min-Max-MH` because it is dominated by `Hashing`. We use a buffer size of 15% for testing the buffered algorithms `REF`, `REF_RLX`, and `REF_RLX_SV`, following the best results outlined in [35]. We omit the first two of them since they are dominated by the latter one, which is referred to as `RRS(0.15)` from now on. Since `Min-Max-L`ℓ is not dominated by any other algorithm, we exhibit its results with $\ell = 5$, as seen in the best results in [35], and we refer to it as `MM-L5` from this point.

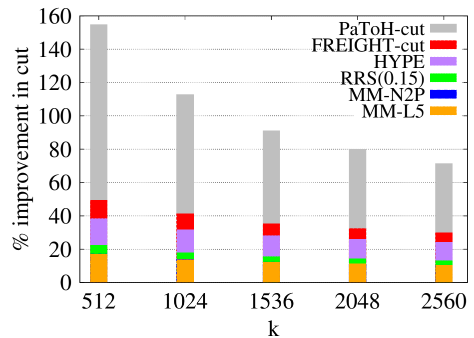
Connectivity. We start by looking at the connectivity metric. In Figure 4a, we plot the average connectivity improvement over `Hashing` for each value of k. `PaToH-con` produces the best connectivity on average, yielding an average improvement of 443% when compared to `Hashing`. This is in line with previous works in the area of (hyper)graph partitioning, i.e. streaming algorithms typically compute worse solutions than internal memory algorithms, which have access to the whole graph. `FREIGHT-con` is found to be the second best algorithm in terms of connectivity, outperforming both the internal memory algorithm `HYPE` and the buffered streaming algorithm `RRS(0.15)`. On average, these three algorithms improve 194%, 171%, and 136% over `Hashing`, respectively. Finally, `MM-N2P` and `MM-L5` compute solutions which improve 111% and 96% over `Hashing` on average, respectively. In direct



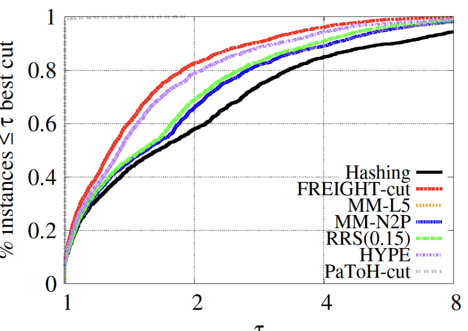
(a) Connectivity improvement plot over `Hashing`.



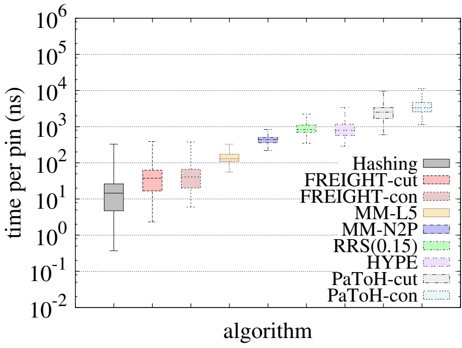
(b) Connectivity performance profile.



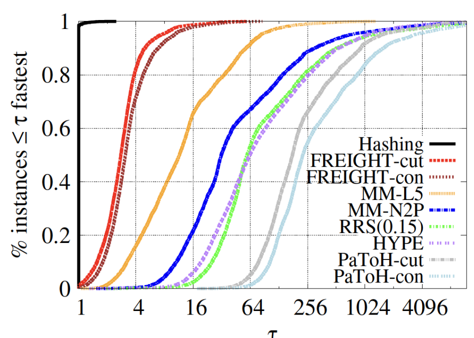
(c) Cut-net improvement plot over `Hashing`.



(d) Cut-net performance profile.



(e) Running time boxplots.



(f) Running time performance profile.

Figure 4 Comparison against the state-of-the-art streaming algorithms for hypergraph partitioning. We show performance profiles, improvement plots over `Hashing`, and boxplots. Note that `PaToH-con`, `PaToH-cut`, and `Hashing` align almost perfectly with the y-axis in Figures 4b, 4d, and 4f, respectively. Also the curves and bars of `MM-N2P` and `MM-L5` roughly overlap with one another in Figure 4d and Figure 4c.

comparison, FREIGHT-con shows average connectivity improvements of 8%, 24%, 39%, and 50% over HYPE, RRS(0.15), MM-N2P, and MM-L5, respectively. Note that each algorithm retains its relative ranking in terms of average connectivity over all values of k.

In Figure 4b, we plot connectivity performance profiles across all experiments. PaToH-con produces the best overall connectivity for 96.4% of the instances, while FREIGHT-con produces the best connectivity for 3.1% of the instances and no other algorithm computes the best connectivity for more than 0.35% of the instances. The connectivity produced by FREIGHT-con, HYPE, RRS(0.15), MM-N2P, MM-L5, and Hashing are within a factor 2 of the best found connectivity for 67%, 61%, 47%, 41%, 34%, and 9% of the instances, respectively. In summary, FREIGHT-con produces the best connectivity among (buffered) streaming competitors, outperforming even in-memory algorithm HYPE.

Cut-Net. Next we examine at the cut-net metric. In Figure 4c, we plot the cut-net improvement over Hashing. PaToH-cut produces the best overall cut-net, with an average improvement of 100% compared to Hashing. FREIGHT-cut is found to be the second best algorithm with respect to cut-net, superior to internal-memory algorithm HYPE and buffered streaming algorithm RRS(0.15). These three algorithms improve connectivity over Hashing by 37%, 30%, and 17% respectively. Finally, both MM-N2P and MM-L5 improve connectivity by 13% on average over Hashing. In direct comparison, FREIGHT-cut shows average connectivity improvements of 6%, 18%, 22%, and 22% over HYPE, RRS(0.15), MM-N2P, and MM-L5, respectively. Each algorithm preserves its relative ranking in average cut-net across all values of k.

In Figure 4d, we plot cut-net performance profiles across all experiments. In the plot, PaToH-cut produces the best overall connectivity for 98.0% of the instances, while FREIGHT-cut and HYPE produce the best cut-net for 6.8% and 5.2% of the instances and all other streaming algorithms (RRS(0.15), MM-N2P, MM-L5, and Hashing) produce the best cut-net for 4.8% of the instances. The cut-net results produced by FREIGHT-cut, HYPE, RRS(0.15), MM-N2P, MM-L5, and Hashing are within a factor 2 of the best found cut-net for 83%, 79%, 69%, 66%, 66%, and 58% of the instances, respectively. This shows that FREIGHT-cut produces the best cut-net among all (buffered) streaming competitors and even beats the in-memory algorithm HYPE.

Running Time. Now we compare the algorithms' runtime. Boxes and whiskers in Figure 4e display the distribution of the running time per pin, measured in nanoseconds, for all instances. Hashing, FREIGHT-cut, and FREIGHT-con are the three fastest algorithms, with median runtimes per pin of 15ns, 38ns, and 41ns, respectively. MM-L5, MM-N2P, HYPE, and RRS(0.15) follow with median runtimes per pin of 130ns, 437ns, 792ns, and 833ns, respectively. Lastly, the algorithms with the highest median runtime per pin are PaToH-cut and PaToH-con, with 2 516ns and 3 333ns respectively. The measured runtime per pin for both HYPE and PaToH align with values reported in prior research [32].

In Figure 4f, we show running time performance profiles. Hashing is the fastest algorithm for 98.3% of the instances, while FREIGHT-cut is the fastest one for 1.2% of the instances and no other algorithm is the fastest one for more than 0.4% of the instances. The running time of FREIGHT-cut and FREIGHT-con is within a factor 4 of that of Hashing for 82% and 72% of instances, respectively. In contrast, for only 16% of instances does this occur for MM-L5, and for less than 0.4% of instances for all other algorithms. The close running times of FREIGHT to Hashing are surprising given FREIGHT's superior solution quality compared to Hashing and all other streaming algorithms and even HYPE.

Further Comparisons. For graph vertex partitioning FREIGHT and Fennel are mathematically equivalent. However, FREIGHT exhibits a lower computational complexity of $O(m+n)$ compared to the standard implementation of Fennel, which has a complexity of $O(m+nk)$ due to evaluating all blocks for each node. To optimize its performance for this use case, we have implemented an optimized version of FREIGHT with a memory consumption of $O(n+k)$, matching that of Fennel. In our experiments, we utilized the same graphs as in [14] and tested with $k \in \{512, 1024, 1536, 2048, 2560\}$. On average, FREIGHT proves to be 109 times faster than the standard implementation of Fennel. Moreover, the performance gap is found to increase as the value of k grow, with FREIGHT reaching up to 261 times faster than Fennel in some instances.

5 Conclusion

In this work, we introduce FREIGHT, a highly efficient and effective streaming algorithm for hypergraph partitioning. Our algorithm leverages an optimized data structure, resulting in linear running time with respect to pin-count and linear memory consumption in relation to the numbers of nets and blocks. The results of our extensive experimentation demonstrate that the running time of FREIGHT is competitive with the Hashing algorithm, with a maximum difference of a factor of four observed in three fourths of the instances. Importantly, our findings indicate that FREIGHT consistently outperforms all existing (buffered) streaming algorithms and even the in-memory algorithm HYPE, with regards to both cut-net and connectivity measures. This underscores the significance of our proposed algorithm as a highly efficient and effective solution for hypergraph partitioning in the context of large-scale and dynamic data processing.

References

1. Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming graph partitioning: An experimental study. *Proc. VLDB Endow.*, 11(11):1590–1603, 2018. doi:10.14778/3236187.3236208.
2. Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. High-quality shared-memory graph partitioning. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, volume 11014 of *Lecture Notes in Computer Science*, pages 659–671. Springer, 2018. doi:10.1007/978-3-319-96983-1_47.
3. Dan Alistarh, Jennifer Iglesias, and Milan Vojnovic. Streaming min-max hypergraph partitioning. In *Advances in Neural Information Processing Systems*, pages 1900–1908, 2015. doi:10.5555/2969442.2969452.
4. Charles J. Alpert. The ISPD98 circuit benchmark suite. In Majid Sarrafzadeh, editor, *Proceedings of the 1998 International Symposium on Physical Design, ISPD 1998, Monterey, CA, USA, April 6-8, 1998*, pages 80–85. ACM, 1998. doi:10.1145/274535.274546.
5. Amel Awadelkarim and Johan Ugander. Prioritized restreaming algorithms for balanced graph partitioning. In Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash, editors, *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 1877–1887. ACM, 2020. doi:10.1145/3394486.3403239.
6. Anton Belov, Daiel Diepold, Marijn Heule, and Matti Järvisalo. The sat competition 2014. http://www.satcompetition.org/2014/, 2014.
7. Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE transactions on knowledge and data engineering*, 20(2):172–188, 2007. doi:10.1109/TKDE.2007.190689.

8 Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Inf. Process. Lett.*, 42(3):153–159, 1992. `doi:10.1016/0020-0190(92)90140-Q`.

9 Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent Advances in Graph Partitioning*, pages 117–158. Springer International Publishing, Cham, 2016. `doi:10.1007/978-3-319-49487-6_4`.

10 Ümit V. Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs). In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1479–1487. Springer, 2011. `doi:10.1007/978-0-387-09766-4_93`.

11 Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More recent advances in (hyper)graph partitioning. *ACM Computing Surveys*, 2023. `doi:doi.org/10.1145/3571808`.

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

13 Marcelo Fonseca Faraj and Christian Schulz. Buffered streaming graph partitioning. *ACM J. Exp. Algorithmics*, 27, October 2022. `doi:10.1145/3546911`.

14 Marcelo Fonseca Faraj and Christian Schulz. Recursive multi-section on the fly: Shared-memory streaming algorithms for hierarchical graph partitioning and process mapping. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 473–483, 2022. `doi:10.1109/CLUSTER51413.2022.00057`.

15 Marcelo Fonseca Faraj, Alexander van der Grinten, Henning Meyerhenke, Jesper Larsson Träff, and Christian Schulz. High-quality hierarchical process mapping. In *18th International Symposium on Experimental Algorithms, SEA*, volume 160 of *LIPIcs*, pages 4:1–4:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.SEA.2020.4`.

16 Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*, pages 47–63. ACM, 1974. `doi:10.1145/800119.803884`.

17 Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable Shared-Memory Hypergraph Partitioning. In *Proceedings of the Symposium on Algorithm Engineering and Experiments ALENEX*, pages 16–30, 2021. `doi:10.1137/1.9781611976472.2`.

18 Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep multilevel graph partitioning. In *29th Annual European Symposium on Algorithms, ESA*, volume 204 of *LIPIcs*, pages 48:1–48:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ESA.2021.48`.

19 Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Cusp: A customizable streaming edge partitioner for distributed graph analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 439–450. IEEE, 2019. `doi:10.1109/IPDPS.2019.00054`.

20 Nazanin Jafari, Oguz Selvitopi, and Cevdet Aykanat. Fast shared-memory streaming multilevel graph partitioning. *Journal of Parallel and Distributed Computing*, 147:140–151, 2021. `doi:10.1016/j.jpdc.2020.09.004`.

21 George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 35. IEEE Computer Society, 1996. `doi:10.1109/SC.1996.32`.

22 George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In Mary Jane Irwin, editor, *Proceedings of the 36th Conference on Design Automation*, pages 343–348. ACM Press, 1999. `doi:10.1145/309847.309954`.

23 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`.

24 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 (IEEE BigData)*, pages 458–467. IEEE, 2018. doi:10.1109/BigData.2018.8621968.

25 Christian Mayer, Ruben Mayer, Muhammad Adnan Tariq, Heiko Geppert, Larissa Laich, Lukas Rieger, and Kurt Rothermel. Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 685–695. IEEE, 2018. doi:10.1109/ICDCS.2018.00072.

26 Joel Nishimura and Johan Ugander. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1106–1114, 2013. doi:10.1145/2487575.2487696.

27 Md Anwarul Kaium Patwary, Saurabh Kumar Garg, and Byeong Kang. Window-based streaming graph partitioning algorithm. In *Proceedings of the Australasian Computer Science Week Multiconference, ACSW*, pages 51:1–51:10. ACM, 2019. doi:10.1145/3290688.3290711.

28 François Pellegrini and Jean Roman. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. Technical report, TR 1038-96, LaBRI, 1996. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=94b913363b57e019b8a32529b076a8d4181587ac.

29 Maria Predari, Charilaos Tzovas, Christian Schulz, and Henning Meyerhenke. An mpi-based algorithm for mapping complex networks onto hierarchical architectures. In *Euro-Par 2021: Parallel Processing - 27th International Conference on Parallel and Distributed Computing*, volume 12820 of *LNCS*, pages 167–182. Springer, 2021. doi:10.1007/978-3-030-85665-6_11.

30 Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Experimental Algorithms, 12th International Symposium, SEA*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013. doi:10.1007/978-3-642-38527-8_16.

31 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 Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX*, pages 53–67. SIAM, 2016. doi:10.1137/1.9781611974317.5.

32 Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithms (JEA)*, 2022. doi:10.1145/3529090.

33 Christian Schulz and Darren Strash. Graph partitioning: Formulations and applications to big data. In *Encyclopedia of Big Data Technologies*. Springer, 2019. doi:10.1007/978-3-319-63962-8_312-2.

34 Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230, 2012. doi:10.1145/2339530.2339722.

35 Fatih Taşyaran, Berkay Demireller, Kamer Kaya, and Bora Uçar. Streaming Hypergraph Partitioning Algorithms on Limited Memory Environments. In *HPCS 2020 - International Conference on High Performance Computing & Simulation*, pages 1–8. IEEE, 2021. URL: https://hal.archives-ouvertes.fr/hal-03182122.

36 Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342, 2014. doi:10.1145/2556195.2556213.

Arc-Flags Meet Trip-Based Public Transit Routing

Ernestine Großmann
Universität Heidelberg, Germany

Jonas Sauer
Karlsruhe Institute of Technology, Germany

Christian Schulz
Universität Heidelberg, Germany

Patrick Steil
Universität Heidelberg, Germany

Abstract

We present Arc-Flag TB, a journey planning algorithm for public transit networks which combines Trip-Based Public Transit Routing (TB) with the Arc-Flags speedup technique. Compared to previous attempts to apply Arc-Flags to public transit networks, which saw limited success, our approach uses stronger pruning rules to reduce the search space. Our experiments show that Arc-Flag TB achieves a speedup of up to two orders of magnitude over TB, offering query times of less than a millisecond even on large countrywide networks. Compared to the state-of-the-art speedup technique Trip-Based Public Transit Routing Using Condensed Search Trees (TB-CST), our algorithm achieves similar query times but requires significantly less additional memory. Other state-of-the-art algorithms which achieve even faster query times, e.g., Public Transit Labeling, require enormous memory usage. In contrast, Arc-Flag TB offers a tradeoff between query performance and memory usage due to the fact that the number of regions in the network partition required by our algorithm is a configurable parameter. We also identify a previously undiscovered issue in the transfer precomputation of TB, which causes both TB-CST and Arc-Flag TB to answer some queries incorrectly. We provide discussion on how to resolve this issue in the future. Currently, Arc-Flag TB answers 1–6% of queries incorrectly, compared to over 20% for TB-CST on some networks.

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

Keywords and phrases Public transit routing, graph algorithms, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.16

Supplementary Material *Software (Code)*: `https://github.com/TransitRouting/Arc-FlagTB`
archived at `swh:1:dir:6788b9abdd6ad1056ecdec1365b88effd6057558`

Funding We acknowledge support by DFG grants SCHU 2567/3-1 and WA 654/23-2.

Acknowledgements We want to thank Dr. Patrick Brosi for providing us with the Germany dataset and Sascha Witt for providing us with the source code for TB-CST.

1 Introduction

Interactive journey planning applications which provide routing information in real time have become a part of our everyday lives. While Dijkstra's Algorithm [17] solves the shortest path problem in quasi-linear time, it still takes several seconds on continental-sized networks, which is too slow for interactive use. Practical applications therefore rely on *speedup techniques*, which compute auxiliary data in a preprocessing phase and then use this data to speed up the query phase. Recent decades have seen the development of many successful speedup techniques for route planning on road networks [4]. These achieve query times of less than a millisecond with only moderate preprocessing time and space consumption.

© Ernestine Großmann, Jonas Sauer, Christian Schulz, and Patrick Steil;
licensed under Creative Commons License CC-BY 4.0

For public transit networks, the state of the art is not as satisfactory. In order to achieve query times below a millisecond on large country-sized networks, existing techniques must precompute data in the tens to hundreds of gigabytes. This discrepancy has been explained by the fact that road networks exhibit beneficial structural properties that are not as pronounced in public transit networks [2]. An additional challenge is that passengers in public transportation systems typically consider more criteria than just the travel time when evaluating journeys. Most recent algorithms in the literature Pareto-optimize at least two criteria: arrival time and the number of used trips. For these reasons, a speedup technique which achieves very low query times with only a moderate amount of precomputed data remains elusive.

State of the Art. For this work, we consider algorithms for journey planning in public transit networks which Pareto-optimize arrival time and number of trips. For a more general overview of journey planning algorithms, we refer to [4]. The classical approach is to model the public transit timetable as a graph and then apply a multicriteria variant of Dijkstra's Algorithm [19, 22, 18]. The time-dependent and time-expanded approaches are the two most prominent ways of modeling the timetable. In the *time-dependent* model [11, 23], stops in the network are represented by nodes in the graph and connections between them as edges with a time-dependent, piecewise linear travel time function. This yields a compact graph but requires a time-dependent version of Dijkstra's Algorithm. By contrast, the *time-expanded* model [22, 23] introduces a node for each event in the timetable (e.g., a vehicle arriving or departing from a stop). Edges connect consecutive events of the same trip and events between which a transfer is possible. The resulting graph is significantly larger but has scalar edge weights, allowing Dijkstra's Algorithm to be applied without modification.

Using a graph-based model has the advantage that speedup techniques for Dijkstra's Algorithm can be applied. However, the achieved speedups are much smaller than on road networks [2, 7]. A notable technique which has been applied to bicriteria optimization in public transit networks is Arc-Flags [21]. Its basic idea is to partition the graph into regions and to compute a *flag* for each combination of edge and region, which indicates whether the edge is required to reach the region. Dijkstra's Algorithm can then be sped up by ignoring unflagged edges. Arc-Flags has been applied to both time-dependent [10] and time-expanded [15] graphs, although only arrival time was optimized in the latter case. This yielded speedups of 3 and 4, respectively, whereas Arc-Flags on road networks achieves speedups of over 500 [21].

More recent algorithms do not model the timetable as a graph but employ more cache-efficient data structures to achieve faster query times. Notable examples are RAPTOR [16] and Trip-Based Public Transit Routing (TB) [25]. The latter employs a lightweight preprocessing phase which precomputes relevant transfers between individual trips. This yields query times in the tens of milliseconds, even on large networks, significantly improving upon graph-based techniques. HypRAPTOR [14] achieves a speedup of 2 over RAPTOR by using hypergraph partitioning to group the vehicle routes into cells and precomputing a set of *fill-in routes* which are required to cross cell boundaries. Applying the same approach to TB has only yielded a speedup of 20–40% [1].

Algorithms which reduce query times to the sub-millisecond range do so by precomputing auxiliary data whose size is quadratic in the size of the network. Public Transit Labeling (PTL) [13] adapts the ideas of Hub Labeling [12] to time-expanded graphs. While this yields query times of a few microseconds, it requires tens of gigabytes of space on metropolitan networks. Moreover, this does not include the additional overhead required for *journey unpacking*, i.e., retrieving descriptions of the optimal journeys, which would increase the size of the auxiliary data into the hundreds of gigabytes. Transfer Pat-

terns (TP) [3] employs a preprocessing phase which essentially answers every possible query in advance. Since storing a full description of every optimal journey would require too much space, TP condenses this information into a generalized search graph for each possible source stop, which is then explored during the query phase. On the network of Germany, TP answers queries in less than a millisecond but requires hundreds of hours of preprocessing time and over 100 GB of space. Scalable Transfer Patterns [5] reduces the preprocessing effort with a clustering-based approach, but the resulting query times are only barely competitive with TB. Trip-Based Routing Using Condensed Search Trees (TB-CST) [26] re-engineers the ideas of TP with a faster, TB-based preprocessing algorithm and by splitting the computed search graphs in order to save space.

Contribution. We revisit the concept of Arc-Flags for public transit journey planning. In contrast to previous approaches, we use modern TB-based algorithms in preprocessing and query phases. The high cache efficiency and stronger pruning rules of these algorithms drastically reduce the search space and running times. The resulting algorithm, Arc-Flag TB, matches or exceeds the performance of TB-CST with a similar precomputation time and significantly lower space consumption. Compared to TB, it achieves a speedup of one order of magnitude on metropolitan networks and two orders of magnitude on country networks. Since the number of regions in the underlying network partition is a configurable parameter, Arc-Flag TB additionally offers a tradeoff between query performance and the size of the precomputed data.

We identify an issue in the transfer precomputation of TB, which both TB-CST and Arc-Flag TB rely on. As a result, both algorithms answer some queries incorrectly. We discuss how this issue can be resolved in the future. In its current configuration, Arc-Flag TB answers 1–6% of queries incorrectly, depending on the network, compared to over 20% for TB-CST on some networks. Altogether, we show that Arc-Flags for public transit networks has more potential than previously thought.

2 Preliminaries

2.1 Basic Concepts

Public Transit Network. A *public transit network* is a 4-tuple $(\mathcal{S}, \mathcal{L}, \mathcal{T}, \mathcal{F})$ consisting of a set of stops $\mathcal{S}$, a set of lines $\mathcal{L}$, a set of trips $\mathcal{T}$, and a set of footpaths $\mathcal{F} \subseteq \mathcal{S} \times \mathcal{S}$. A *stop* $p \in \mathcal{S}$ is a location where a vehicle stops and passengers can enter or exit the vehicle. A *trip* is a sequence $T = \langle T[0], T[1], \dots \rangle$ of *stop events*, where each stop event $T[i]$ has an associated arrival time $\tau_{\text{arr}}(T[i])$, departure time $\tau_{\text{dep}}(T[i])$ and stop $p(T[i]) \in \mathcal{S}$. We denote the number of stop events in T as $|T|$. Trips with the same stop sequence that do not overtake each other are grouped into *lines*. A trip $T_a \in \mathcal{T}$ overtakes another trip $T_b \in \mathcal{T}$ if there are stops $p, q \in \mathcal{S}$ such that T_b arrives (or departs) later at p than T_a, but T_a arrives (or departs) earlier than T_b at q. The set of all trips belonging to a line L is denoted as $\mathcal{T}(L)$. Since trips $T_a, T_b \in \mathcal{T}(L)$ cannot overtake each other, we can define a total ordering

$$T_a \preceq T_b \iff \forall i \in [0, |T_a|) : \tau_{\text{arr}}(T_a[i]) \leq \tau_{\text{arr}}(T_b[i])$$
$$T_a \prec T_b \iff T_a \preceq T_b \land \exists i \in [0, |T_a|) : \tau_{\text{arr}}(T_a[i]) < \tau_{\text{arr}}(T_b[i]).$$

A *footpath* $(p, q) \in \mathcal{F}$ allows passengers to transfer between stops p and q with the *transfer time* $\Delta\tau_{\text{fp}}(p, q)$. If no footpath between p and q exists, we define $\Delta\tau_{\text{fp}}(p, q) = \infty$. If $p = q$, then $\Delta\tau_{\text{fp}}(p, q) = 0$. We require that the set of footpaths is transitively closed and fulfills the triangle inequality, i.e., if there are stops $p, q, r \in \mathcal{S}$ with $(p, q) \in \mathcal{F}$ and $(q, r) \in \mathcal{F}$, then there must be a footpath $(p, r) \in \mathcal{F}$ with $\Delta\tau_{\text{fp}}(p, r) \leq \Delta\tau_{\text{fp}}(p, q) + \Delta\tau_{\text{fp}}(q, r)$.

A *trip segment* $T[i,j]$ $(0 \leq i < j < |T|)$ is the subsequence of trip T between the two stop events $T[i]$ and $T[j]$. A *transfer* $T_a[i] \to T_b[j]$ represents a passenger transferring from T_a to T_b at the corresponding stop events. Note that this requires $\tau_{\text{arr}}(T_a[i]) + \Delta\tau_{\text{fp}}(p(T_a[i]), p(T_b[j])) \leq \tau_{\text{dep}}(T_b[j])$. A *journey* J from a source stop p_s to a target stop p_t is an sequence of trip segments such that every pair of consecutive trip segments is connected by a transfer. In addition, a journey contains an *initial* and *final footpath*, where the initial footpath connects p_s to the first stop event, and the final footpath connects the last stop event to p_t.

Problem Statement. A journey J from p_s to p_t is evaluated according to two criteria: its arrival time at p_t, and the number of trips used by J. We say that J *weakly dominates* another journey J' if J is not worse than J' in either of the two criteria. Moreover, J *strongly dominates* J' if J weakly dominates J' and J is strictly better in at least one criterion. Given source and target stops $p_s, p_t \in \mathcal{S}$ and an earliest departure time τ_{dep} at p_s, a journey J from p_s to p_t is *feasible* if its departure time at p_s is not earlier than τ_{dep}. A *Pareto set* $\mathcal{P}$ is a set of journeys such that $\mathcal{P}$ has minimal size and every feasible journey is weakly dominated by a journey in $\mathcal{P}$. Given source and target stops $p_s, p_t \in \mathcal{S}$ and a departure time τ_{dep}, the *fixed departure time problem* asks for a Pareto set with respect to the two criteria arrival time and number of trips. For the *profile problem*, we are given an interval $[\tau_1, \tau_2]$ of possible departure times in addition to p_s and p_t. Here, the objective is to find the union of the Pareto sets for each distinct departure time $\tau \in [\tau_1, \tau_2]$. In the *full-range profile problem*, the departure time interval spans the entire service duration of the network.

Graph. A directed, weighted graph $G = (V, E, c)$ is a triple consisting of a set of nodes V, a set of edges $E \subseteq V \times V$, and an edge weight function $c : E \to \mathbb{R}$. We denote by $n = |V|$ the number of nodes and $m = |E|$ the number of edges. A *path* $P = \langle v_1, v_2, \ldots, v_k \rangle$ is a sequence of nodes between v_1 and v_k such that an edge connects each pair of consecutive nodes. The weight of a path is the sum of the weights of all edges in the path. A path $P = \langle v_s, \ldots, v_t \rangle$ between a source node v_s and a target node v_t is called the *shortest path* if there is no other path between v_s and v_t with a smaller weight.

Given a value $k \in \mathbb{N}$ and a graph $G = (V, E, c)$, a (k-way) *partition* of G is a function $r : V \to \{1, \ldots, k\}$ which partitions the node set V into k *cells*. The set of nodes in cell i is denoted as $V_i := r^{-1}(i)$. An edge (u, v) is called a *cut edge* if its endpoints u and v belong to different cells. A node is called a *boundary node* if it is incident to a cut edge. The partition is called *balanced* for an imbalance parameter $\varepsilon > 0$ if the size of each cell V_i is bounded by $|V_i| \leq (1 + \varepsilon) \left\lceil \frac{|V|}{k} \right\rceil$. The *graph partitioning problem* asks for a balanced partition that minimizes the weighted sum of all cut edges.

2.2 Related Work

Trip-Based Public Transit Routing. The Trip-Based Public Transit Routing (TB) algorithm [25] solves the fixed departure time problem on a public transit network. It employs a precomputation phase, which first generates all possible transfers between stop events. Then, using a set of pruning rules, transfers that are not required to answer queries are discarded. We denote the remaining set of transfers as $\mathfrak{T}$. Note that $\mathfrak{T}$ may still contain transfers which do not occur in any Pareto-optimal journey.

The TB query algorithm is a modified breadth-first search on the set of trips and the precomputed transfers. The algorithm tracks which parts of the network have already been explored by maintaining a *reached index* $R(T)$ for each trip T. This is the index of

the first reached stop event of T, or $|T|$ if none have been reached yet. The TB query operates in *rounds*, where round i finds Pareto-optimal journeys which use i trips. Each round maintains a FIFO (first-in-first-out) queue of newly reached trip segments; these are then scanned during the round. A trip segment $T_a[i, j]$ is scanned by iterating over the stop events $T_a[k]$ with $i \leq k \leq j$ and relaxing all outgoing transfers $(T_a[k], T_b[\ell]) \in \mathfrak{T}$. If $\ell < R(T_b)$, then the trip segment $T_b[\ell, R(T_b) - 1]$ is added to the queue for the next round. Additionally, for every succeeding trip T'_b of the same line with $T_b \preceq T'_b$, the reached index $R(T'_b)$ is set to $\min(R(T'_b), \ell)$. This ensures that the search only enters the earliest reachable trip of each line, a principle we call *line pruning*.

Profile-TB is an extension of TB for solving the profile problem. It exploits the observation that journeys with a later departure time weakly dominate journeys with an earlier arrival time if they are equivalent or better in the other criteria. Therefore, it collects all possible departure times at p_s within the departure time interval $[\tau_1, \tau_2]$ and processes them in descending order. For each departure time, a run of the TB query algorithm is performed. All data structures, including reached indices, are not reset between runs. This allows results from earlier runs to prune suboptimal results in the current run, a principle called *self-pruning*. In order to obtain correct results, the definition of reached indices must be modified slightly. For each trip T and each possible number of trips i, the algorithm now maintains a reached index $R_i(T)$, which is the index of the first stop event in T which was reached with i or fewer trips. Whenever $R_i(T)$ is updated to $\min(R_i(T), k)$ for some value k, the same is done for the reached indices $R_j(T)$ with $j \geq i$.

Condensed Search Trees. Trip-Based Routing Using Condensed Search Trees (TB-CST) [26] employs Profile-TB to precompute search graphs which allow for extremely fast queries. The preprocessing phase solves the full-range profile problem for every possible pair of source and target stops by running a modified *one-to-all* version of Profile-TB from every stop. Consider the Profile-TB search for a source stop p_s. After each TB run, all newly found Pareto-optimal journeys are unpacked. This yields a breadth-first search tree with p_s as the root, trip segments as inner nodes, the reached target stops as leaves, and footpaths and transfers as edges. The search trees of all runs are merged into the *prefix tree* of p_s. Here, each trip segment $T[i, j]$ is replaced with a tuple (L, i) consisting of the line L with $T \in \mathcal{T}(L)$ and the stop index i where the line is entered.

To answer one-to-all queries, Profile-TB additionally maintains an earliest arrival time $\tau_{\text{arr}}(p, n)$ for each stop p and number of trips n. Like the reached indices, these arrival times are not reset between runs. When scanning a stop event $T[k]$ in round n, the algorithm iterates over all stops p with $\Delta\tau_{\text{fp}}(p(T[k]), p) < \infty$ and computes $\overline{\tau_{\text{arr}}} = \tau_{\text{arr}}(T[k]) + \Delta\tau_{\text{fp}}(p(T[k]), p)$. If $\overline{\tau_{\text{arr}}} < \tau_{\text{arr}}(p, n)$, then the best known journey to p with n trips was improved, so $\tau_{\text{arr}}(p, m)$ is set to $\min(\overline{\tau_{\text{arr}}}, \tau_{\text{arr}}(p, m))$ for all $m \geq n$.

To answer a query between source stop p_s and target stop p_t, TB-CST constructs a *query graph* from the prefix tree of p_s by extracting all paths which lead to a leaf representing p_t. Then a variant of Dijkstra's Algorithm is run on the query graph. Since the prefix tree only provides information about lines but not specific trips, these must be reconstructed during the query. When relaxing an edge from p_s to the first used line, the earliest reachable trip is identified based on the departure time at p_s. When relaxing an edge between lines L_1 and L_2, the used trip T_1 of L_1 is already known, so the algorithm explores the outgoing transfers of T_1 in $\mathfrak{T}$ to find the earliest reachable trip T_2 of L_2.

The space required to store all prefix trees can be reduced by extracting *postfix trees*. Consider the prefix tree for a source stop p_s. For each path from the root to a leaf representing a target stop p_t, a *cut node* is chosen. The subpath from the cut node to the leaf is then removed

from the prefix tree of p_s and added to the postfix tree of p_t. Since many of these extracted subpaths occur in multiple prefix trees, moving them into a shared postfix tree considerably reduces memory consumption. To construct the query graph for a source stop p_s and target stop p_t, the prefix tree p_s and the postfix tree of p_t are spliced back together at the cut nodes.

Arc-Flags. ARC-FLAGS is a speedup technique for DIJKSTRA'S ALGORITHM. Its basic idea is to precompute *flags* for each edge, which indicate whether the edge is necessary to reach a particular region of the graph. This allows DIJKSTRA'S ALGORITHM to reduce the search space during a query by ignoring edges which are not flagged for the target region.

Given a weighted graph $G = (V, E, c)$, the preprocessing phase of ARC-FLAGS performs two steps: First, it computes a partition $r : V \to \{1, \ldots, k\}$ of the node set into k cells, where k is a freely chosen parameter. Then, a *flags function* $b : E \times \{1, \ldots, k\} \to \{0, 1\}$ is computed. Each individual value $b(e, i)$ for an edge e and a cell i is called a *flag*, hence the name ARC-FLAGS. The flags function must have the following property: for each pair of source node v_s and target node v_t, there is at least one shortest path P from v_s to v_t such that $b(e, r(v_t)) = 1$ for every edge e in P. With this precomputed information, a shortest path query between v_s and v_t can be answered by running DIJKSTRA'S ALGORITHM but only relaxing edges e for which $b(e, r(v_t)) = 1$. The parameter k imposes a tradeoff between query speed and memory consumption. The space required to store the flags is in $\Theta(km)$, which is manageable for $k \ll n$. On the other hand, the search space of the query decreases for larger values of k since fewer flags will be set to 1 if the target cell is smaller.

Flags can be computed naively by solving the *all-pairs shortest path* problem, i.e., computing the shortest path between every pair of nodes. However, this requires $\Omega(n^2)$ precomputation time. The precomputation can be sped up by exploiting the observation that every shortest path that leads into a cell must pass through a boundary node. Thus, it is sufficient to compute backward shortest-path trees from all boundary nodes. For more details, see [20].

Arc-Flags for Public Transit Networks. Berger et al. [10] applied ARC-FLAGS to a time-dependent graph model in a problem setting which asks for *all* Pareto-optimal paths, including duplicates (i.e., Pareto-optimal paths which are equivalent in both criteria). They observed that for nearly every combination of edge e and cell i, there is at least one point in time during which e occurs on a Pareto-optimal path to a node in cell i. To solve this problem, the authors divided the service period of the network into two-hour intervals and computed a flag for each combination of edge, cell and time interval. However, this approach merely achieved a speedup of ≈ 3 over DIJKSTRA'S ALGORITHM.

Time resolution is not an issue in time-expanded graphs, where each node is associated with a specific point in time. However, Delling et al. [15] observed a different problem when applying ARC-FLAGS to a time-expanded graph, even when optimizing only arrival time. Since the arrival time of a path depends only on its target node, all valid paths are optimal. Delling et al. therefore evaluated various tiebreaking strategies to decide which optimal paths should be flagged. The most successful strategy only achieved a speedup of ≈ 4 over DIJKSTRA'S ALGORITHM. In the same paper, Delling et al. proposed a pruning technique called *Node-Blocking*, which applies the principle of line pruning to DIJKSTRA'S ALGORITHM in time-expanded graphs. The authors observed that Node-Blocking conflicts with their tiebreaking choices for ARC-FLAGS, leading to incorrectly answered queries. Therefore, they only evaluated ARC-FLAGS without Node-Blocking.

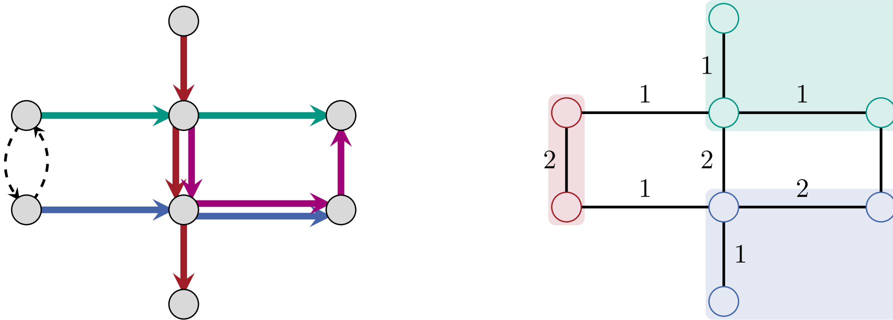


Figure 1 *Left:* An example network with stops as nodes, trips as colored solid edges and transfers as dashed edges. *Right:* The corresponding layout graph with edges weighted by the number of corresponding connections. Node groupings indicate a possible 3-way partition of the graph.

3 Arc-Flag TB

We now present the core ideas of our new algorithm Arc-Flag TB, which applies the main idea of Arc-Flags to TB. We first explain the general idea and then discuss details and optimizations. Finally, we compare our approach to similar algorithms.

The Arc-Flag TB precomputation performs two tasks: First it partitions the set $\mathcal{S}$ of stops into k cells, which yields a partition function $r : \mathcal{S} \to \{1, \ldots, k\}$. Then it computes a flag for each transfer $\mathfrak{t} \in \mathfrak{T}$ and cell i which indicates whether $\mathfrak{t}$ is required to reach any target stops in cell i. Formally, this yields a flags function $b : \mathfrak{T} \times \{1, \ldots, k\} \to \{0, 1\}$ with the following property: for each query with source stop p_s, target stop p_t and departure time τ_{dep}, there is a Pareto set $\mathcal{P}$ such that $b(\mathfrak{t}, r(p_t)) = 1$ for every transfer $\mathfrak{t} = T_a[i] \to T_b[j]$ that occurs in a journey $J \in \mathcal{P}$. A query between source stop p_s and target stop p_t is answered by running the TB query algorithm with one modification: a transfer $\mathfrak{t} \in \mathfrak{T}$ is only explored if the flag for the target cell is set to 1, i.e., $b(\mathfrak{t}, r(p_t)) = 1$.

3.1 Partitioning

To represent the topology of the public transit network without its time dependency, we define the *layout graph* G_L. The set of *connections* between a pair p, q of stops is given by

$$X(p, q) := \{T[i, i+1] \mid T \in \mathcal{T}, p(T[i]) = p, p(T[i+1]) = q\} \cup \{(p, q) \mid (p, q) \in \mathcal{F}\}.$$

Thus, a connection is either a trip segment between two consecutive stops or a footpath. Then the layout graph is defined as $G_L = (\mathcal{S}, E_L, c_L)$, with the set of edges $E_L \subseteq \mathcal{S} \times \mathcal{S}$ and edge weight function $c_L : E_L \to \mathbb{N}$ defined by

$$E_L := \{(p, q) \mid X(p, q) \neq \emptyset\},$$
$$c_L((p, q)) := |X(p, q)|.$$

An illustration of a layout graph is given in Figure 1. The stop partition r is obtained by generating the layout graph and running a graph partitioning algorithm of choice. Due to the weight function, the partitioning algorithm will attempt to avoid separating stops which have many connections between them.

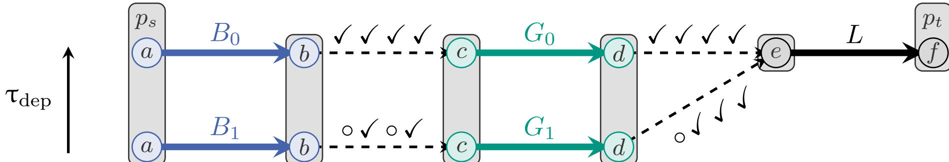


Figure 2 An example network illustrating the need for departure time fixing. Grey boxes represent stops. Nodes within the boxes represent stop events and are labeled with their indices along the respective trip. Within a stop, events are depicted in increasing order of time from bottom to top. Colored edges represent trips, with trips of the same line using the same color. Dashed edges with arrows represent transfers. Assume that p_t is the only stop in cell i of the stop partition r. For each transfer $\mathfrak{t}$, checkmarks indicate whether the flag of $\mathfrak{t}$ for cell i is set to 1. From left to right, these represent various configurations of the flag computation algorithm: unmodified Profile-TB, departure time buffering, flag augmentation, buffering + augmentation.

3.2 Flag Computation

To compute the flags, the full-range profile problem is solved for all pairs of source and target stops. As with TB-CST, this is done by running one-to-all Profile-TB search for every possible source stop. After each TB run of the Profile-TB search, all newly found journeys are unpacked. For a journey J to a target stop p_t and each transfer $\mathfrak{t}$ in J, the flag $b(\mathfrak{t}, r(p_t))$ is set to 1. Once all flags have been computed, transfers for which no flags are set to 1 can be removed from $\mathfrak{T}$.

Flag Compression. For an edge e, we call the set of flags $b(e, i)$ for $1 \leq i \leq k$ its *flag pattern*. Bauer et al. [6] observed for Arc-Flags on road networks that many edges in the graph share the same flag pattern. They exploit this with the following compression technique: All flag patterns which occur in the graph are stored in a global array A. For each edge e, the algorithm does not store the flag pattern of e directly, but rather the index i for which $A[i]$ holds the flag pattern of e. This significantly reduces memory consumption at the cost of an additional pointer access whenever an edge is relaxed. We also apply this compression technique in Arc-Flag TB and sort the flag pattern array in decreasing order of occurrence. This ensures that the most commonly accessed flag patterns are stored close together in memory, which increases the likelihood of cache hits.

3.3 Resolving Issues with Correctness

Departure Time Buffering. Due to line pruning, a TB query always enters the earliest reachable trip of a line; later trips of the same line are not explored. However, because Profile-TB processes departure times in decreasing order and applies self-pruning, it returns journeys which depart as late as possible. These two pruning rules conflict, leading to situations where Arc-Flag TB fails to find a Pareto-optimal journey. An example of this is shown in Figure 2. An unmodified Profile-TB search from p_s will find the journey $J_0 := \langle B_0[a, b], G_0[c, d], L[e, f]\rangle$ and flag it for cell i. However, it will not flag the journey $J_1 := \langle B_1[a, b], G_1[c, d], L[e, f]\rangle$, which has an earlier departure time and is therefore processed in a later run, but has the same arrival time and number of trips. An Arc-Flag TB query from p_s to p_t with departure time $\tau_{\text{dep}}(B_1[a])$ will enter B_1 but not relax the unflagged transfer $B_1[b] \rightarrow G_1[c]$. While $B_0[b] \rightarrow G_0[c]$ is flagged, the query will not enter B_0 due to line pruning and therefore not relax this transfer either.

To solve this issue, we introduce the notion of the itinerary. An *itinerary* is a generalized description of a journey which specifies the lines used and the stop indices where they are entered and exited, but not the trips used. Corresponding to a stop event $T[i]$ is the *line event* $L[i]$ where $T \in \mathcal{T}(L)$. The stop visited by $L[i]$ is denoted as $p(L[i])$. A trip segment $T[i,j]$ corresponds to the *line segment* $L[i,j]$. An itinerary is therefore a sequence of line segments. The itinerary describing a journey $J = \langle T_1[b_1,e_1],\dots,T_k[b_k,e_k]\rangle$ is given by $\mathcal{I}(J) = \langle L_1[b_1,e_1],\dots,L_k[b_k,e_k]\rangle$, where $T_i \in \mathcal{T}(L_i)$ for $1 \le i \le k$. For a line L, an index i and a departure time τ_{dep}, let $T_{\min}(L,i,\tau_{\mathrm{dep}})$ denote the earliest trip of L which departs at $p(L[i])$ no earlier than τ_{dep}. For an itinerary $\mathcal{I} = \langle L_1[b_1,e_1],\dots,L_k[b_k,e_k]\rangle$, the journey $J_{\min}(\mathcal{I},\tau_{\mathrm{dep}})$ is the journey with itinerary $\mathcal{I}$ which takes the earliest reachable trip of every line when starting with departure time τ_{dep}. Formally, $J_{\min}(\mathcal{I},\tau_{\mathrm{dep}}) = \langle T_1[b_1,e_1],\dots,T_k[b_k,e_k]\rangle$ with $T_i = T_{\min}(L_i,b_i,\tau^i_{\mathrm{dep}})$ and

$$\tau^i_{\mathrm{dep}} = \begin{cases} \tau_{\mathrm{dep}} + \Delta\tau_{\mathrm{fp}}(p_s, p(L_1[b_1])) & \text{if } i = 1,\\ \tau_{\mathrm{arr}}(T_{i-1}[e_{i-1}]) + \Delta\tau_{\mathrm{fp}}(p(L_{i-1}[e_{i-1}]), p(L_i[b_i])) & \text{otherwise.} \end{cases}$$

In Figure 2, J_0 and J_1 have the same itinerary $\mathcal{I}$. To ensure that the query from p_s to p_t with departure time $\tau_{\mathrm{dep}}(B_1[a])$ is answered correctly by Arc-Flag TB, $J_{\min}(\mathcal{I},\tau_{\mathrm{dep}}(B_1[a])) = J_1$ must be flagged as well. In general, consider the one-to-all Profile-TB search from a source stop p_s. For a TB run with departure time τ_{dep} and a target stop p_t, let $\mathcal{P}$ be the found Pareto set. We define the *buffered Pareto set*

$$\mathcal{P}_{\mathrm{buf}}(\mathcal{P},\tau_{\mathrm{dep}}) \coloneqq \{J_{\min}(\mathcal{I}(J),\tau_{\mathrm{dep}}) \mid J \in \mathcal{P}\}.$$

Since $\mathcal{P}$ is a Pareto set, we know that every journey $J \in \mathcal{P}$ has the same arrival time as $J_{\min}(J,\tau_{\mathrm{dep}})$. Hence, the last trip segment of $J_{\min}(\mathcal{I}(J),\tau_{\mathrm{dep}})$ and J is always identical. However, for the other trip segments, $J_{\min}(\mathcal{I}(J),\tau_{\mathrm{dep}})$ may use earlier trips than J. We modify the Profile-TB search to flag all transfers in $\mathcal{P}_{\mathrm{buf}}(\mathcal{P},\tau_{\mathrm{dep}})$. To do this efficiently, we employ an approach which we call *departure time buffering*. An itinerary $\mathcal{I}$ beginning with the line segment $L_1[b_1,e_1]$ is *unpacked* within the interval $(\tau_1,\tau_2]$ as follows: For a trip $T_1 \in \mathcal{T}(L_1)$, let $\tau_{\mathrm{dep}}(\mathcal{I},T_1) \coloneqq \tau_{\mathrm{dep}}(T_1[b_1]) - \Delta\tau_{\mathrm{fp}}(p_s,p(T_1[b_1]))$ be the departure time of a journey with the itinerary $\mathcal{I}$ that uses T_1 as the first trip. For each trip $T_1 \in \mathcal{T}(L_1)$ with $\tau_{\mathrm{dep}}(\mathcal{I},T_1) \in (\tau_1,\tau_2]$, the journey $J_{\min}(\mathcal{I},\tau_{\mathrm{dep}}(\mathcal{I},T_1))$ is constructed and its transfers are flagged.

For each stop p and round n, the algorithm maintains not only the earliest arrival time $\tau_{\mathrm{arr}}(p,n)$ but a buffered itinerary $\mathcal{I}(p,n)$, which represents the journey associated with $\tau_{\mathrm{arr}}(p,n)$, as well as the departure time $\tau_{\mathrm{dep}}(p,n)$ of the run in which $\tau_{\mathrm{arr}}(p,n)$ was last changed. If $\tau_{\mathrm{arr}}(p,n)$ is improved during a run with departure time τ_{dep}, then the journey corresponding to this arrival time is not flagged right away. Instead, after the end of the run, the algorithm unpacks the buffered itinerary $\mathcal{I}(p,n)$ within the interval $(\tau_{\mathrm{dep}},\tau_{\mathrm{dep}}(p,n))$ (unless $\mathcal{I}(p,n)$ has not been set before). Afterwards, the buffered itinerary $\mathcal{I}(p,n)$ is updated by unpacking the journey corresponding to the new value of $\tau_{\mathrm{arr}}(p,n)$. After the last TB run of the profile search, the remaining buffered itineraries are processed. For every stop p and round n such that $\tau_{\mathrm{arr}}(p,n) < \infty$, the itinerary $\mathcal{I}(p,n)$ is unpacked within the interval $(-\infty,\tau_{\mathrm{dep}}(p,n))$.

Flag Augmentation. Departure time buffering does not fix all issues caused by the incompatibility between line pruning and self-pruning. Consider the example shown in Figure 3. Once again, an unmodified Profile-TB search from p_s will find the journey $J_0 \coloneqq \langle B[a,b], G_0[c,d], L[e,f]\rangle$ and flag it for cell i, whereas the equivalent

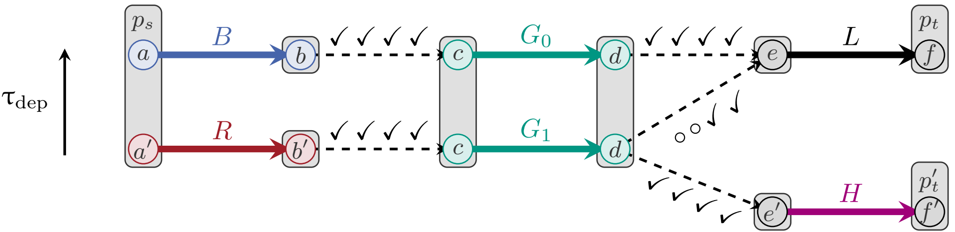


Figure 3 An example network illustrating the need for flag augmentation. Nodes, edges and checkmarks have the same meaning as in Figure 2. Assume that the stops p_t and p'_t are the only stops in cell i of the stop partition r.

journey $J_1 := \langle R[a', b'], G_1[c, d], L[e, f]\rangle$ is discarded. In this case, however, departure time buffering will not cause J_1 to be flagged either because it starts with a different line than J_0. Once again, consider an Arc-Flag TB query from p_s to p_t with departure time $\tau_{dep}(R[a'])$. If the transfer $R[b'] \to G_1[c]$ is not flagged, then the algorithm will enter G_0 and find J_0. However, in the example network, this transfer is flagged due to another journey $J'_1 := \langle R[a', b'], G_1[c, d], H[e', f']\rangle$, which leads to another target stop p'_t in cell i. As a consequence, G_1 is entered, but the unflagged transfer $G_1[d] \to L[e]$ is not relaxed, while G_0 is not entered due to line pruning.

To fix this issue, we define the *augmented flags function* $\hat{b} : \mathfrak{T} \times \{1, \ldots, k\} \to \{0, 1\}$. Consider a line L, a trip $T_a \in \mathcal{T}(L)$ and a transfer $\mathfrak{t} = T_a[i] \to T_b[j] \in \mathfrak{T}$. We define the set of *successor transfers* $\mathfrak{T}^{\uparrow}(\mathfrak{t})$ as

$$\mathfrak{T}^{\uparrow}(T_a[i] \to T_b[j]) := \{T'_a[i] \to T_b[j] \in \mathfrak{T} \mid T'_a \in \mathcal{T}(L), T_a \preceq T'_a\}.$$

Then $\hat{b}$ is defined as follows for a transfer $\mathfrak{t} \in \mathfrak{T}$ and cell i:

$$\hat{b}(\mathfrak{t}, i) := \bigvee_{\mathfrak{t}' \in \mathfrak{T}^{\uparrow}(\mathfrak{t})} b(\mathfrak{t}, i)$$

In the example from Figure 3, using $\hat{b}$ instead of b resolves the problem, provided that all transfers which occur in J_1 are included in the set $\mathfrak{T}$ of transfers generated by the TB preprocessing phase. Note that flag augmentation alone (without departure time buffering) will not fix the issue shown in Figure 2 because the transfer $B_1[b] \to G_1[c]$ will not be flagged. Thus, both fixes must be combined.

TB Transfer Precomputation. A final issue is due to the TB preprocessing phase, which computes the set $\mathfrak{T}$ of transfers. Unfortunately, its rules for pruning unnecessary transfers are too strong to guarantee that the transfers required by the Arc-Flag TB preprocessing are always generated. It is possible to construct examples akin to Figure 2 where the transfer $G_1[d] \to L[f]$ is not included in $\mathfrak{T}$, e.g., because a transfer to a different trip than L is preferred. In this case, both Arc-Flag TB and TB-CST[1] will fail to return correct results. Adapting the pruning rules of the TB preprocessing phase to resolve these issues remains an open problem.

To show that they can be resolved in principle, we implemented a prototypical variant of the Arc-Flag TB preprocessing that performs the profile searches with rRAPTOR [16], the profile variant of RAPTOR. While rRAPTOR does not rely on precomputed transfers,

[1] We reported this issue to the author of TB-CST, who concurred with our findings. The original TB-CST publication [26] mainly focused on evaluating profile queries, where this issue does not occur.

it suffers from the same conflict between line pruning and self-pruning as Profile-TB. This conflict was previously noticed and resolved in the context of ULTRA [8, 9], a preprocessing technique for multimodal journey planning which is based on rRAPTOR. We applied the modifications proposed for ULTRA to our rRAPTOR implementation. The resulting variant of Arc-Flag TB has significantly higher preprocessing times than the Profile-TB-based one but answered all queries correctly in our preliminary experiments.

3.4 Comparison

We conclude this section by comparing Arc-Flag TB to similar approaches.

TB-CST. TB-CST (without split trees) stores a generalized shortest path tree for every possible source stop. This offers a near-perfect reduction in the query search space but at the expense of quadratic memory consumption. The memory consumption of Arc-Flag TB is in $\Theta(|\mathfrak{T}|\, k)$, where k is the number of cells. Thus, Arc-Flag TB can be seen as a way to interpolate between TB and TB-CST regarding query search space and memory consumption. For $k = 1$, every non-superfluous transfer will be flagged, and thus the search space will be identical to that of TB with a minimal set of transfers. For $k = n$, the flags provide perfect information about whether a transfer is required to reach the target node.

An advantage of our approach is that the transfer flags provide information about which specific trips should be entered. In contrast, the TB-CST search graph only provides information about entire lines. This means that Arc-Flag TB does not have to invest additional effort during the query phase to find the earliest reachable trip of each line.

Time-Expanded Arc-Flags. Conceptually, our approach is similar to Arc-Flags on a time-expanded graph, albeit with TB as a query algorithm instead of Dijkstra's Algorithm. Delling et al. [15] observed low speedups when applying Arc-Flags to time-expanded graphs. We analyze some of the issues causing this and how Arc-Flag TB overcomes them. In a time-expanded graph, each visit of a vehicle at a stop is modeled with three nodes: an *arrival node*, a *transfer node* and a *departure node*. A journey corresponds to a path between two transfer nodes. However, boundary nodes in the partition may also be departure or arrival nodes. Consider, for example, a boundary node v of cell i which is an arrival node corresponding to the stop event $T[i]$. Arc-Flags will compute and flag a backward shortest-path tree rooted in v. A path in this tree corresponds to a “journey” which ends with the passenger remaining seated in T. However, there is no guarantee that this path can be extended to a Pareto-optimal journey which ends at a transfer node in cell i. Entering T may never be required to enter cell i. In this case, Arc-Flags produces superfluous flags. Arc-Flag TB avoids this problem by performing a one-to-all profile search from all stops, including those which are not boundary nodes in the layout graph. While this requires $\Omega(|\mathcal{S}|^2)$ preprocessing time, it considerably reduces the number of set flags.

Another feature of Arc-Flag TB that reduces the search space is that it flags transfers between stop events. In the time-expanded graph, a transfer corresponds to an entire path between an arrival and a departure node, which may pass through several transfer nodes. Consider two flagged transfers $T_1[i_1] \to T_2[i_2]$ and $T_3[i_3] \to T_4[i_4]$ whose paths in the time-expanded graph intersect. This has the effect of creating “virtual” transfers $T_1[i_1] \to T_4[i_4]$ and $T_3[i_3] \to T_2[i_2]$, which may not be flagged. Arc-Flags on the time-expanded graph will explore these transfers, whereas Arc-Flag TB will not. Note that Arc-Flag TB only flags transfers, not trip segments. This is because flagging trip segments would not provide any benefit beyond the first round of an Arc-Flag TB query: If a

Table 1 An overview of the networks on which we performed our experiments. Stops, lines, trips and footpaths are from the GTFS datasets. Transfers were generated by the TB precomputation.

Network	Stops	Lines	Trips	Footpaths	Transfers
Germany	441 465	207 801	1 559 118	1 172 464	60 919 877
Paris	41 757	9 558	215 526	445 912	23 284 123
Sweden	48 007	15 627	248 977	2 118	14 771 466
Switzerland	30 861	18 235	559 752	20 864	9 142 826

trip segment is not flagged for a specific cell, then neither are its incoming or outgoing transfers. Thus, an unflagged trip segment can only be entered during the first round, and no further trip segments are reachable from there.

Finally, Delling et al. note that all paths in a time-expanded graph have optimal arrival time and that a speedup is only achieved with suitable tiebreaking choices between equivalent paths. They observed that their tiebreaking choices conflicted with their implementation of line pruning, Node-Blocking. In Arc-Flag TB, the tiebreaking choices are dictated by the self-pruning of Profile-TB. While this also produces conflicts with line pruning, we resolved them by applying departure time buffering and flag augmentation. This allows Arc-Flag TB to fully benefit from both pruning rules, unlike previous approaches.

4 Experimental Evaluation

We evaluate the performance of Arc-Flag TB on a selection of real-world public transit networks. All experiments were run on a machine equipped with an `AMD EPYC 7702P` CPU with 64 cores, 128 threads, and 1 TB of RAM. Code for TB and Arc-Flag TB was written in `C++` and compiled using GCC with optimizations enabled (`-march=native -O3`). For TB-CST, we used the original code provided to us by the author [25]. The preprocessing phases of TB-CST and Arc-Flag TB, which run one-to-all Profile-TB from each stop, were parallelized with 128 threads.

Our datasets are taken from GTFS feeds of the public transit networks of Germany[2], Paris[3], Sweden[4] and Switzerland[5]. Details are listed in Table 1. For each network, we extracted the timetable of two consecutive weekdays to allow for overnight journeys.

Partitioning. We use the KaHIP[6] [24] open-source graph partitioning library to partition our networks. KaHIP is based on a multilevel approach, i.e., the input graph is coarsened, initially partitioned, and locally improved during uncoarsening. In our experiments, overall better results are obtained when coarsening is computed using clustering rather than edge matching as usual. More specifically, we use the memetic algorithm `kaffpaE` with the strong social configuration and an imbalance parameter of 5% in all of our experiments. As a time limit, we set 10 minutes for all networks regardless of the number k of desired cells. In our experiments, higher time limits did not significantly improve the results regarding the total number of flags set and average query times.

[2] https://gtfs.de/
[3] © https://navitia.io/
[4] https://trafiklab.se/
[5] https://opentransportdata.swiss/
[6] https://github.com/KaHIP/KaHIP

Table 2 Performance of Arc-Flag TB depending on the number of cells k. Departure time fixing and flag augmentation are enabled for all experiments. Query times and success rates are averaged over 10 000 random queries. Success rates are the percentage of queries for which Arc-Flag TB found a correct Pareto set, and the percentage of journeys in the correct Pareto sets for which Arc-Flag TB found an equivalent journey, respectively. Query times and memory consumption are measured with and without flag compression. Note that the preprocessing time does not include the partitioning, which was limited to 10 minutes in all configurations. Query times for $k = 1$ are for TB.

Network	k	Prepro. [hh:mm:ss]	Query time [μs]		Memory [MB]		Success rate [%]	
			Uncomp.	Comp.	Uncomp.	Comp.	Queries	Journeys
Germany	1	–	105 809	–	–	–	–	–
	1 024	34:12:44	739	1 068	5 120	725	93.53	95.12
	2 048	34:09:26	578	912	8 704	1 126	93.02	94.74
	4 096	34:10:59	548	745	16 384	2 474	92.22	94.24
Paris	1	–	4 502	–	–	–	–	–
	64	00:37:15	865	1 528	450	95	99.17	99.30
	128	00:37:19	671	1 486	513	133	99.08	99.25
	256	00:37:29	502	1 230	639	187	98.98	99.18
	512	00:37:35	393	982	891	282	98.76	99.03
	1 024	00:37:45	331	757	1 434	468	98.43	98.78
Sweden	1	–	7 583	–	–	–	–	–
	64	00:17:24	265	288	376	60	95.77	96.61
	128	00:17:26	167	202	428	69	95.36	96.24
	256	00:17:27	121	164	534	84	95.14	96.01
	512	00:17:30	97	140	744	116	94.86	95.82
	1 024	00:17:32	88	127	1 229	184	94.37	95.49
Switzerland	1	–	7 043	–	–	–	–	–
	64	00:13:07	223	225	222	35	96.74	97.53
	128	00:13:08	154	172	253	40	96.29	97.23
	256	00:13:09	112	136	315	50	95.75	96.86
	512	00:13:13	91	118	440	70	95.29	96.60
	1 024	00:13:14	81	108	698	114	94.80	96.29

Arc-Flag TB Performance. Performance measurements for Arc-Flag TB, including the impact of flag compression and the number of cells k, are shown in Table 2. For each configuration, we performed 10 000 queries with the source and target stops chosen uniformly at random and the departure time chosen uniformly at random within the first day of the timetable. As expected, the preprocessing time is mostly unaffected by k. Without flag compression and with the highest number of cells, Arc-Flag TB achieves a speedup of 193.1 on Germany, 13.6 on Paris, 86.2 on Sweden and 87.0 on Switzerland. Even without compression, the memory consumption for the computed flags is moderate at roughly 1 GB for the smaller networks and 16 GB for Germany. On all networks except Paris, flag compression is very effective: it reduces the memory consumption by a factor of 6–8 at the expense of 20–40% of additional query time. On Paris, the compression is less successful but still reduces the memory consumption by a factor of 3 while roughly doubling the query time. Figure 4 plots the speedup over TB and the memory consumption, with and without flag compression, depending on k. While the performance gains from doubling the

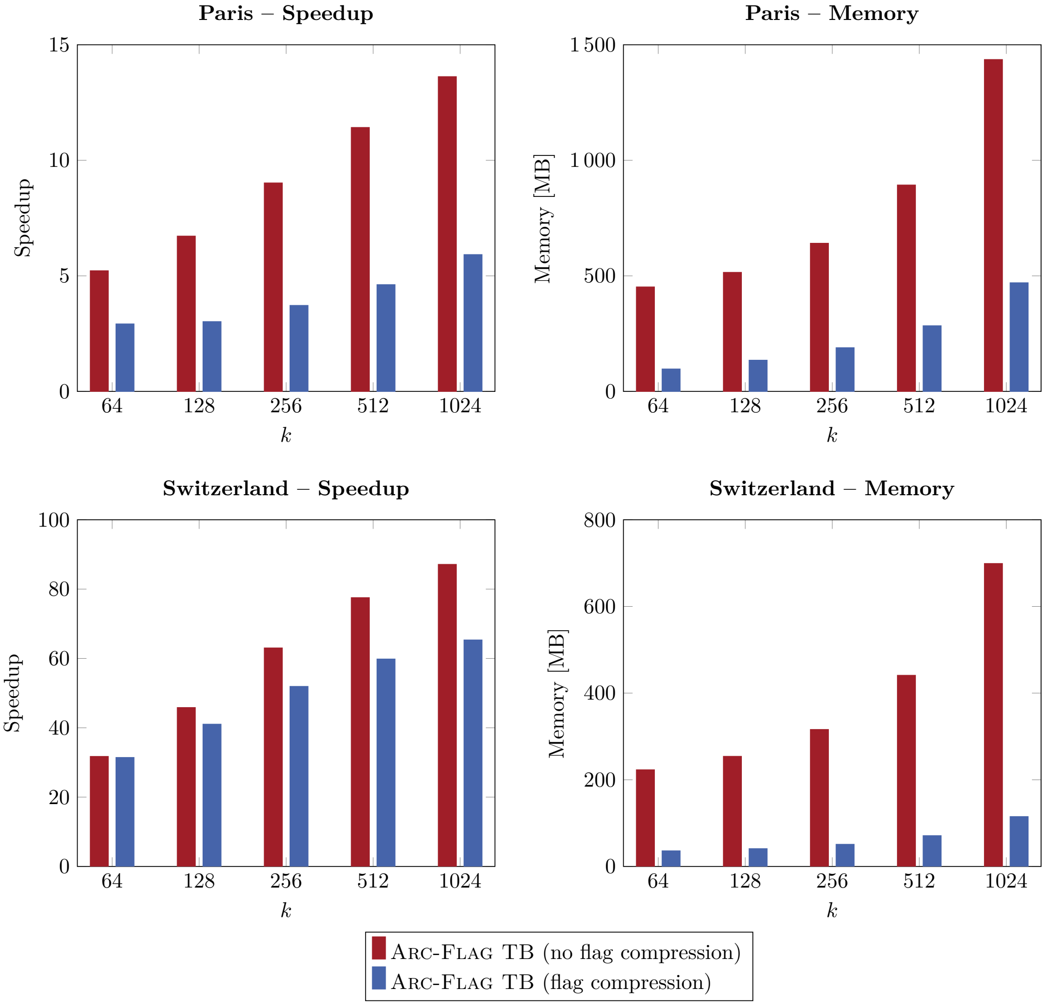


Figure 4 Average speedup over TB and memory consumption of Arc-Flag TB (with and without flag compression) on the Paris and Switzerland networks, depending on the number of cells k.

number of cells eventually decline, they still remain strong up to $k = 1\,024$. The rate of incorrectly answered queries is around 5% on the country networks and 1% on Paris, and only slightly increases with k. The lower speedup for Paris is explained by the fact that it is a dense metropolitan network with a less hierarchical structure and, therefore, harder to partition. Similar discrepancies in the performance between metropolitan networks and country networks were observed for Transfer Patterns [3] and TB-CST [26].

Result Quality. Table 3 shows the impact of departure time buffering and flag augmentation on the result quality of Arc-Flag TB. Departure time buffering significantly increases the preprocessing time, but this pays off in terms of the error rate, which is reduced from almost 30% to 6%. Flag augmentation on its own also reduces the number of incorrectly answered queries, but not as much. Combining both only slightly reduces the error rate compared to departure time buffering alone, which indicates that the scenario depicted in Figure 3 is rare. The results for our prototypical rRAPTOR-based preprocessing

Table 3 Impact of the preprocessing algorithm, departure time buffering (Buf.) and flag augmentation (Aug.) on the performance and success rate of ARC-FLAG TB, measured on the Switzerland network with $k = 1\,024$. Query times are measured without flag compression.

Algorithm	Buf.	Aug.	Prepro. [hh:mm:ss]	Query time [μs]	Success rate [%]	
					Queries	Journeys
Profile-TB	○	○	00:05:03	46	70.37	75.54
Profile-TB	○	●	00:05:05	56	81.87	85.39
Profile-TB	●	○	00:13:13	77	94.05	95.70
Profile-TB	●	●	00:13:14	81	94.80	96.29
rRAPTOR	–	–	01:26:51	58	100.00	100.00

algorithm are promising: While the preprocessing times are not practical, all queries are answered correctly. Furthermore, query times actually decrease compared to Profile-TB with buffering since flags are no longer set unnecessarily.

TB-CST. Finally, we compare ARC-FLAG TB against Witt's implementation of TB-CST with split trees [26] on our networks. The results are shown in Table 4. We do not report the performance of TB-CST with unsplit prefix trees since the precomputed data requires over 100 GB of memory even on the smaller networks. Therefore, a comparison would not be fair. Excluding the partitioning step, which always took 10 minutes in our experiments, the precomputation time of ARC-FLAG TB is 2–6 times higher, depending on the network. Although both techniques perform a one-to-all Profile-TB search from every stop, our algorithm additionally performs departure time buffering, which increases the precomputation time. The remaining difference, which amounts to a factor of 2 on the Switzerland network, is due to the fact that our implementation of Profile-TB is less optimized than Witt's. The memory consumption of ARC-FLAG TB is much lower than that of TB-CST, even with 1 024 cells and without flag compression. On the three smaller networks, our query times are similar or better. A proper comparison for the Germany network is difficult because we were not able to execute the provided code for TB-CST on this instance, and the query times reported in the original paper [26] are for a smaller version of the network. Nevertheless, we observe that ARC-FLAG TB with $k = 2\,048$ is at most four times slower than TB-CST while consuming less than a tenth of the space.

Overall, ARC-FLAG TB matches the query performance of TB-CST while requiring much less space. This is for two reasons: Firstly, the query time of TB-CST is dominated by the time required to construct the query graph. ARC-FLAG TB does not require this step. Secondly, the TB-CST query algorithm must reconstruct the earliest reachable trip of each used line at query time, whereas ARC-FLAG TB can rely directly on the precomputed transfers. Furthermore, we observe that TB-CST has a much higher error rate on Sweden and Switzerland than ARC-FLAG TB. We expect that this is because ARC-FLAG TB aggregates the flags by cell. Thus, even if the precomputation fails to find a required journey to a particular target stop, the transfers in that journey may still be flagged if they occur in journeys to other target stops in the same cell.

Table 4 Performance of TB-CST with split trees for 10 000 random queries. Note that we were not able to run TB-CST queries on the Germany network due to issues with the provided code. We instead list the query time reported in [26].

Network	Prepro. [hh:mm:ss]	Query time [μs]	Memory [MB]	Success rate [%]	
				Queries	Journeys
Germany	06:36:27	(156)	114 080	–	–
Paris	00:20:30	507	6 992	98.98	99.05
Sweden	00:07:42	91	3 400	75.99	91.67
Switzerland	00:02:22	66	1 586	80.72	89.88

5 Conclusion

We developed Arc-Flag TB, a speedup technique for public transit journey planning which combines Arc-Flags and Trip-Based Public Transit Routing (TB). We demonstrated that the stronger pruning rules of TB allow our approach to overcome previous obstacles in applying Arc-Flags to public transit networks. This allows Arc-Flag TB to achieve up to two orders of magnitude speedup over TB. Compared to TB-CST, a state-of-the-art speedup technique for TB, our algorithm achieves roughly the same query times with a similar precomputation time and only a fraction of the memory consumption. Unlike TB-CST, the query performance and memory consumption are configurable via the number of cells in the computed network partition. Currently, both algorithms answer some queries incorrectly due to an issue with the TB precomputation phase. However, we showed that the error rate of Arc-Flag TB is low and presented a prototypical variant of the algorithm which answers all queries correctly. In the future, it would be interesting to examine whether the performance of Arc-Flag TB can still be achieved with a subquadratic precomputation phase which only runs searches from the boundary nodes of the partition.

References

1. Prateek Agarwal and Tarun Rambha. Scalable Algorithms for Bicriterion Trip-Based Transit Routing. Technical report, Department of Civil Engineering, Indian Institute of Science, 2022. arXiv:2111.06654.
2. Hannah Bast. Car or Public Transport – Two Worlds. In *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science (LNCS)*, pages 355–367. Springer, 2009. doi:10.1007/978-3-642-03456-5_24.
3. Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast Routing in Very Large Public Transportation Networks using Transfer Patterns. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*, volume 6346 of *Lecture Notes in Computer Science (LNCS)*, pages 290–301. Springer, 2010. doi:10.1007/978-3-642-15775-2_25.
4. Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In *Algorithm Engineering: Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science (LNCS)*, pages 19–80. Springer, 2016. doi:10.1007/978-3-319-49487-6_2.
5. Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable Transfer Patterns. In *Proc. 18th Workshop on Algorithm Engineering and Experiments (ALENEX'16)*, pages 15–29. Society for Industrial and Applied Mathematics (SIAM), 2016. doi:10.1137/1.9781611974317.2.

6 Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *Journal of Experimental Algorithmics (JEA)*, 14:4.1–4.29, 2009. doi:10.1145/1498698.1537599.

7 Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study of Speed Up Techniques for Timetable Information Systems. *Networks*, 57:38–52, 2011. doi:10.1002/net.20382.

8 Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA'19)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.14.

9 Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution. Technical report, ITI Wagner, Department of Informatics, Karlsruhe Institute of Technology (KIT), 2023. arXiv:1906.04832.

10 Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller-Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'09)*, volume 12 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2009. doi:10.4230/OASIcs.ATMOS.2009.2148.

11 Gerth S. Brodal and Riko Jacob. Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. In *Proceedings of the 3rd Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'03)*, volume 92, pages 3–15. Elsevier, 2004. doi:10.1016/j.entcs.2003.12.019.

12 Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and Distance Queries via 2-Hop Labels. *SIAM Journal on Computing (SICOMP)*, 32(5):1338–1355, 2003. doi:10.1137/S0097539702403098.

13 Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. Public Transit Labeling. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, volume 9125 of *Lecture Notes in Computer Science (LNCS)*, pages 273–285. Springer, 2015. doi:10.1007/978-3-319-20086-6_21.

14 Daniel Delling, Julian Dibbelt, Thomas Pajor, and Tobias Zündorf. Faster Transit Routing by Hyper Partitioning. In *Proceedings of the 17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'17)*, volume 59 of *OpenAccess Series in Informatics (OASIcs)*, pages 8:1–8:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/OASIcs.ATMOS.2017.8.

15 Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering Time-Expanded Graphs for Faster Timetable Information. In *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*, volume 5868 of *Lecture Notes in Computer Science (LNCS)*, pages 182–206. Springer, 2009. doi:10.1007/978-3-642-05465-5_7.

16 Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. *Transportation Science*, 49:591–604, 2015. doi:10.1287/trsc.2014.0534.

17 Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959. doi:10.1007/BF01386390.

18 Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria Shortest Paths in Time-Dependent Train Networks. In *Proceedings of the 7th International Workshop on Experimental and Efficient Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science (LNCS)*, pages 347–361. Springer, 2008. doi:10.1007/978-3-540-68552-4_26.

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

20 Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 41–72. American Mathematical Society (AMS), 2009. doi:10.1090/dimacs/074/03.

21 Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. *Journal of Experimental Algorithmics (JEA)*, 11:2.8:1–2.8:29, 2006. doi:10.1007/11427186_18.

22 Matthias Müller-Hannemann and Mathias Schnee. Finding All Attractive Train Connections by Multi-Criteria Pareto Search. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science (LNCS)*, pages 246–263. Springer, 2007. doi:10.1007/978-3-540-74247-0_13.

23 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *Journal of Experimental Algorithmics (JEA)*, 12:2.4:1–2.4:39, 2008. doi:10.1145/1227161.1227166.

24 Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science (LNCS)*, pages 164–175. Springer, 2013. doi:10.1007/978-3-642-38527-8_16.

25 Sascha Witt. Trip-Based Public Transit Routing. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, volume 9294 of *Lecture Notes in Computer Science (LNCS)*, pages 1025–1036. Springer, 2015. doi:10.1007/978-3-662-48350-3_85.

26 Sascha Witt. Trip-Based Public Transit Routing Using Condensed Search Trees. In *Proc. 16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'16)*, volume 54 of *OpenAccess Series in Informatics (OASIcs)*, pages 10:1–10:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.ATMOS.2016.10.

Greedy Heuristics for Judicious Hypergraph Partitioning

Noah Wahl
Karlsruhe Institute of Technology, Germany

Lars Gottesbüren
Karlsruhe Institute of Technology, Germany

Abstract

We investigate the efficacy of greedy heuristics for the judicious hypergraph partitioning problem. In contrast to balanced partitioning problems, the goal of judicious hypergraph partitioning is to minimize the maximum load over all blocks of the partition. We devise strategies for initial partitioning and FM-style post-processing. In combination with a multilevel scheme, they beat the previous state-of-the-art solver – based on greedy set covers – in both running time (two to four orders of magnitude) and solution quality (18% to 45%). A major challenge that makes local greedy approaches difficult to use for this problem is the high frequency of *zero-gain moves*, for which we present and evaluate counteracting mechanisms.

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

Keywords and phrases hypergraph partitioning, local search algorithms, load balancing, local search

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.17

Supplementary Material
Software (Source Code): `https://github.com/kahypar/mt-kahypar/tree/judicious_refinement`
Other (Experimental Results and Phylo Instances): `https://github.com/noahares/PhyloBenchmarkSet`

1 Introduction

In this paper, we propose and study greedy heuristics for a variant of hypergraph partitioning named *judicious partitioning* [31], which has applications in load-balanced data distribution, for example in phylogenetic inference [2, 29]. Given a hypergraph $H = (V, E)$ and a number of blocks k, the goal is to partition the nodes V into k disjoint non-empty blocks $V_1, \dots, V_k$, such that the maximum *load* across blocks is minimized. The load $L(V_i)$ of a block V_i is defined as the weight-sum of hyperedges intersecting V_i, i.e., $L(V_i) = \sum_{e \in E, |e \cap V_i| > 0} \omega(e)$.

Contrary to the well-studied balanced partitioning problems with cut-based metrics [10, 23, 27, 14, 17, 15], the judicious variant does not impose a balance constraint on the blocks. Instead, balance is integrated as part of the objective, in the gap between the minimum and maximum load. Yet, just as the balanced variants the judicious partitioning problem is NP-hard [28], such that we focus on heuristics.

Phylogenetic Background

Phylogenetic inference takes a multiple sequence alignment (MSA) as input and tries to derive a *phylogenetic tree*, which is a strictly binary, unrooted tree that estimates the shared evolutionary history of the input. A potential tree topology is scored via a phylogenetic likelihood function (PLF) to estimate the likelihood of the tree, given the MSA. An MSA is a set of n strings from the DNA alphabet (A, T, C, G) and gap-characters such that all strings have the same length l and some distance function is minimized between pairs of strings.

© Noah Wahl and Lars Gottesbüren;
licensed under Creative Commons License CC-BY 4.0
21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis; Article No. 17; pp. 17:1–17:16
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Site: 1 2 3
C C C
G G G
A T T
T C C

Number of calculations
without site repeats: 9
with site repeats: $9 - 2 - 1 - 1 = 5$

Figure 1 Number of calculations for the PLF with and without site repeats.

It can be thought of as an $n \times l$ matrix where the strings form the rows. A single column of the MSA is called a *site*. To allow for different model parameters (e.g. different genes that evolve at different rates), the sites are split into p disjoint partitions. Because only the strings at the leaves are fixed, the likelihoods of all possible assignments of individual sites at inner nodes have to be calculated under the parameter model(s) to find the likelihood of the whole tree. This is computationally infeasible, so conditional likelihoods for each of the 4 possible characters at each individual site are calculated in a post-order traversal of the tree and combined at the root. However, this still incurs a lot of computation and accounts for 85-95% of total running time of phylogenetic inference tools. In Figure 1 the 4 MSA strings represent the leaves of the proposed tree on the left. For example, at the parent of CCC and GGG, the conditional likelihood for the first site asks for the likelihood of being assigned to A, C, T or G respectively, given that the children are fixed to C and G. For site 1 at the root of this example, the conditional likelihood given that the characters of the first site of the leaves are C, G, A and T, has to be calculated. This results in a total of $3 * 3 = 9$ conditional likelihood calculations for this tree. To parallelize the calculation of the PLF, sites can be split across cores because per-site likelihood calculations are independent of each other. Hence, we need to compute an assignment of sites to cores that minimizes load imbalance. So far, splitting a partition between cores incurs redundant calculations for the model parameters, but sites have equal costs.

The site repeats technique [24] is an optimization to eliminate redundant calculations. It identifies repeating patterns (repeat classes) in parts of distinct sites such that intermediate results can be reused among multiple sites, if they share the same partition and are assigned to the same core (otherwise the results would need to be communicated between cores which adds a scheduling component to the problem). This leads to varying costs for each site in a partition and makes it significantly more difficult to establish load balance between cores. Figure 1 shows an example of site repeats in a single partition that is assigned to a single core. Reusing the results for the pairs C-G (dark red), T-C (blue) and the quadruple C-G-T-C (orange) reduces the number of calculations to 5. Therefore, the goal is to assign sites to cores such that the maximum load is as small as possible, and to keep redundant calculations low, due to repeats split across different cores. Modeled as a hypergraph, each site is a hypernode and each repeats class is a hyperedge. As each hyperedge counts once towards the block-load, this corresponds to judicious partitioning.

Bottleneck Objectives

Objective functions where the value is obtained by taking the maximum across blocks are called *bottleneck* objectives; another example is maximum communication volume, where the maximum cut of edges from a block is minimized. These objectives are particularly challenging for greedy local search heuristics, because all node moves that do not involve the maximum load block do not change the objective function at all. Research on this problem

has so far been focused on extremal results [5], particularly for special classes such as bounded degree [7] or uniform hypergraphs [6, 19, 22]. We are only aware of one algorithm, that by Tan et al. [31], and one publicly available implementation thereof called HyperPhylo [2], which improves upon Tan et al.'s work via parallelization and several instance-specific optimizations. Roughly speaking, the idea is to enumerate increasing objective values and determine via a reduction to set cover whether a solution with this load exists. The resulting set cover is then transformed to a node-partition with this load. The set cover problem is solved greedily, which makes suboptimal solutions possible.

While there is only a small amount of literature on judicious partitioning, there is a vast amount on cut-based partitioning. We refer to recent surveys [8, 27, 9] for a broad overview. In this field, the most successful approaches are based on greedy heuristics, which motivates our study in this paper.

Contributions

Despite the difficulties faced by greedy heuristics, we demonstrate that when combined, our approaches significantly beat the existing state-of-the-art algorithm [2] both in terms of objective value (between 18% - 45%) and execution time (between two and four orders of magnitude). Our technical contributions are an iterative improvement algorithm inspired by the classical FM local search [13] (described in Section 3), as well as three greedy construction heuristics (Section 4). We show that randomized repetitions are a simple but effective technique to improve the solution quality and deal with the issue of many *zero-gain moves* to choose from during initial partitioning. Additionally, we considered a simple tie-breaking scheme which favors more balanced loads, but show that it does not lead to improved solutions. To address the issue of scalability of direct k-way initial partitioning for large k, we show that recursive partitioning is a viable option for many types of hypergraphs, but struggles with the class of regular hypergraphs that are encountered in data distribution problems for phylogenetic inference. Furthermore, we integrate our approaches in a state-of-the-art multilevel solver for balanced partitioning [16], leveraging its existing coarsening algorithms to obtain a multilevel solver for judicious partitioning.

Outline

For each component, we conduct thorough experiments on configuration and design choices, before comparing the full system with HyperPhylo in Section 5. In Section 2 we introduce preliminaries, including experimental setup. Each algorithmic component description is directly followed by evaluation and configuration experiments for said component, due to the large number of parameters. Only the best performing configuration moves on to the next section. We refrain from discussing the bio-informatics application in detail, and instead refer to the HyperPhylo paper [2] which describes the connection in detail. In the same vein, we do not conduct parallel phylogenetic inference simulations. Rather, we compare with HyperPhylo in terms of objective values on their benchmark set.

2 Preliminaries

A weighted hypergraph $H = (V, E, w)$ is defined as a set of nodes V and a set of hyperedges $E \subseteq 2^{|V|}$ with hyperedge weights $w : E \to \mathbb{R}_{>0}$. Let $n := |V|$ and $m := |E|$ denote the number of nodes and hyperedges. Functions on sets of nodes or hyperedges are extended to the sum over the set, e.g., $w(T) := \sum_{e \in T} w(e)$ for $T \subseteq E$. The nodes of a hyperedge are

called its pins. A node v is called incident to a hyperedge e if $v \in e$. $I(v)$ is the set of all incident hyperedges of v and $|I(v)|$ is the degree of v. By $p = \sum_{e \in E} |e| = \sum_{v \in V} |I(v)|$, we denote the number of pins. Furthermore, we use $[r]$ to denote $[r] \coloneqq \{1, \dots r\}$ for $r \in \mathbb{N}$.

Partitions

A k-way partition is a surjective function $\Pi : V \to [k]$. The blocks $V_i \coloneqq \Pi^{-1}(i)$ of Π are the inverse images. A 2-way partition is also called a bipartition.

The number of pins of a hyperedge e in block V_i is denoted by $\Phi(e, V_i) \coloneqq |V_i \cap e|$. For a block V_i, its *load* is defined as $L(V_i) \coloneqq w(\{e \in E \mid \Phi(e, V_i) > 0\})$. In the *judicious partitioning problem*, the goal is to minimize $max(L(V_1), \dots, L(V_k))$, the maximum load across blocks, which we also call the *judicious load*.

Node Moves: Penalties and Benefits

Our algorithms are based on node moves, i.e., reassigning a given node u from its current block $\Pi(u) = s$ to a different block $\Pi(u) \leftarrow t$. To calculate the difference in the objective function, we use two terms: the benefit and penalties. The *benefit* of removing a node from its current block is $b(u) = w(\{e \in I(u) \mid \Phi(e, \Pi(u)) = 1\})$, i.e., the weight of hyperedges e for which u is the last pin in the block. The *penalty* for adding node u to a given target block t is $p_t(u) = w(\{e \in I(u) \mid \Phi(e, t) = 0\})$, i.e., the weight of hyperedges e which did not intersect the block before, but will now. These values can be efficiently updated after a move, see [26, 18] for more details, where they are used for cut-based objectives. The running time for moving all nodes once with gain updates is $O(pk)$.

Performance Profile Plots

To compare the solution quality of different algorithms, we use *performance profiles* [12]. Let $\mathcal{A}$ be the set of algorithms we want to compare, $\mathcal{I}$ the set of instances, and $q_A(I)$ the maximum load of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. For each algorithm A, we plot the fraction of instances (y-axis) for which $q_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} q_{A'}(I)$, where τ is on the x-axis. Achieving higher fractions at lower τ-values is considered better. For $\tau = 1$, the y-value indicates the percentage of instances for which an algorithm performs best.

Machine Setup

All experiments are run on an AMD EPYC Rome 7702P with 2x64 cores clocked at 2.0-3.35 GHz with 1024 GB DDR4 RAM at 3200 MHz. Our proposed algorithms are single-threaded. The only parallelism used in our solver is during coarsening and for randomized repetitions during initial partitioning. Coarsening usually has negligible running time, and in the main experiments we use at most 5 repetitions. For HyperPhylo we used all 128 cores.

Benchmark Sets

We use two separate established benchmark sets for our experimental evaluations, which we refer to as set A and set P. The input instances are unweighted, however during multilevel coarsening hyperedges are aggregated and thus receive non-uniform weights.

Set A [21] consists of 488 real-world hypergraphs from different application domains for cut-based hypergraph partitioning, such as VLSI design [32, 1], SAT solving [4], and sparse matrices [11]. It is available from `https://algo2.iti.kit.edu/schlag/sea2017/` in hMetis format [23]. The hypergraphs in set A contain between 6K - 100M pins, 160 - 13M hyperedges and 7K - 13M nodes, with more detailed statistics available on the website.

Set P is derived from the data of Baar et.al. [2] used in their evaluation of HyperPhylo. It consists of a total of 11 hypergraphs; 7 smaller hypergraphs derived from sequence data from collaborative studies with biologists (prefixed with 59, 128 and 404 in our experiments) [30] and 4 larger hypergraphs from the one thousand insect transcriptome evolution project (the so called *supermatrix*, prefixed with sm in out experiments) [25]. An important property of the hypergraphs of set P is that they are regular, i.e. all nodes have the same degree, because each site has exactly one repeats class per inner node of the phylogenetic tree. These instances are available from their repository at `https://github.com/lukashuebner/HyperPhylo`. We converted these graphs to the hMetis format and made them available at `https://github.com/noahares/PhyloBenchmarkSet`.

We predominantly use set A for configuration experiments, due to its size and variety of hypergraphs. Set P is used to verify our results for these regular graphs, so we can later use our best configuration for the comparison with HyperPhylo. We include results for set P in our experimental sections alongside results on set A to show significant differences. The horse-race comparison with HyperPhylo in Section 5 is conducted only on set P, since HyperPhylo requires uniform node degrees (this is an implementation restriction to enable some optimizations).

3 Iterative Improvement

In this section, we introduce our first algorithmic contribution, namely an iterative improvement algorithm. To refine an initial partition we employ a local moving strategy similar to FM [13]. The full algorithm is shown in Algorithm 1. Contrary to non-bottleneck objectives, the only way to improve judicious load directly is to move nodes out of the block with the highest load. Let us denote this block by V_s. The order in which nodes are moved is prioritized by a gain function g defined in Equation 1, which represents the difference in load if a node $u \in V_s$ is moved to a different block V_t. In each step, we determine the highest gain node u and associated target block t, see line 4 in Algorithm 1. We then move u from s to t and update their loads.

$$g(u,t) = \begin{cases} b(u) & \text{if } L(V_t) + p_t(u) \leq L(V_s) - b(u) \\ L(V_s) - L(V_t) - p_t(u) & \text{else} \end{cases} \tag{1}$$

There are three possible scenarios for the maximum load after a move. If V_s remains the heaviest block, the load is decreased by $b(u)$. Otherwise, if V_t becomes the new heaviest block, then $b(u)$ has no influence because we only care about by how much $L(V_t) + p_t(u)$ differs from the prior maximum load $L(V_s)$. Furthermore, let V_i be the block with the highest load other than V_s. If $L(V_s) - b(u) < L(V_i)$, the actual gain is additionally capped at $L(V_i) - L(V_s) + b(u)$, because V_i becomes the new highest load block. To encourage some further optimization on V_s before moving on to V_i, we ignore this cap when calculating the next move. However, if the target block V_t becomes the new heaviest block through the move, the gain is negative, and we finish the optimization on V_s, see line 5.

In line 9, we update the gain values $b(v), p_s(v), p_t(v)$ for neighbors v with $\{u, v\} \subset e$ such that $\Phi(e, s) \leq 1$ or $\Phi(e, t) = 1$ (after the move). These are (almost) the same updates as for the cut-based objectives, see for example [16, 27] for details (we can omit an update for $\Phi(e, t) = 2$). In the worst case, the total cost of updates across the entire run is $O(kp)$, but in practice we often observe behavior closer to $O(p)$ since the average number of blocks intersecting a hyperedge is close to constant. To determine the highest gain move, we use one priority queue per target block, which we now update to reflect the new gain values of neighbors after the move.

Algorithm 1 Judicious FM.

1. **while** $\max_{i\in[k]}(L(V_i)) > \min_{i\in[k]}(L(V_i)) \cdot \delta$ **do**
2. $\quad s \leftarrow \arg\max_{i\in[k]}(L(V_i))$
3. $\quad$ **while** $\max_{i\in[k]-s}(L(V_i)) < L(V_s) \cdot \gamma$ **do**
4. $\quad\quad (u,t) \leftarrow \arg\max_{v\in V_s, i\in[k]-s}(g(v,i))$
5. $\quad\quad$ **if** $g(u,t) < 0$ **then break**
6. $\quad\quad \Pi(u) \leftarrow t$
7. $\quad\quad L(V_s) \leftarrow L(V_s) - b(u)$
8. $\quad\quad L(V_t) \leftarrow L(V_t) + p_t(u)$
9. $\quad\quad$ run gain updates for neighbors of u
10. $\quad$ **if** $s = \arg\max_{i\in[k]}(L(V_i))$ **then break**

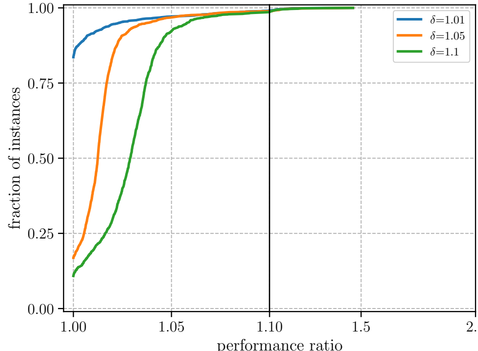


Figure 2 Performance profiles for varying δ in judicious FM on set A.

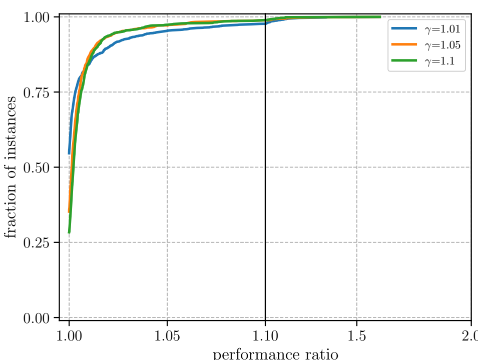


Figure 3 Performance profiles for varying γ in judicious FM on set A.

We continue the inner loop at line 3 until V_s is no longer the block with the highest load, subject to some relative margin $\gamma > 1$. This encourages further optimization on V_s and avoids frequently alternating between the two highest loaded blocks. However, once V_s is no longer the highest load block, we can worsen the solution (due to the ignored additional cap). This happens for example when all blocks have similar loads. As mentioned, we thus also break out of the loop once only negative gain moves remain.

We continue the outer loop (line 1) as long as there is a block with smaller load that can take in new nodes, again subject to a relative margin parameter $\delta > 1$. Notice that judicious FM cannot achieve loads smaller than the initial minimum load. We experimented with ideas to break out of local minima, but were unsuccessful. Thus, an important consideration for initial partitioning heuristics is to produce solutions with a considerable gap between minimum and maximum load.

Furthermore, notice that δ offers a trade-off between solution quality and running time, where smaller values give better loads, but need to perform more optimization. We note however, that, since we quickly steer into a local minimum, the running time of FM is usually negligible compared to initial partitioning.

3.1 Experiments

In the following, we evaluate the impact of the two parameters γ and δ, as well as how much judicious FM improves over the initial partition. For δ we expect small values to give the best solutions, whereas for γ the picture is not as clear. We want the other heaviest block to

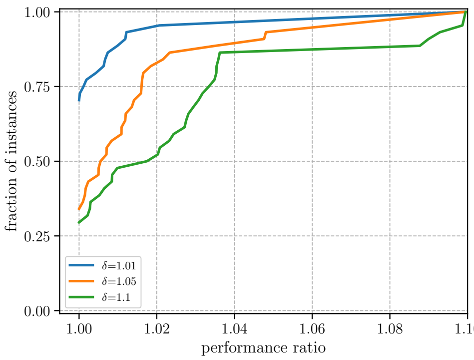


Figure 4 Performance profiles for varying δ in judicious FM on set P.

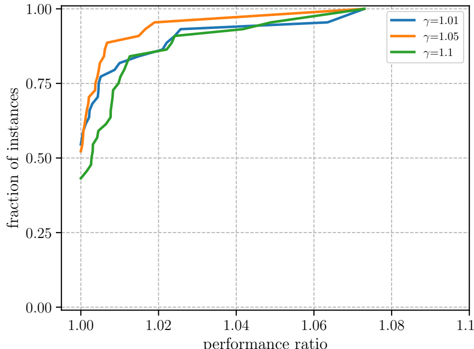


Figure 5 Performance profiles for varying γ in judicious FM on set P.

Table 1 Mean and percentile of the load achieved by judicious FM divided by the load of the initial solutions.

mean	gmean	p_{10}	p_{25}	p_{50}	p_{75}	p_{90}	max
1.43323	1.34457	1.05399	1.10473	1.21581	1.45799	1.89561	11.45786

be significantly heavier than the current one, to enable more improvement before we switch blocks again. However, if the gap between minimum and maximum load is small, there is not much room for improvement, and we are quickly left with only negative gains.

For both parameters, we tried the values $1.01, 1.05$ and 1.1. We ran a quadratic grid-search, but for readability the following figures display one parameter varied at a time, with the other fixed to the best value. Figure 2 (set A) and Figure 4 (set P) demonstrate that smaller values indeed perform better for δ. As the refinement scheme converges very quickly, its running time is often negligible compared to initial partitioning, so we use $\delta = 1.01$ in the following. In Figure 5, we see that $\gamma = 1.05$ performs best for set P, whereas Figure 3 shows that there is no significant difference for set A. We observed that the inner loop is more frequently terminated by negative gains than the stopping condition, which explains this behavior.

Finally, in Table 1 we show average and percentile improvements over the initial solution, that is the ratio between the load achieved by judicious FM divided by that of the initial solution. We see a geometric mean improvement of 34.4% and a median of 21.5% from using FM. These values are similar to what we can expect in cut-based partitioning [3]. The initial solutions for these experiments were obtained with the best performing configuration for initial partitioning from Section 4, which we discuss next.

4 Initial Partitioning

In this section, we present our greedy initial partitioning algorithms and evaluate their performance. We start with an approach where we construct all k blocks at the same time, and subsequently leverage the presented strategies as a subroutine for recursive partitioning.

4.1 Direct k-way Initial Partitioning

In initial partitioning, we start with all nodes unassigned and do not move already assigned nodes (until the refinement stage later on).

Similar to FM, the order in which nodes are assigned is determined greedily via a loss function $\varrho(v, i)$. At each step, we determine the node and target block with the lowest loss, and then update the unassigned nodes' losses. The difference is that all nodes must be moved (assigned) at the end. In the following we propose three strategies to define ϱ. Since nodes are initially unassigned, the loss functions are not based on benefit values $b(v)$ as in the move gain, just penalty terms $p_t(v)$ are incorporated. The time complexity of all three variants is $O(\log(n)(n + p)k)$. This bound stems from the updates to penalty values after an assignment [26, 18], and the priority queue updates.

Penalty

In the penalty strategy, we use $\varrho(v, t) = p_t(v)$, i.e., the next node to be assigned has the lowest $p_t(v)$ value globally. In this strategy, nodes are attracted to blocks that already contain many of its neighbors. Thus, already highly loaded blocks are preferred. As we do not impose a constraint on the block size or load, there is no mechanism to achieve a balanced distribution of nodes, as we would need for the traditional balanced partitioning problem. Despite this, it is an interesting strategy to consider, as it nudges the solution into very different local minima than the following strategies, which focus more on balanced blocks. While not competitive on the initial solutions, on some instances we observed better solutions after FM refinement, because the gap between the minimum and maximum load is wider, and thus the refinement has more room for improvement.

Block Load

In the block load strategy, we first select the currently lightest block t, and only then choose the node v with minimal $p_t(v)$. The advantage over the penalty strategy is that it keeps block loads evenly distributed. This comes at the cost of not considering good moves to blocks that are not the lightest.

Judicious Increase

In the judicious increase strategy, the loss of assigning a node v to block t is $\varrho(v, t) = \max(L(V_t) + p_t(v) - \max_{i\in[k]}(L(V_i)),\ 0)$. It keeps balance in mind by trying to increase the loads of all blocks evenly. Since this loss definition directly corresponds to the loss in maximum load, we expected the judicious increase strategy to perform the best, which is confirmed in the experiments. Yet, we decided to also evaluate the other two strategies as the judicious increase strategy is particularly prone to long streaks of consecutive equal losses, which makes it hard to distinguish between the different possible assignments to perform next.

Furthermore, we investigate a simple tie-breaking scheme for the judicious increase strategy. Allowing losses to become negative by omitting the outer *max* introduces tie-breaking of zero-loss moves by $L(V_t) + p_t(v)$. Intuitively speaking, this optimizes for more balanced loads, if we are not adding to the heaviest block.

4.2 Randomization

We use priority queues for selecting the target block first and then the node to move to that target block. Because all three strategies result in many equal loss-scores, the order in which moves are chosen heavily depends on the order of insertions into the priority queues. To break this dependency we introduce randomization, combined with multiple differently seeded repetitions, to cover a variety of possible assignment sequences. We implement this by attaching a randomly generated tag to each move added to a priority queue, which is used as the secondary comparison criterion. The same approach can be used for choosing different blocks in case of ties, to prevent assigning all nodes to one single block. Randomization and repetitions result in significantly increased odds of finding a good initial partition.

4.3 Coarsening

One of the most important components in cut-based partitioning is the multilevel scheme [20]. Initial partitioning and iterative improvement are not run directly on the input hypergraph. The hypergraph is iteratively coarsened by repeatedly contracting node clusters, which approximately preserve the structure, until the hypergraph is fairly small. Each iteration and associated hypergraph constitutes a level in the hierarchy. On the lowest level (the smallest hypergraph) an initial partition is computed, which is then projected back through the hierarchy, by assigning clustered nodes to the same block. Furthermore, iterative improvement is run on each level.

While designed for cut-based objectives, this scheme is directly applicable to the judicious partitioning problem, with a small modification. Hyperedges of size 1 cannot be removed, since they still contribute to the volume of their pin's block. To speed up the algorithmic components, we still remove such hyperedges and track the removed volume at each node.

We stop coarsening (to transition into initial partitioning) once the current hypergraph has less than $C \cdot k$ nodes, for a constant parameter C. This ensures, that we can place C nodes in each block on average, and thus the optimization algorithms have some leeway. In the following experiments we use $C = 50$, which was determined in preliminary experiments that are omitted here. The multilevel scheme has two major advantages over flat partitioning: iterative improvement can optimize the solution at different levels of granularity, and it is substantially faster since initial partitioning runs on small inputs.

4.4 Experiments on Direct k-way Initial Partitioning

For the evaluation of our direct k-way initial partitioning strategies we expected the judicious increase strategy to clearly outperform the other two. Figure 6 indeed shows a wide gap between the strategies. However, we were also interested in the trade-off between load-balanced initial partitions versus creating more room for improvement in the refinement phase. Figure 7 initially shows potential for the penalty strategy combined with FM post-processing, as it achieves more of the best solutions than the block-load strategy. However, the curve remains flat, being overtaken by block-load at just the 1.01 mark, indicating that on the solutions where the penalty strategy performs worse, it is significantly worse. On set P, see Figures 8, 9, the results are even more clearly in favor of judicious increase. We conclude that the judicious increase strategy produces the best k-way initial partitioning results, and thus use it as the default algorithm from now. Note that there are only small differences in running time between the strategies, which is why we omit plots for these here.

Next, we look at the effect of randomization on solution quality. This is shown in Figure 10. We clearly see an improvement when using 5 repetitions, but diminishing returns for 10 repetitions. Hence, we use 5 repetitions as the default value from now.

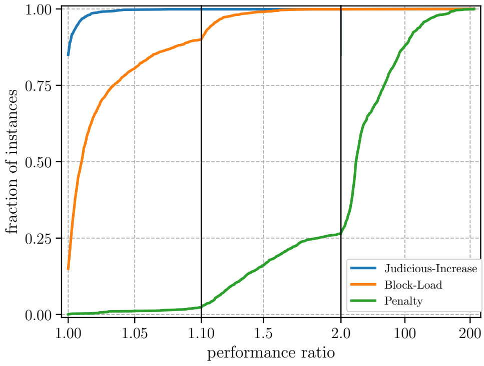


Figure 6 Performance profiles for initial partitioning strategies on set A.

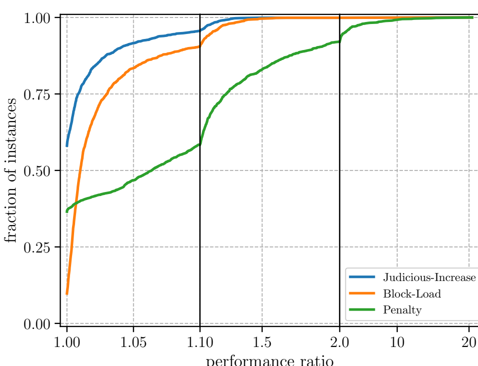


Figure 7 Performance profiles for initial partitioning strategies with FM post-processing on set A.

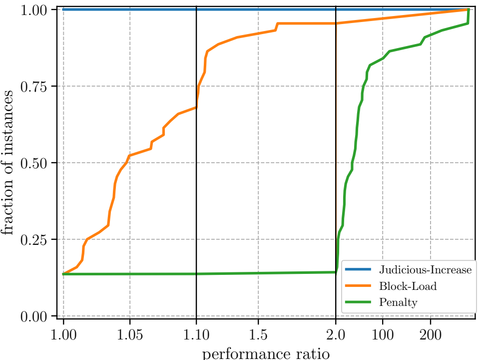


Figure 8 Performance profiles for initial partitioning strategies on set P.

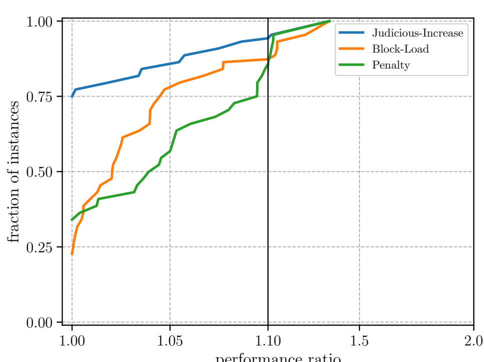


Figure 9 Performance profiles for initial partitioning strategies with FM post-processing on set P.

Lastly, Figure 11 shows that tie-breaking has no effect, neither positive nor negative. We assume that this stems from the fact that more balanced solutions early on inhibit the optimization potential of FM post-processing later on.

Still, we argue that there is a need for tie-breaking schemes. In Table 2 we report in how many steps of the initial partitioning procedure the best loss value is zero for the supermatrix instances of set P. There is a clear correlation between the number of zero loss moves and k. This is expected, as there are more blocks that offer free assignments, before becoming the new heaviest block. While for $k = 48$ only 10-20% of assignments have zero loss, for $k = 2048$ it is 97%-99%. These results indicate that there is certainly a need to distinguish zero-loss moves, but further investigation is needed to find better tie-breaking mechanisms.

4.5 Recursive Partitioning

The difficulty of many ties in the scores when constructing k-way partitions directly, was already observed for cut-based objectives. There, the solution is to restrict the assignment options by recursively bipartitioning on the coarsest graph, and transition to k-way local search only in the uncoarsening phase. This yields better solution quality than direct k-way [27] for initial partitions but we will show this is not the case for judicious partitioning. One reason to still consider recursive partitioning is the improved time complexity of $O(p \log k)$ compared to direct k-way's complexity of $O(pk)$.

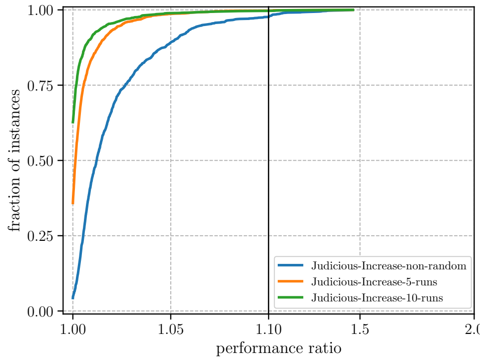


Figure 10 Effect of randomized repetitions on set A.

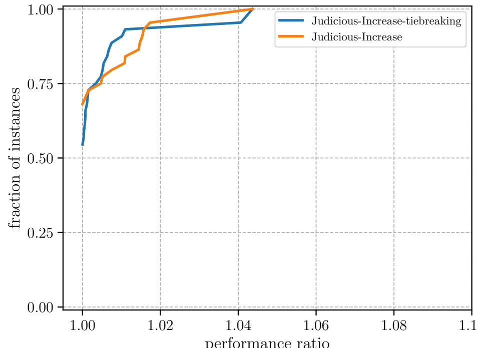


Figure 11 Effect of tie-breaking on set P.

Table 2 Fraction of zero-loss assignments on supermatrix instances of set P for direct k-way initial partitioning with the judicious increase strategy.

k	graph	$\frac{\lvert\varrho(v,t)=0\rvert}{\#moves}$	k	graph	$\frac{\lvert\varrho(v,t)=0\rvert}{\#moves}$
48	sm_part1_170859	0.113	256	sm_part1_170859	0.762
	sm_part3_31854	0.151		sm_part3_31854	0.802
	sm_part12_20753	0.152		sm_part12_20753	0.825
	sm_part24_11756	0.212		sm_part24_11756	0.866
160	sm_part1_170859	0.574	2048	sm_part1_170859	0.991
	sm_part3_31854	0.602		sm_part3_31854	0.984
	sm_part12_20753	0.636		sm_part12_20753	0.979
	sm_part24_11756	0.725		sm_part24_11756	0.976

In recursive bipartitioning, we first split into two blocks, then extract the sub-hypergraphs induced by the two blocks, and recursively partition these, leading to a binary recursion tree of depth $\lceil \log_2(k) \rceil$ to obtain a k-way partition. There are several issues with recursive partitioning, such as not optimizing the objective function directly, and the fact that splitting a node pair is irreversible. On the judicious metric we note one additional problem, namely if k is not a power of 2. Let $k_1 = \lfloor k/2 \rfloor, k_2 = \lceil k/2 \rceil$. We would need to target an imbalance of (k_1/k) to (k_2/k) in the loads of the two blocks, in order to (let us say hand-wavingly) pass load evenly down the recursion. Unfortunately, our algorithms are not designed for this. Instead, we optimize for balanced load on bipartitions (using judicious FM at each level), and assign the smaller number of final blocks k_1 to the recursive call on the block with smaller load. As additional base cases to the recursion, we perform direct 3-way or 5-way partitioning, which are the two cases with the highest required imbalance.

4.6 Experiments on Recursive Partitioning

In Figure 13, we show that direct k-way partitioning (DK) gives better solution quality than recursive bipartitioning (RB) for initial partitioning on set P. On set A the situation is less clear, see Figure 12, where recursive partitioning achieves more of the best solutions, but direct k-way converges faster towards 1. It is unclear why we see such a large discrepancy in initial judicious loads between the benchmark sets. Looking at geometric mean running times on set A, we have $0.039s$ for direct k-way versus $0.29s$ for recursive partitioning with

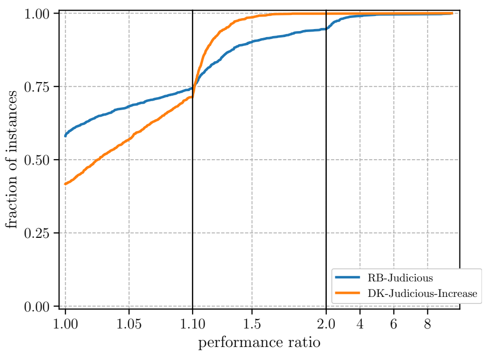


Figure 12 Performance profiles for initial partitioning judicious loads on set A.

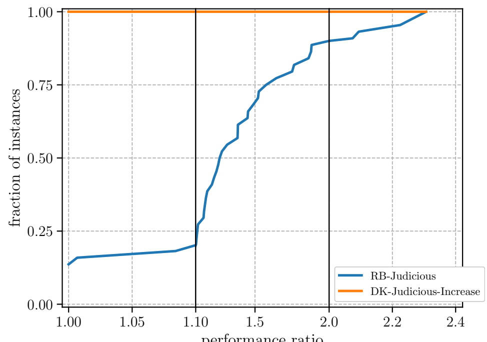


Figure 13 Performance profiles for initial partitioning judicious loads on set P.

$k = 48$, $0.354s$ versus $0.622s$ with $k = 160$, and $0.96s$ versus $0.81s$ with $k = 256$. Recursive partitioning becomes faster at $k = 256$, and for larger k there will be a point where flat direct k-way is no longer feasible. On set P, the geometric mean time for all k is 0.1s for recursive partitioning and 0.05s for direct k-way, and respectively 0.013s versus 0.017s for $k = 2048$. We conclude that direct k-way partitioning is the method of choice over recursive partitioning for initial judicious partitioning, as long as k is moderate.

5 Comparison to HyperPhylo

In this section, we evaluate the performance of our solver against that of HyperPhylo [2], the existing state of the art for judicious partitioning. Our solver uses the multilevel scheme with coarsening down to $50 \cdot k$ nodes, flat k-way partitioning with the judicious increase strategy on the coarsest hypergraph (randomization enabled, tie-breaking disabled), and refinement with judicious FM ($\delta = 1.01, \gamma = 1.05$) on each level.

Since the HyperPhylo implementation[1] is restricted to instances with uniform node degree (for optimization purposes), the comparison is restricted to set P. Table 3 shows loads and running times for our solver, as well as loads and times relative to ours for HyperPhylo (ours divided by HyperPhylo). Values below one thus correspond to cases where our solver performed better. These are highlighted in bold. Cases with $n \leq k$ are omitted because the optimal partition is to place each node in its own block, which is trivial.

The table is split into two parts: smaller instances at the top, and larger instances (supermatrix) at the bottom, which are deemed most important. We see that our solver outperforms HyperPhylo by a substantial margin in terms of running time and load in *all* reported cases on the supermatrix instances. The running time improvements range from two to four orders of magnitude, whereas the load improvements are between 18% to 45%.

On the seven smaller instances, our running time advantage is less pronounced. HyperPhylo even beats our solver on 6 out of 21 runs, particularly for the largest value of $k = 2048$. This is because larger values of k incur smaller maximum loads, which is an advantage of HyperPhylo, since it enumerates increasing objective values. On the other hand, our initial partitioning algorithms become slower, the larger k is, since there are more assignments to evaluate in each step. The largest slowdown is a factor of 59 (50ms vs 3s) for the instance 404-1 with $n = 2161$ on $k = 2048$, where both solvers achieve maximum load 402. This value

[1] `https://github.com/lukashuebner/HyperPhylo`

is the degree lower bound (each node must be assigned to one block), such that HyperPhylo finishes in the first iteration, whereas our solver has to go through all optimization steps. In terms of solution quality, our solver remains superior: it is beaten on only 4 out of 21 runs.

Table 3 Running times and loads of our solver versus HyperPhylo on benchmark set P. Values where our solver performs better are highlighted in bold.

graph	n	m	k	our load	$\frac{\text{our load}}{\text{HyperPhylo}}$	our time [s]	$\frac{\text{our time}}{\text{HyperPhylo}}$
59-s	160	671	48	77	1.35088	0.00450	**0.15511**
128-s	204	1170	48	167	1.06369	0.00648	**0.34105**
			160	126	1.00000	0.01151	1.27943
404-s	588	2525	48	511	**0.99031**	0.05110	**0.68127**
			160	438	1.08955	0.10469	7.47786
			256	402	1.00000	0.12523	8.34880
128-0	857	10853	48	550	**0.59524**	0.02196	**0.01923**
			160	289	**0.54735**	0.04136	**0.04452**
			256	243	**0.71261**	0.05950	**0.12501**
404-l	2161	40648	48	1718	**0.67162**	0.14527	**0.02432**
			160	879	**0.75451**	0.32195	**0.07700**
			256	771	**0.84262**	0.46104	**0.13795**
			2048	402	1.00000	3.09564	59.53160
59-l	2183	10205	48	359	**0.67608**	0.02833	**0.03818**
			160	164	**0.70386**	0.05991	**0.14332**
			256	129	**0.73295**	0.09367	**0.29363**
			2048	57	**0.98276**	1.23261	51.35861
128-l	2933	23618	48	1023	**0.69782**	0.07136	**0.01432**
			160	466	**0.71472**	0.17621	**0.04306**
			256	370	**0.68773**	0.25684	**0.06748**
			2048	204	1.22156	2.18591	2.41271
sm_part24_11756	11756	99713	48	5814	**0.71355**	0.44393	**0.00137**
			160	2436	**0.61702**	0.90976	**0.00307**
			256	1714	**0.61324**	1.83163	**0.00635**
			2048	431	**0.55256**	9.69627	**0.04347**
sm_part12_20753	20753	163514	48	8018	**0.70968**	0.76193	**0.00120**
			160	3331	**0.63255**	1.28946	**0.00225**
			256	2365	**0.59707**	2.20258	**0.00393**
			2048	567	**0.60512**	17.96227	**0.04103**
sm_part3_31854	31854	185662	48	9107	**0.71126**	1.03926	**0.00121**
			160	3902	**0.66474**	1.91767	**0.00247**
			256	2691	**0.65110**	2.64924	**0.00352**
			2048	646	**0.62056**	27.02443	**0.04377**
sm_part1_170859	170859	196836	48	11515	**0.82174**	3.33007	**0.00065**
			160	5104	**0.81209**	5.27129	**0.00107**
			256	3778	**0.76493**	7.45439	**0.00152**
			2048	956	**0.74339**	98.21790	**0.02070**

6 Conclusion and Future Work

In this paper, we designed and evaluated a set of greedy heuristics for the judicious hypergraph partitioning problem, namely an iterative improvement algorithm based on FM, and three initial partitioning strategies. We argued that greedy heuristics face severe challenges (such as equal gains/losses and scalability issues), and presented remedies such as randomization, tie-breaking and recursive partitioning. While these did not work as well as intended, we demonstrate nonetheless that combined with the multilevel framework, our algorithms are faster (two to four orders of magnitude) and yield substantially better solution quality (18% to 45%) than the previous state-of-the-art algorithm.

Future work should focus on ways to escape local minima during refinement (such as simulated annealing), ideas such as higher level gains as tie-breakers for the issue with many equal gains, as well as parallelization. Furthermore, we are interested in evaluating the impact of our approach on applications, such as the phylogenetic inference application that motivated HyperPhylo [2].

References

1. Charles J. Alpert. The ISPD98 Circuit Benchmark Suite. In *International Symposium on Physical Design (ISPD)*, pages 80–85, April 1998. `doi:10.1145/274535.274546`.
2. Ivo Baar, Lukas Hübner, Peter Oettig, Adrian Zapletal, Sebastian Schlag, Alexandros Stamatakis, and Benoit Morel. Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 175–184. IEEE, 2019. `doi:10.1109/IPDPSW.2019.00038`.
3. R Battiti, A Bertossi, and R Rizzi. Randomized Greedy Algorithms for the Hypergraph Partitioning Problem. *Randomization Methods in Algorithm Design, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 43:21–35, 1999. `doi:10.1090/dimacs/043/02`.
4. Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. The SAT Competition 2014. `http://www.satcompetition.org/2014/`, 2014.
5. Béla Bollobás and Alex D. Scott. Judicious Partitions of Hypergraphs. *Journal of Combinatorial Theory*, 78(1):15–31, 1997. `doi:10.1006/jcta.1996.2744`.
6. Béla Bollobás and Alex D. Scott. Judicious Partitions of 3-uniform Hypergraphs. *European Journal of Combinatorics*, 21(3):289–300, 2000. `doi:10.1006/eujc.1998.0266`.
7. Béla Bollobás and Alex D. Scott. Judicious partitions of bounded-degree graphs. *Journal of Graph Theory*, 46(2):131–143, 2004. `doi:10.1002/jgt.10174`.
8. Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. In *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`.
9. Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More Recent Advances in (Hyper) Graph Partitioning. *ACM Computing Surveys*, 2022. `doi:10.1145/3571808`.
10. Ümit V. Catalyurek and Cevdet Aykanat. Hypergraph-Partitioning-based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999. `doi:10.1109/71.780863`.
11. Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, November 2011. `doi:10.1145/2049662.2049663`.

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

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

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

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

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

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

18 Lars Gottesbüren. *Parallel and Flow-Based High-Quality Hypergraph Partitioning*. Dissertation, Karlsruhe Institute of Technology, 2022.

19 John Haslegrave. The Bollobás-Thomason conjecture for 3-uniform hypergraphs. *Combinatorica*, 32(4):451–471, 2012. doi:10.1007/s00493-012-2696-x.

20 Bruce Hendrickson and Robert W. Leland. A Multi-Level Algorithm For Partitioning Graphs. In Sidney Karin, editor, *Proceedings Supercomputing '95, San Diego, CA, USA, December 4-8, 1995*, page 28. ACM, 1995. doi:10.1145/224170.224228.

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

22 Jianfeng Hou, Shufei Wu, and Guiying Yan. On judicious partitions of uniform hypergraphs. *Journal of Combinatorial Theory*, 141:16–32, 2016. doi:10.1016/j.jcta.2016.02.004.

23 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.

24 Kassian Kobert, Alexandros Stamatakis, and Tomáš Flouri. Efficient detection of repeating sites to accelerate phylogenetic likelihood calculations. *Systematic biology*, 66(2):205–217, 2017.

25 Bernhard Misof, Shanlin Liu, Karen Meusemann, Ralph S Peters, Alexander Donath, Christoph Mayer, Paul B Frandsen, Jessica Ware, Tomáš Flouri, Rolf G Beutel, et al. Phylogenomics resolves the timing and pattern of insect evolution. *Science*, 346(6210):763–767, 2014.

26 Laura A. Sanchis. Multiple-Way Network Partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989. doi:10.1109/12.8730.

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

28 Farhad Shahrokhi and László A Székely. The complexity of the bottleneck graph bipartition problem. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 15(94):221–226, 1994.

29 Alexandros Stamatakis. Raxml version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 30(9):1312–1313, 2014. doi:10.1093/bioinformatics/btu033.

30 Alexandros Stamatakis and Nikolaos Alachiotis. Time and memory efficient likelihood-based tree searches on phylogenomic alignments with missing data. *Bioinformatics*, 26(12):i132–i139, 2010.

31 Tunzi Tan, Jihong Gui, Sainan Wang, Suixiang Gao, and Wenguo Yang. An Efficient Algorithm for Judicious Partition of Hypergraphs. In *Combinatorial Optimization and Applications - 11th International Conference, COCOA 2017, Shanghai, China, December 16-18, 2017, Proceedings, Part II*, volume 10628 of *Lecture Notes in Computer Science*, pages 466–474. Springer, 2017. doi:10.1007/978-3-319-71147-8_33.

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

Hierarchical Relative Lempel-Ziv Compression

Philip Bille
Department of Computer Science and Applied Mathematics,
Technical University of Denmark, Lyngby, Denmark

Inge Li Gørtz
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Simon J. Puglisi
Helsinki Institute for Information Technology (HIIT), Finland
Department of Computer Science, University of Helsinki, Finland

Simon R. Tarnow
DTU Compute, Technical University of Denmark, Lyngby, Denmark

Abstract

Relative Lempel-Ziv (RLZ) parsing is a dictionary compression method in which a string S is compressed relative to a second string R (called the reference) by parsing S into a sequence of substrings that occur in R. RLZ is particularly effective at compressing sets of strings that have a high degree of similarity to the reference string, such as a set of genomes of individuals from the same species. With the now cheap cost of DNA sequencing, such datasets have become extremely abundant and are rapidly growing. In this paper, instead of using a single reference string for the entire collection, we investigate the use of different reference strings for subsets of the collection, with the aim of improving compression. In particular, we propose a new compression scheme hierarchical relative Lempel-Ziv (HRLZ) which form a rooted tree (or hierarchy) on the strings and then compress each string using RLZ with parent as reference, storing only the root of the tree in plain text. To decompress, we traverse the tree in BFS order starting at the root, decompressing children with respect to their parent. We show that this approach leads to a twofold improvement in compression on bacterial genome datasets, with negligible effect on decompression time compared to the standard single reference approach. We show that an effective hierarchy for a given set of strings can be constructed by computing the optimal arborescence of a completed weighted digraph of the strings, with weights as the number of phrases in the RLZ parsing of the source and destination vertices. We further show that instead of computing the complete graph, a sparse graph derived using locality-sensitive hashing can significantly reduce the cost of computing a good hierarchy, without adversely effecting compression performance.

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

Keywords and phrases Relative compression, Lempel-Ziv compression, RLZ, LZ77, string collections, compressed representation, data structures, efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.18

Related Version *Previous Version*: https://arxiv.org/abs/2208.11371

Funding This work was supported in part by the Academy of Finland via grants 339070 and 351150 and the Danish Research Council via grant DFF-8021-002498.

1 Introduction

Given a collection of m strings $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of total length n, the *relative Lempel-Ziv (RLZ) compression scheme* parses each string S_i, $i > 1$, into a sequence of substrings (called *phrases*) of the string S_1 (we give a precise definition below). If the strings in S are highly similar, the number of phrases in the parsing is small relative to the total length of the collection. In order to achieve compression, the string S_1 is stored explicitly and the phrases

© Philip Bille, Inge Li Gørtz, Simon J. Puglisi, and Simon R. Tarnow;
licensed under Creative Commons License CC-BY 4.0
21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis; Article No. 18; pp. 18:1–18:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of the other strings are encoded by their starting and ending positions in S_1. To decompress, we simply replace the encoding of each phrase by the corresponding substring in S_1. Note that we can even efficiently support *sequence retrieval*, that is, decompressing a single specified string S_i from $\mathcal{S}$, by simply decompressing the phrases of the S_i independently of the rest of the collection.

RLZ is ideal for compressing and storing collections of highly similar strings while supporting efficient sequence retrieval. In particular, RLZ is a natural choice for databases of full genome sequences of individuals of the same species [18, 6, 5, 36, 24]. Since these sequences are highly similar, RLZ is able to compress them well, while still supporting efficient sequence retrieval needed by applications. Enormous reductions over the past two decades in the cost of DNA sequencing has led to large and growing data bases containing hundreds of thousands of full genome sequences of strains of many known bacteria and viruses. These databases are key, for example, to the field of genomic epidemiology, to screen patient samples (which are sequenced and compared against the genome database) for known pathogenic strains to arrive at diagnosis and suitable treatments (see, e.g., [21]).

For a given bacterial species, a genome database may contain genome sequences of thousands of different strains. While all these strains are relatively similar to each other, some share a higher degree of similarity with genomes in a cluster of related strains than they do with other sequences in the database. With this is mind, while RLZ may result in good compression when a single arbitrary sequence is selected as the reference, intuitively it would seem that even more effective compression of the database could be achieved by selecting a different reference for each cluster of strains.

Our Contributions. In this paper we explore the use of more than one reference sequence in the context of RLZ compression. We present a new compression scheme, called *hierarchical relative Lempel-Ziv (HRLZ) compression*, that arranges the sequences in $\mathcal{S}$ into a rooted tree H, with root r, such that each node v corresponds to a unique string $S(v)$ from $\mathcal{S}$. To compress the collection we greedily parse $S(v)$ wrt. $S(\text{parent}(v))$ using RLZ for each non-root node v. The compressed representation then consists of $S(r)$, the edges of H, and the encoding of the $m-1$ parsings of the non-root strings. Note that RLZ may be viewed as the special case of HRLZ on a tree consisting of a root with $m-1$ children.

A key challenge in HRLZ compression is finding a hierarchy for the collection that achieves strong compression. We show how to adapt an approach from delta compression of string collections [25] to our relative compression scenario. This leads to a number of interesting algorithmic challenges:

1. To derive an effective hierarchical arrangement for a given set of strings, we compute the optimal arborescence of a complete weighted digraph of the strings, with edge weights assigned as the number of phrases in the RLZ parsing of the source and destination vertices. We show that this scheme leads to a factor of 2 improvement to compression on bacterial genomes, and up to a factor 10 on viral genomes, without adversely affecting the speed of decompression.
2. While the optimal arborescence leads to pleasing compression improvements, it adds significantly to compression time. We show that by sparsifying the graph via locality-sensitive hashing, compression time can be kept reasonable, while not sacrificing compression gains.
3. Along the way, we describe an efficient implementation of the optimal arborescence algorithm of Tarjan [32] that uses a two-level heap engineered for efficient meld operations, which may be of independent interest.

Our resulting HRLZ compression scheme achieves improved tradeoffs for genome databases. While the time to compress the database with HRLZ is slower than RLZ, HRLZ always improves the compression ratio (measured by the number of phrases) in some cases enormously – by a factor of up to 19 times in our experiments. HRLZ also matches or even slightly improves whole database decompression time and achieves very similar single sequence retrieval times (the time taken to extract a single requested genome from the database) as RLZ does. Thus, in practical scenarios where space and sequence retrieval time are the main bottlenecks (such as a genomic database) HRLZ provides an attractive alternative to RLZ.

We also compared HRLZ to the classic Lempel-Ziv 77 (LZ77) compression scheme. LZ77 provides a natural lower on the number of phrases we can hope to achieve with RLZ and HRLZ and thus provides a baseline for the compression ratio achieved by those schemes. Our experiments show that while HRLZ is larger than LZ77 (by a factor of 2 to 16) on small collections, HRLZ is able to scale to collections with sizes well beyond those LZ77 is able to process, and is also orders of magnitude faster for sequence retrieval compared to LZ77 on the all datasets we tested.

Related Work. The idea of constructing a hierarchy of compressed sequences from collections was proposed in the context of *delta-compression* [25]. To the best of our knowledge, this idea has not been explored for RLZ compression. A related notion is mentioned cryptically in Storer and Szymanski [31], but appears never to have been implemented. The closest work we could find in the literature is due to Deorowicz and Grabowski [6], who describe an RLZ-based scheme for genome compression in which a sequence is compressed relative to multiple reference sequences, with each phrase storing which reference sequence it is from (see also [5]). Another more recent hierarchical compression scenario is the persistent strings model [3]. Both of these are quite different to the hierarchical arrangement of sequences we describe here.

Beyond genomics applications, RLZ has also found wider use as a compressor for large text corpora in contexts where random-access support for individual documents is needed [14, 34, 35, 26, 20, 2] and as a general data compressor [17, 16]. In those contexts, S_1 is usually first constructed using substrings sampled from other strings in the collection in a preprocessing phase (Hoobin et al. [14] show that random sampling of substrings works well). The structure of the RLZ parsing reveals a great deal about the repetitive structure of the string collection and several authors have shown that this can be exploited to design efficient compressed indexes for pattern matching [13, 8, 23]. More recently, the practical utility of RLZ as a more general tool for compressed data structuring has also been demonstrated, compressing suffix arrays [27, 29], document arrays [28] and various components of suffix trees [9].

Outline. In Section 2 we set down notation and basic concepts used throughout. In Section 3 we formally define hierarchical relative Lempel-Ziv compression, and then go onto describe efficient methods for computing it in Section 4 and Section 5. Section 6 describes our engineering of the arborescence algorithm of Tarjan [32]. Our experimental results on three genomic datasets are presented in Section 7, before conclusions and reflections are offered.

2 Basics

Throughout we will consider a *string* $S = S[1..n] = S[1]S[2]\dots S[n]$ on an integer alphabet Σ of σ symbols. The *substring* of S that starts at position i and ends at position j, $j \geq i$, denoted $S[i..j]$, is the string $S[i]S[i+1]\dots S[j]$. If $i > j$, then $S[i..j]$ is the empty string ε. A suffix of S is a substring with ending position $j = n$, and a prefix is a substring with starting position $i = 1$.

Parsings. A *parsing* of a string S wrt. a reference string R is a sequence of substrings of R – called phrases – $R[i_1, i_1 + l_1 - 1], R[i_2, i_2 + l_2 - 1], \ldots, R[i_z, i_z + l_z - 1]$ such that $S = R[i_1, i_1 + l_1 - 1] \cdot R[i_2, i_2 + l_2 - 1] \cdots R[i_z, i_z + l_z - 1]$. The *encoding* of a parsing consists of the sequence of starting indices and lengths of the phrases $(i_1, l_1), (i_2, l_2), \ldots, (i_z, l_z)$.

The *greedy parsing* of S wrt. R is the parsing obtained by processing S from left to right and choosing the longest possible phrase at each step. For example, let $R = actccta$ and $S = ctctcc$. The greedy parsing of S wrt. R gives the phrases $R[2, 4] = ctc$, $R[3, 5] = tcc$ and the encoding $(2, 3), (3, 3)$. We can construct the parsing in $O(|R| + |S|)$ time using a suffix tree.

Relative Lempel-Ziv Compression. Throughout the rest of the paper let $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ be a collection of m strings of total length $n = \sum_{i=1}^{m} |S_i|$. The *relative Lempel-Ziv* (RLZ) compression of $\mathcal{S}$ greedily parses each string S_i, $i > 1$, wrt. S_1. The *RLZ compressed representation* of $\mathcal{S}$ then consists of S_1 and the encoding of the parsings of each of the strings $S_2, \ldots, S_m$. For each string, we also save the number of phrases. In total, compression takes $O(n)$ time. Let z_i be the number of phrases in the parsing of S_i and let $z_R = \sum_{i=2}^{m} z_i$ denote the total number of phrases. The size of the RLZ compression is thus $O(|S_1| + z_R)$. Note that the size depends on the choice of the reference string (i.e. S_1) among the strings in $\mathcal{S}$. To decompress, we decode the phrases of each string using the explicitly stored reference string. This uses $O(\sum_{i=1}^{m} |S_i|) = O(n)$ time.

Throughout this paper, we use the number of phrases as the measure of compression. In a real compressor, the phrase positions and lengths undergo further processing in order to reduce the total number of bits used by the encoding (see, e.g., [11]). We remark that our hierarchical RLZ methods can be trivially adapted to use different encoding costs.

Graphs. Let G be a weighted directed strongly connected graph G. A *spanning arborescence* A of G with root r is a subgraph of G that is a directed rooted tree where all nodes are reachable from r. The weight of an arborescence S is the sum of the weight of the edges in A. A *minimum weight spanning arborescence* (MWSA) A is a spanning arborescence of minimum weight. Note that the root is not fixed in our definition and can thus be any node. For simplicity, we have assumed that G is strongly connected in our definition of MWSA since this is always the case in our scenario. Finally, for a node v in a tree, the parent of a node is denoted $\text{parent}(v)$.

3 Hierarchical Relative Lempel-Ziv Compression

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ be a collection of strings of total size n as above. We construct a rooted tree H, with root r, such that each node v represents a unique string $S(v)$ from $\mathcal{S}$. The *hierarchical relative Lempel-Ziv* (HRLZ) compression of $\mathcal{S}$ wrt. H greedily parses $S(v)$ wrt. $S(\text{parent}(v))$ for each non-root node v. In total, compression takes $O(n)$ time. The *HRLZ compressed representation* consists of $S(r)$, the edges of H, and the encoding of the $m - 1$ parsings of the non-root strings. Let $z_H = \sum_{v \in H \setminus \{r\}} z_v$, where z_v is the number of phrases in the parsing of $S(v)$. Thus the size of the HRLZ compression is $O(|H| + |S(r)| + z_H) = O(|S(r)| + z_H)$. Note that the size depends on the choice of tree and assignment of strings from $\mathcal{S}$ to the nodes.

To decompress, we traverse the tree in breadth first search (BFS) order from the root. We decode the string at each node using the string of the parent node by decoding each phrase. As the string of the parent node is always decoded before or explicitly stored as the root node, this uses $O(\sum_{i=1}^{m} |S_i|) = O(n)$ time. Note that the output order of the sequences can differ from their input order since they are recovered in BFS order of the hierarchy.

4 Constructing an Optimal Tree

We first give a simple and inefficient algorithm to construct an optimal tree for the HRLZ compression. The algorithm forms the basis of our efficient algorithm in the following section. Recall that the collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ consists of m strings of total size n. The algorithm proceeds as follows:

Step 1: Construct Cost Graph. We first construct a complete weighted directed graph G with m nodes numbered $\{1, \ldots, m\}$ called the *cost graph* of $\mathcal{S}$. Node i corresponds to the string S_i in $\mathcal{S}$ and the weight of edge (i, j) is the number of phrases in the greedy parsing of S_j wrt. S_i.

We have that G contains m nodes and m^2 edges. Computing the weight of edge (i, j) takes $O(|S_j|)$ time. Thus in total we use $O((m-1)\sum_j |S_j|) = O(nm)$ time. The space is $O(m^2)$.

Step 2: Construct Minimum Weight Spanning Arborescence. We then construct a MWSA A of the cost graph G using the algorithm by Tarjan [32, 4]. This uses $O(e \log m) = O(m^2 \log m)$ time. Here e denotes the number of edges in the graph.

Step 3: Construct Compressed Representation. Finally, we construct the HRLZ compression from the MWSA A. This uses $O(n)$ time.

In total the algorithm uses $O(mn)$ time and $O(m^2)$ space. Note that the algorithm constructs an optimal tree but not necessarily the optimal HRLZ compression since the HRLZ compression also needs to explicitly encode the string of the root of the tree. It is straightforward to include the cost of encoding the root string in the algorithm, by adding an additional virtual root s and adding edges (s, i) to every other node i, $1 \leq i \leq m$, with weight $|S_i|$. The MWSA of the new graph G' will be rooted in s and the unique edge out of s determines the optimal root string for HRLZ compression. While G' is not strongly connected, the MWSA is still well-defined and the MWSA algorithm produces the correct result in the same complexity. In practice, our datasets consist of very similar length strings and hence we have chosen not to implement this extension.

5 Sparsifying the Cost Graph via Locality-Sensitive Hashing

The main bottleneck in the simple algorithm from Section 4 is the construction of the complete cost graph in Step 1. In this section, we show how to efficiently sparsify the graph using locality-sensitive hashing.

We first construct a sparse subgraph $\overline{G}$ of the complete cost graph. We do this in rounds keeping an auxiliary set of strings R as follows. Initially, we set $\overline{G}$ to be the graph with m nodes and no edges, and $R = \mathcal{S}$. We repeat the following steps until $\overline{G}$ is strongly connected.

Step 1: Generate fingerprints. We first generate fingerprints for each string in R using locality-sensitive hashing. Our locality-sensitive hashing scheme is based on k-mers (a substring of length k) combined with min-hashing. More precisely, given parameters k and q we pick q hash functions $h_1, \ldots, h_q$ and hash each k-mer of each string S in R. The fingerprint of S is the sequence $\min_1, \ldots, \min_q$ where $\min_i$ is a minimum value hash obtained with h_i. For fast hashing we use the simple *multiply-shift* hashing scheme [7].

Step 2: Generate edges. Let C be a group of strings in R with the same fingerprint. If $|C| \leq T$ for a threshold $T \geq 2$ then for each ordered pair (i, j) of strings in C we add (i, j) to $\overline{G}$.

Step 3: Pruning R. After every c-th round for some tuneable parameter c we prune the set R as follows.

For every connected component in $\overline{G}$ pick the string s that has had the most collisions until now (the total number of collisions of a string s is equal to the sum of the size of the buckets it has been in). We then continue with R being the set of representatives. If $|R| \leq T$ then for each ordered pair (i, j) of strings in R we add (i, j) to $\overline{G}$.

Finally, we compute the weight of the edges of the strongly connected graph $\overline{G}$, i.e., for each edge (i, j) we compute the number of phrases in the greedy parsing of S_j wrt. S_i.

The computed cost graph is likely to be sparse and thus step 1 and step 2 of the algorithm from Section 4 will be much faster, leading to a much faster solution. Note that the constructed tree is no longer guaranteed to be optimal. We show experimentally in Section 7 that the size of the compression in nearly all cases is within 5% of optimal.

6 Speeding Up the Minimum Weight Spanning Arborescence Algorithm

We now show how to efficiently implement Step 2 of the algorithm from Section 4 on the sparse cost graph computed in Section 5.

Let $\overline{G}$ be the sparse cost graph with m nodes and e edges computed in Section 5. The MWSA algorithm by Tarjan [32, 4] uses m priority queues $Q_1, Q_2, \ldots, Q_m$, one for each node, where Q_i consists of all edges going into node v_i. The queues support the following operations:

- init(L): Constructs a queue Q containing all the elements in the list L.
- extract-min(Q_i): Returns and removes the minimum element in the queue Q_i.
- add(Q_i, c): Adds a constant c to the value of all elements in the queue Q_i.
- meld(Q_i, Q_j): Adds the elements from queue Q_j to the queue Q_i.

The MWSA algorithm uses a *pairing heap* [12] to support init in $O(|L|)$ time and the other operations in $O(\log m)$ time. The algorithm uses $O(m)$ meld and init operations, $O(e)$ add and extract-min operations, and the total length of the lists for the init operations is $O(e)$. Thus, the total run time of the queue operations in the MWSA algorithm is $O(e \log m)$, and this is also the total runtime of the MWSA algorithm.

We present a simple and practical alternative to the pairing heap that we call a *two-level heap*. Our two-level heap leads to a slightly worse theoretical bound of $O(e \log m + m \log^2 m)$ time for the MWSA algorithm. However, we have found that our implementation significantly outperforms the pairing heap in practice. We note that Larkin, Sen, and Tarjan include a similarly modified pairing heap as one of the variants in their empirical study of priority queues [19]. That study, however, neglects the meld operation, which is essential to our implementation of the minimum weight spanning aborescense algorithm described above. We therefore now describe our two-level heap implementation in full.

The two-level heap consists of a *top heap* t and a list of q *bottom heaps* $B = \{b_1, b_2, \ldots b_q\}$. All heaps are implemented using standard binary heaps [37]. Each heap h has an associated *offset* o_h, such that any stored element x in h represents that actual value $x + o_h$. The top heap t consists of the minimum element in each bottom heap $b \in B$. For each element in the top heap we also store which botton heap it is from. We implement each of the operations as follows.

init(L). We construct a two-level heap consisting of a single bottom heap $B = \{b\}$ containing the elements of L and a top heap t containing the minimum element of b. We set the offsets o_b and o_t of b and t, respectively, to be 0. This uses $O(|L|)$ time and hence the total time for init in the MWSA algorithm is $O(e)$.

extract-min(Q_i). We extract the minimum element x from the top heap t and return $x + o_t$. Let b be the bottom heap that stored x. We extract x from b, find the new minimum element y in b, and copy y into the top heap with offset o_b. This uses $O(\log m)$ time and hence the total time for extract-min in the MWSA algorithm is $O(e \log m)$.

add(Q_i, c). We add c to the offset of the top heap t, i.e., we set $o_t = o_t + c$. This takes constant time and hence the total time for add in the MWSA algorithm is $O(e)$.

meld(Q_i, Q_j). Let $Q_i = (t_i, B_i)$ and $Q_j = (t_j, B_j)$ be the two-level heaps that we want to meld. Let $|B_i|$ and $|B_j|$ be the number of bottom heaps associated with two-level heap Q_i and Q_j, respectively, and assume wlog. that $|B_i| \geq |B_j|$. We move each bottom heap $b \in B_j$ into B_i, insert the minimum element of b into t_i with offset $o_b + o_{t_j} - o_{t_i}$, and update the offset associated with b to $o_b = o_b + o_{t_j} - o_{t_i}$.

Each time an element in a bottom heap b is moved, we must insert the minimum element of b into a top heap using $O(\log m)$ time. We only move the bottom heaps of the two-level heap with the fewest bottom heaps and hence the number of times a bottom heap can be moved is $O(\log m)$. It follows that total time for meld in the MWSA algorithm is $O(m \log^2 m)$.

In total the MSWA algorithm implemented with the two-level heap uses $O(e \log m + m \log^2 m)$ time.

7 Experimental Results

We implemented the methods for building hierarchical references described in the previous sections and measured their performance on real biological data.

7.1 Setup

Experiments were run on Nixos 21.11 kernel version 5.10.115. The compiler was `g++` version 11.3.0 with `-Wall -Wextra -pedantic -O3 -funroll-loops -DNDEBUG -fopenmp -std=gnu++20` options. OpenMP version 4.5 was used to compute the string fingerprints in parallel and compute the edge weights on the cost graph. The CPU was an AMD Ryzen 3900X 12 Core CPU clocked at 4.1 GHz with L1, L2 caches of size 64KiB, 512KiB, per core respectively and a shared L3 cache of size 64MiB. The system had 32GiB of DDR4 3600 MHz memory. We recorded the CPU wall time using GNU time and `C++` chrono library. Source code is available on request.

7.2 Datasets

We evaluated our method using 1,000 copies of human chromosome 19 from the 1000 Genomes Project [33]; 219 *E. coli* genomes taken from the GenomeTrakr project [30], and 400,000 *SARS-CoV2* genomes from EBI's COVID-19 data portal [1]. See Table 1 for a brief summary of the datasets.

We also ran our tests on prefixes of various sizes of these datasets.

Table 1 Datasets used in experimentation. Columns labelled σ, n, and m, give the alphabet size, total collection size, and number of sequences, respectively. The final column shows the average sequence length, for convenience.

Name	Description	σ	n	m	n/m
E.coli	*E.coli* genomes	4	1,130,374,882	219	5,161,529
SARS-CoV2	Covid-19 genomes	5	11,949,531,820	400,000	29,873
chr19	Human chromosome 19 assemblies	5	59,125,151,874	1,000	59,125,151

7.3 Methods Tested

We included the following methods in our experimental evaluation.

RLZ. This corresponds to standard, single reference RLZ. Because the choice of reference can affect overall compression, we report results across a number of reference selections. Specifically, we randomly sampled roughly 0.5% of the sequences of each dataset, corresponding to 2, 2000, and 5 different reference sequences from *E. coli*, *SARS-CoV2*, and the *chr19* dataset respectively as our reference in RLZ.

Optimal HRLZ. This is the method described in Section 4, i.e., optimal hierarchical RLZ making use of full weight information.

Approximate HRLZ. The LSH variant of hierarchical RLZ as described in Section 5. Specifically, we used k-mers of 256 characters in size and choose the number of hash functions to $q = 4$. We pruned the set R every $c = 10$ rounds. We used $T = 2 \cdot \sqrt{m}$ as our threshold.

LZ. As a compression baseline, we also compute the full LZ77 parsing of our datasets using the KKP-SE external memory algorithm and software of [15][1]. Because it allows phrases to have their source at any previous position in the collection, computing the LZ77 parsing is more computationally demanding than RLZ parsing, and so we compute it only for some prefixes of the collections. For similar reasons, although in principle the above RLZ-based methods could attain parsings as small as the LZ77 parsing, we expect them not to.

7.4 Compression Performance

In this section, we compare the compression size as measured by the number of phrases generated by single reference RLZ to Optimal HRLZ and Approximate HRLZ on the datasets with LZ as a baseline. We also compare the compression time of the algorithms.

The results of our compression experiments are shown in Figure 1. Some of the results for the compression size of RLZ on the *SARS-CoV2* dataset have been left out of the figure because they were orders of magnitude larger than the other algorithms. Furthermore, we were unable to get data points for LZ on the 1000 sequences of the *chr19* dataset and the Optimal HRLZ on the *SARS-CoV2* for anything more than 125000 sequences on our test machine. The specific values behind Figure 1 (including the leftout results for RLZ) can be read in Tables 2–7 in the appendix.

[1] Code available at `https://www.cs.helsinki.fi/group/pads/em_lz77.html`.

We observe that, as expected LZ consistently produces the best compression size, but was infeasible to run on the full *chr19* dataset – which is the largest dataset measured in the number of symbols – on our test machine due to space consumption.

While RLZ outperforms all other algorithms in regard to compression speed it also consistently produces the worst compression size. Results on the *SARS-CoV2* dataset show just how bad the compression size can deteriorate if an ill-fitting reference is chosen as the reference used by RLZ. We observe that both versions of HRLZ obtain a better compression that RLZ in all cases. Measured in the number of phrases approximate HRLZ improves the compression ratio by a factor 1.8 on the *E. coli* dataset and 19.3 on the *SARS-CoV2* dataset.

For the Optimal HRLZ we see that the compression time is the worst of all the algorithms and grows quadratically with the number of sequences and quickly becomes infeasible on the *SARS-CoV2* dataset. It does however consistently produce a smaller compression size than RLZ, as we would expect given that HRLZ could produce a star graph with a single sequence as the reference for all other sequences and thus yielding the same result as RLZ. The compression time for Approximate HRLZ is significantly less than Optimal HRLZ and is consistently within a factor 2 of LZ – even outperforming LZ on some of the larger experiments of the *chr19* dataset. Furthermore, we note that the compression size of the Approximate HRLZ is no more than 15% greater than the compression size of Optimal HRLZ and always better than that of RLZ.

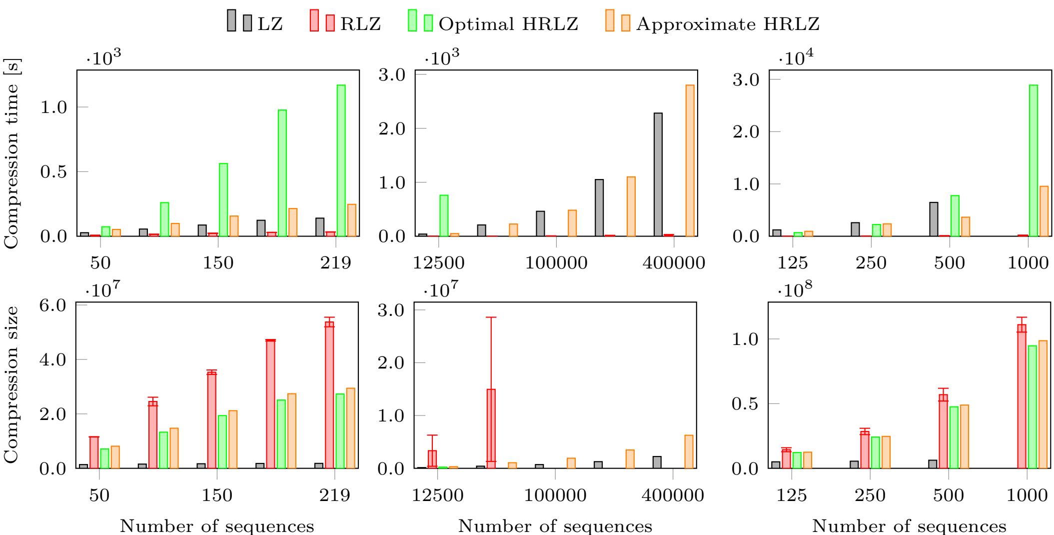


Figure 1 Compression time (top) and compression size (bottom) measured in the number of phrases as a function of the number of sequences in the in the *E. coli* (left), *SARS-CoV2* (center) and *chr19* (right) dataset. RLZ compression sizes were left out from the *SARS-CoV2* (center) plot because it is orders of magnitude larger than the rest of the compression sizes. We were unable to get data points for LZ on the 1000 sequences of the *chr19* dataset and Optimal HRLZ on the *SARS-CoV2* for anything more than 12500 sequences on our test machine due to memory consumption.

7.5 Decompression Performance

In this section, we compare the decompression time of the algorithms. The experiment measured the time to decompress the generated compressed dataset from our compression experiments and write the result to disk. This kind of streaming decompression use case is typ-

ical for, e.g., multi-pass index construction, machine learning, and data mining processes [10]. All methods have to write the same amount of data to storage when decompressing. Our HRLZ variants decompress sequences in BFS order according to hierarchy imposed on the sequences. This may require sequences that have previously written to disk being read back into memory (at most once) when they are needed as a reference in the decompression of other sequences.

We also compared the time it takes to decompress a single sequence for both variants of HRLZ and RLZ. We call this *sequence retrieval time*. For RLZ we took the best compressed sample for each experiment and used this for the test. We did not implement an equivalent solution for LZ since LZ is not built to easily decompress a single sequence. For each experiment we found the average and standard deviation of the sequence retrieval time over all sequences for that specific experiment.

The results of our decompression and sequence retrieval experiments are shown in Figure 2. We do not have data points for the experiments that we where unable to perform compression on. The specific values behind Figure 2 can be read in Tables 8–13 in the appendix. For Optimal HRLZ and Approximate HRLZ we also recorded the average and maximum node depth of the nodes in the arborescence. This can be viewed in table 3 in the appendix.

We observe that RLZ and the HRLZ variants have similar decompression performance characteristics, while LZ performs significantly worse. Interestingly, both versions of HRLZ outperformed RLZ in decompression time on most of the experiments. We believe this is because the longer phrases produced by the HRLZ variants result in fewer cache misses; that is, for every access to the reference made by HRLZ, more symbols are sequentially copied, improving cache performance and overall runtime. We believe that this also explains why both variants of HRLZ perform within a factor of 5 with respect to RLZ in sequence retrieval time, even though the average node depth in the minimum spanning arborescence on the largest *SARS-CoV2* dataset experiment is 100.

On larger datasets, which approach the size of RAM on the test machine, RLZ and HRLZ have similar decompression times, with RLZ being occasionally faster (on the largest of the *SARS-CoV2* datasets, for example). This can be explained by the above mentioned need for the HRLZ variants to read previously written sequences back into memory (when they are needed as reference sequences).

8 Concluding Remarks

We have shown that, from a space point of view, traditional single-reference RLZ compression can be significantly outperformed by imposing a hierarchy on the sequences to be compressed using a sequence's parent in the hierarchy as its reference sequence. Moreover, we have described efficient methods by which hierarchies can be efficiently obtained. Our experiments show that the time to subsequently decompress the set of sequences are at worst negligibly slower, and many times even faster than the single reference baseline.

There are many directions future work could take. Apart from compression, another feature of RLZ that makes it attractive in a genomic context is its ready support for efficient random access to individual sequences (and indeed substrings): having a compact, easily accessible representation of the genome sequences compliments popular indexing methods that do not readily support random access themselves (e.g., [22]). Supporting random access at a substring level for HRLZ compressed data (as opposed to sequence-level access we currently support) is an interesting avenue for future work.

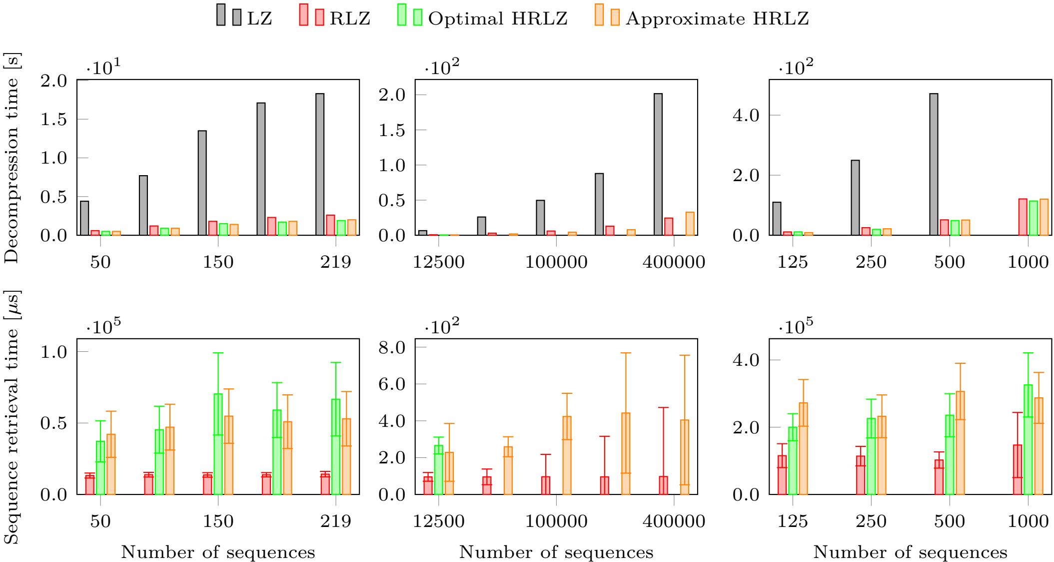


Figure 2 Decompression time (top) and sequence retrieval time (bottom) as a function of the number of sequences in the *E. coli* (left), *SARS-CoV2* (center) and *chr19* (right) dataset. Note that the missing samples are due to no compressed result being available.

As noted in the introduction, several recent works have demonstrated the practical utility of using RLZ as a tool for compressed data structuring [9, 27, 29, 28]. In that context, an *artificial* reference sequence is constructed from repeated pieces (e.g., subarrays, or subtrees) of the data structure to be compressed. It would be interesting to see if our methods could be adapted to construct better artificial reference sequences for use in those scenarios.

Finally, in this work we have used purely algorithmic methods to derive hierarchies for datasets: no biological characteristics of the sequences have been used. However, the field of phylogenetics has developed many techniques for imposing a hierarchy on a set of individuals based on biologically meaningful features in their genomic content. It may be interesting to examine any similarities between phylogenetic trees and our RLZ-based hierarchies, and whether phylogenetic trees may serve as good hierarchies in the context of compression.

References

1 Coronavirus genomes – NCBI datasets. Accessed 18/05/2022, `https://www.ncbi.nlm.nih.gov/datasets/coronavirus/genomes/`.

2 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic relative compression, dynamic partial sums, and substring concatenation. *Algorithmica*, 80(11):3207–3224, 2018.

3 Philip Bille and Inge Li Gørtz. Random access in persistent strings. In *Proc. 31st ISAAC*, 2020.

4 P. M. Camerini, L. Fratta, and F. Maffioli. A note on finding optimum branchings. *Networks*, 9(4):309–312, 1979. `doi:10.1002/net.3230090403`.

5 Sebastian Deorowicz, Agnieszka Danek, and Szymon Grabowski. Genome compression: a novel approach for large collections. *Bioinformatics*, 29(20):2572–2578, 2013.

6 Sebastian Deorowicz and Szymon Grabowski. Robust relative compression of genomes with random access. *Bioinformatics*, 27(21):2979–2986, 2011.

7 Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.

8 Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative lempel-ziv self-index for similar sequences. *Theor. Comput. Sci.*, 532:14–30, 2014.

9 Andrea Farruggia, Travis Gagie, Gonzalo Navarro, Simon J. Puglisi, and Jouni Sirén. Relative suffix trees. *Comput. J.*, 61(5):773–788, 2018.

10 Paolo Ferragina and Giovanni Manzini. On compressing the textual web. In *Proc. 3rd WSDM*, pages 391–400, 2010.

11 Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the bit-complexity of Lempel-Ziv compression. *SIAM J. Comput.*, 42(4):1521–1541, 2013.

12 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):111–129, 1986.

13 Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A faster grammar-based self-index. In *Proc. 6th LATA*, pages 240–251, 2012.

14 Christopher Hoobin, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proc. VLDB Endowment*, 5(3):265–273, 2011.

15 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Proc. 24th DCC*, pages 153–162, 2014.

16 Dominik Kempa and Ben Langmead. Fast and space-efficient construction of AVL grammars from the LZ77 parsing. In *Proc. 29th ESA*, pages 56:1–56:14, 2021.

17 Dmitry Kosolobov, Daniel Valenzuela, Gonzalo Navarro, and Simon J. Puglisi. Lempel-ziv-like parsing in small space. *Algorithmica*, 82(11):3195–3215, 2020.

18 Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th SPIRE*, pages 201–206, 2010.

19 Daniel H. Larkin, Siddhartha Sen, and Robert Endre Tarjan. A back-to-basics empirical study of priority queues. In *Proc. 16th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 61–72. SIAM, 2014.

20 Kewen Liao, Matthias Petri, Alistair Moffat, and Anthony Wirth. Effective construction of relative lempel-ziv dictionaries. In *Proc. 25th WWW*, pages 807–816, 2016.

21 Tommi Mäklin, Teemu Kallonen, Jarno Alanko, Ørjan Samuelsen, Kristin Hegstad, Veli Mäkinen, Jukka Corander, Eva Heinz, and Antti Honkela. Bacterial genomic epidemiology with mixed samples. *Microbial Genomics*, 7(11), 2021.

22 Taher Mun, Alan Kuhnle, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Matching reads to many genomes with the r-index. *J. Comput. Biol.*, 27(4):514–518, 2020.

23 Gonzalo Navarro and Victor Sepulveda. Practical indexing of repetitive collections using relative Lempel-Ziv. In *Proc. 29th DCC*, pages 201–210, 2019.

24 Gonzalo Navarro, Victor Sepulveda, Mauricio Marín, and Senén González. Compressed filesystem for managing large genome collections. *Bioinformatics*, 35(20):4120–4128, 2019.

25 Zan Ouyang, Nasir Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *Proc. 3rd WISE*, pages 257–266, 2002.

26 Matthias Petri, Alistair Moffat, P. C. Nagesh, and Anthony Wirth. Access time tradeoffs in archive compression. In *Proc. 11th AIRS*, pages 15–28, 2015.

27 Simon J. Puglisi and Bella Zhukova. Relative Lempel-Ziv compression of suffix arrays. In *Proc. SPIRE*, LNCS 12303, pages 89–96. Springer, 2020.

28 Simon J. Puglisi and Bella Zhukova. Document retrieval hacks. In *Proc. 19th SEA*, pages 12:1–12:12, 2021.

29 Simon J. Puglisi and Bella Zhukova. Smaller RLZ-compressed suffix arrays. In *Proc. 31st DCC*, 2021.

30 E.L. Stevens et al. The public health impact of a publically available, environmental database of microbial genomes. *Front. Microbiol.*, 8(808), 2017.

31 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

32 R. E. Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977. doi:10.1002/net.3230070103.

33 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015.

34 Jiancong Tong, Anthony Wirth, and Justin Zobel. Compact auxiliary dictionaries for incremental compression of large repositories. In *Proc. 23rd CIKM*, pages 1629–1638, 2014.

35 Jiancong Tong, Anthony Wirth, and Justin Zobel. Principled dictionary pruning for low-memory corpus compression. In *Proc. 37th SIGIR*, pages 283–292, 2014.

36 Daniel Valenzuela, Tuukka Norri, Niko Välimäki, Esa Pitkänen, and Veli Mäkinen. Towards pan-genome read alignment to improve variation calling. *BMC Genom.*, 19(S2), 2018.

37 John William Joseph Williams. Algorithm 232: heapsort. *Commun. ACM*, 7:347–348, 1964.

A Additional Figures

This appendix contains additional plots as well as data used to generate plots in the main document.

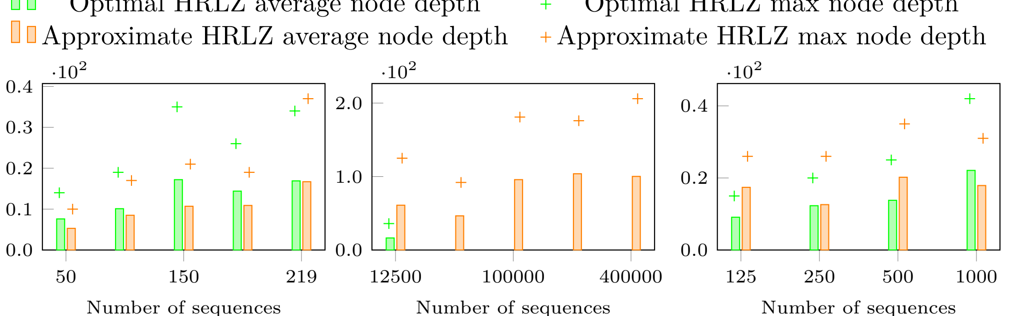


Figure 3 The average node depth and maximum node depth for the generated rooted tree for HRLZ as a function of the number of sequences in the *E. coli* (left), *SARS-CoV2* (center) and *human chromosome 19* (right) dataset.

Table 2 Number of phrases generated for each algorithm as a function of the number of sequences on the *E. coli* dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
50	1379061	11562810.5	19485.7	7119429	8136762
100	1556541	24551113.5	1575921.1	13284440	14736830
150	1681574	35311948.5	820511.9	19369695	21180548
200	1787317	47030288.0	281705.7	25087136	27401597
219	1815168	53757548.5	1759746.2	27307037	29425274

Table 3 Compression time in seconds for each algorithm as a function of the number of sequences on the *E. coli* dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
50	27.4	7.2	0.1	73.4	52.5
100	55.7	15.1	0.6	259.9	98.7
150	86.8	23.0	0.6	562.3	156.3
200	123.4	29.5	0.0	976.3	214.4
219	140.0	32.9	0.8	1169.2	246.9

Table 4 Number of phrases generated for each algorithm as a function of the number of sequences on the *SARS-CoV2* dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
12500	133149	3339721.0	2948529.2	227037	308499
50000	408189	14958546.9	13657427.6	nan	1061000
100000	714174	29867903.0	27657360.9	nan	1919686
200000	1262731	62437789.1	63333546.4	nan	3483747
400000	2235801	120997669.9	112955869.2	nan	6260161

Table 5 Compression time in seconds for each algorithm as a function of the number of sequences on the *SARS-CoV2* dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
12500	40.1	0.5	0.3	759.6	49.0
50000	209.5	2.3	1.6	nan	226.8
100000	462.5	4.5	3.3	nan	482.5
200000	1050.5	9.5	8.0	nan	1099.8
400000	2281.7	18.5	13.9	nan	2800.4

Table 6 Number of phrases generated for each algorithm as a function of the number of sequences on the human chromosome 19 dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
125	5024871	14364972.4	1550636.6	12196621	12470163
250	5526491	28489855.6	2453053.5	24174861	24688193
500	6263992	56926699.6	4927584.0	47547689	48888371
1000	nan	111021292.4	5760037.6	94684015	98637301

Table 7 Compression time in seconds for each algorithm as a function of the number of sequences on the human chromosome 19 dataset.

Size	lz	rlz avg.	rlz stdev.	opt-hrlz	approx-hrlz
125	1215.0	24.8	1.8	694.7	941.9
250	2591.2	46.3	3.3	2238.0	2366.1
500	6463.0	89.1	5.9	7765.9	3641.0
1000	nan	187.4	6.8	28886.1	9546.1

Table 8 Decompression time in seconds as a function of the number of sequences in the *E. coli* dataset.

Size	lz	rlz	opt-hrlz	approx-hrlz
50	4.4	0.576	0.466	0.497
100	7.7	1.193	0.902	0.942
150	13.5	1.771	1.479	1.407
200	17.1	2.257	1.719	1.782
219	18.3	2.642	1.913	1.988

Table 9 Decompression time in seconds as a function of the number of sequences in the *SARS-CoV2* dataset.

Size	lz	rlz	opt-hrlz	approx-hrlz
12500	6.6	0.680	0.522	0.483
50000	26.0	2.360	nan	1.900
100000	49.7	4.492	nan	4.366
200000	87.9	13.252	nan	7.852
400000	201.6	26.827	nan	32.835

Table 10 Decompression time in seconds as a function of the number of sequences in the human chromosome 19 dataset.

Size	lz	rlz	opt-hrlz	approx-hrlz
125	110.0	11.071	11.186	8.759
250	249.6	25.200	19.592	21.600
500	471.5	51.297	49.011	50.697
1000	nan	120.984	113.942	120.307

Table 11 Sequence retrieval time in microseconds as a function of the number of sequences in the *E. coli* dataset.

Size	rlz avg.	rlz stdev.	opt-hrlz avg.	opt-hrlz stdev.	approx-hrlz avg.	approx-hrlz stdev.
50	13236.5	1755.6	37178.5	14394.6	42121.3	16156.5
100	13821.5	1620.0	45319.2	16403.6	47117.6	16014.0
150	13676.9	1570.5	70350.0	28741.0	54822.1	19014.8
200	13793.4	1494.6	59070.6	19208.5	50908.6	18783.8
219	14224.6	1889.1	66639.9	25731.6	52970.7	19047.7

Table 12 Sequence retrieval time in microseconds as a function of the number of sequences in the *SARS-CoV2* dataset.

Size	rlz avg.	rlz stdev.	opt-hrlz avg.	opt-hrlz stdev.	approx-hrlz avg.	approx-hrlz stdev.
12500	95.5	23.7	265.7	45.7	228.5	156.8
50000	95.5	42.7	nan	nan	259.0	54.3
100000	96.2	121.4	nan	nan	423.4	125.8
200000	95.8	220.0	nan	nan	442.3	326.5
400000	97.2	375.0	nan	nan	404.3	351.8

Table 13 Sequence retrieval time in microseconds as a function of the number of sequences in the human chromosome 19 dataset.

Size	rlz avg.	rlz stdev.	opt-hrlz avg.	opt-hrlz stdev.	approx-hrlz avg.	approx-hrlz stdev.
125	115321.9	35591.1	200032.1	40195.9	272391.0	69385.1
250	114071.5	28891.4	225877.9	57460.7	232411.1	63591.6
500	102449.2	24080.0	235679.4	64040.3	306264.9	83814.0
1000	146999.9	96793.4	325701.5	95408.6	287200.1	75598.3

Table 14 The average node depth and maximum node depth for the generated rooted tree for HRLZ as a function of the number of sequences in the *E. coli* dataset.

Size	opt-hrlz avg.	opt-hrlz max	approx-hrlz avg.	approx-hrlz max
50	7.6	14	5.3	10
100	10.1	19	8.5	17
150	17.2	35	10.7	21
200	14.4	26	10.9	19
219	16.9	34	16.7	37

Table 15 The average node depth and maximum node depth for the generated rooted tree for HRLZ as a function of the number of sequences in the *SARS-CoV2* dataset.

Size	opt-hrlz avg.	opt-hrlz max	approx-hrlz avg.	approx-hrlz max
12500	16.4	36	61.0	125
50000	nan	nan	46.5	92
100000	nan	nan	95.8	181
200000	nan	nan	103.8	176
400000	nan	nan	100.2	206

Table 16 The average node depth and maximum node depth for the generated rooted tree for HRLZ as a function of the number of sequences in the human chromosome 19 dataset.

Size	opt-hrlz avg.	opt-hrlz max	approx-hrlz avg.	approx-hrlz max
125	9.1	15	17.4	26
250	12.3	20	12.6	26
500	13.8	25	20.2	35
1000	22.1	42	17.9	31

Exact and Approximate Range Mode Query Data Structures in Practice

Meng He
Dalhousie University, Halifax, Canada

Zhen Liu
Dalhousie University, Halifax, Canada

Abstract

We conduct an experimental study on the range mode problem. In the exact version of the problem, we preprocess an array A, such that given a query range $[a, b]$, the most frequent element in $A[a, b]$ can be found efficiently. For this problem, our most important finding is that the strategy of using succinct data structures to encode more precomputed information not only helped Chan et al. (Linear-space data structures for range mode query in arrays, Theory of Computing Systems, 2013) improve previous results in theory but also helps us achieve the best time/space tradeoff in practice; we even go a step further to replace more components in their solution with succinct data structures and improve the performance further.

In the approximate version of this problem, a $(1+\varepsilon)$-approximate range mode query looks for an element whose occurrences in $A[a, b]$ is at least $F_{a,b}/(1+\varepsilon)$, where $F_{a,b}$ is the frequency of the mode in $A[a, b]$. We implement all previous solutions to this problems and find that, even when $\varepsilon = \frac{1}{2}$, the average approximation ratio of these solutions is close to 1 in practice, and they provide much faster query time than the best exact solution. These solutions achieve different useful time-space tradeoffs, and among them, El-Zein et al. (On Approximate Range Mode and Range Selection, 30th International Symposium on Algorithms and Computation, 2019) provide us with one solution whose space usage is only 35.6% to 93.8% of the cost of storing the input array of 32-bit integers (in most cases, the space cost is closer to the lower end, and the average space cost is 20.2 bits per symbol among all datasets). Its non-succinct version also stands out with query support at least several times faster than other $O(\frac{n}{\varepsilon})$-word structures while using only slightly more space in practice.

2012 ACM Subject Classification Information systems → Data structures

Keywords and phrases range mode query, exact range mode query, approximate range mode query

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.19

Supplementary Material *Software (Source Code)*: `https://github.com/Kolento777/RangeModeQueries`, archived at `swh:1:dir:5d61144576ed7d45a2e424ae08b6b010c1a6e90c`

Funding This work is supported by NSERC.

1 Introduction

The *mode*, or the most frequent element, in a dataset is a widely used descriptive statistic. In the *range mode query problem*, we preprocess an array A of length n, such that, given a query range $[a, b]$, the *mode* in $A[a, b]$ can be computed efficiently. Many problems in data analytics and retrieval can be abstracted to range mode. For example, an online shopping platform may be interested in the most popular item purchased by customers over a certain period, which can be found by a range mode query over the sales records in its database.

Range mode is also connected to matrix multiplication; the product of two $\sqrt{n} \times \sqrt{n}$ Boolean matrices can be computed by answering n range mode queries in an array of length $\mathcal{O}(n)$ [7]. This reduction provides a conditional lower bound showing that, with current knowledge, the time required to preprocess an array and answer n range mode queries must be $\Omega(n^{\omega/2})$, where $\omega < 2.3726$ is the best exponent in matrix multiplication [2].

© Meng He and Zhen Liu;
licensed under Creative Commons License CC-BY 4.0

Furthermore, since the best combinatorial algorithm for Boolean matrix multiplication is only a polylogarithmic factor better than cubic [4], with current knowledge, we cannot use pure combinatorial approaches to solve range mode in $O(n^{3/2-\delta})$ preprocessing time and $O(n^{1/2-\delta})$ query time simultaneously for any constant $\delta \in (0, 1/2)$. To speed up queries, researchers further define the $(1+\varepsilon)$-*approximate range mode query problem*, where $\varepsilon \in (0, 1)$. Given a query range $[a, b]$, let $F_{a,b}$ denote the frequency of the mode in $A[a, b]$. A $(1+\varepsilon)$-approximate range mode query then asks for an element whose occurrences in $A[a, b]$ is at least $F_{a,b}/(1+\varepsilon)$.

Due to the importance in both theory and practice, range mode has been studied extensively [22, 29, 7, 6, 18, 12, 13, 32, 31, 19]. Despite these efforts, we are not aware of any experimental studies on them. Hence, to connect theory to practice, we conduct an empirical study of exact and approximate range mode structures using large practical datasets.

Related Work. Krizanc et al. [22] first considered the exact range mode problem and introduced an $\mathcal{O}(n + s^2)$-word solution with $\mathcal{O}((n/s) \lg n)$ query time for any $s \in [1, n]$, and setting $s = \sqrt{n}$ yields a linear space solution with $\mathcal{O}(\sqrt{n} \lg n)$ query time. They also presented another solution with constant query time and $\mathcal{O}(n^2 \lg \lg n / \lg n)$ words of space cost. Later Petersen et al. [29] proposed an $\mathcal{O}(n^2 \lg \lg n / \lg^2 n)$-word structure with constant query time. Chan et al. [7] further improved the time-space tradeoff of Krizanc et al. by designing an $O(n + s^2/w)$-word data structure with $O(n/s)$ query time, where w is the number of bits in a word. This result implies a linear space solution in words with $\mathcal{O}(\sqrt{n/w})$ query time.

Regarding $(1+\varepsilon)$-approximate range mode, Bose et al. [6] first used persistent search trees to design an $O(\frac{n}{\varepsilon})$-word solution with $\mathcal{O}(\lg \lg n + \lg \frac{1}{\varepsilon})$ query time. Greve et al. [18] provided another structure with $\mathcal{O}(\lg \frac{1}{\varepsilon})$ query time and $\mathcal{O}(\frac{n}{\varepsilon})$ words of space, and they used succinct data structures. More recently, El-Zein et al. [12] designed an encoding data structure occupying only $\mathcal{O}(\frac{n}{\varepsilon})$ bits, and without accessing the original array, it can also report the position of a $(1+\varepsilon)$-approximate mode in the query range in $\mathcal{O}(\lg \frac{1}{\varepsilon})$ time.

Our Work. We first study linear-space exact range mode structures [22, 7]. Much of this study focuses on these two data structures of Chan et al. [7]: a simple linear word structure with $\mathcal{O}(\sqrt{n})$ query time, and a linear word structure with $\mathcal{O}(\sqrt{n/w})$ query time. They both outperform other previous exact solutions, and the latter, which is their final structure, essentially combines the former with succinct data structures to encode more precomputed information. However, in practice, constant-time operations over succinct data structures are usually slower than operations over their non-succinct counterparts when all solutions fit in memory [15, 9, 27, 3]. To see whether the use of succinct data structures by Chan et al. improves performance in practice, we compare different tradeoffs of both structures and find that, when the same amount of space is used, the latter indeed provides much faster query support than the former. This is because the query algorithm only performs a constant number of succinct structure operations, and their execution time is dominated by other steps. Encouraged by this observation, we further use succinct structures to swap out more components, and our variant achieves even better time/space tradeoffs. These results are exciting, as they confirm that, when the same space cost is incurred, careful use of succinct data structures may potentially improve query efficiency in practice.

Regarding $(1+\varepsilon)$-approximate range mode, we focus on solutions by Bose et al. [6], Greve et al. [18] and El-Zein et al. [13], as well as a non-succinct version of the $\mathcal{O}(\frac{n}{\varepsilon})$-bit encoding structure of El-Zein et al. which stores the sequences they encode succinctly in plain arrays instead. When setting $\varepsilon = 1/2$, all these data structures provide much faster query time than

the best exact solution (which already answers a query in microseconds), and the average approximation ratio is between 1.00001 and 1.02630. They also typically use less than $5n$ words and are thus excellent solutions when high average quality of answers is sufficient. When encoded using compressed bit vectors, the space cost of the succinct encoding structure of El-Zein et al. [13] is only 35.6% to 93.8% of the input array of 32-bit integers (the average space cost is 20.2 bits per symbol among all datasets). Its non-succinct version also stands out with query support at least several times faster than other $O(\frac{n}{\varepsilon})$-word structures while using only slightly more space. When decreasing ε to improve worst-case approximation, query times increase at a logarithmic rate, but space costs tend to be proportional to $1/\varepsilon$.

2 Data Structure for Range Mode

We review the data structures that we will implement. When describing them, we adopt the word RAM model with word size w bits and assume that the input is an array $A[1..n]$ of integers from $\{1, 2, \ldots, \Delta\}$, where $\Delta \leq n$. Some solutions use succinct bit vectors as building blocks. These operations are defined over a bit vector $B[1..n]$: $\texttt{rank}_b(i)$, which returns the frequency of bit $b \in \{0, 1\}$ in $B[1..i]$, and $\texttt{select}_b(i)$, which returns the index of the i-th occurrence $b \in \{0, 1\}$ in B. Pătraşcu [28] showed how to represent B in $\lg \binom{n}{t} + \mathcal{O}(\frac{n}{\lg^c n}) \leq n + \mathcal{O}(\frac{n}{\lg^c n})$ bits, where t is the number of 1s in B and c is an arbitrary positive constant, to support rank and select in $\mathcal{O}(1)$ time. A folklore approach encodes a monotonically increasing sequence of n nonnegative integers upper bounded by u by encoding the difference between consecutive elements in unary and performs rank and select operations over the concatenated bit vector to compute any entry. This lemma summarizes its bounds.

▶ **Lemma 1** (folklore). *A monotonically increasing sequence of n nonnegative integers upper bounded by u can be represented in $\lg \binom{n+u}{n} + \mathcal{O}(\frac{n+u}{\lg^c (n+u)}) \leq n + u + \mathcal{O}(\frac{n+u}{\lg^c (n+u)})$ bits for any positive constant c such that any entry in the sequence can be computed in $\mathcal{O}(1)$ time.*

2.1 Exact Range Mode in Linear Space and $\mathcal{O}(\sqrt{n} \lg n)$ Time

To design a solution, Krizanc et al. [22] divide A into s blocks each of size $\lceil \frac{n}{s} \rceil$ for an integer parameter $s \in [1, n]$ and precompute an $s \times s$ table S. For any integers $i, j \in [1, s]$, $S[i, j]$ stores the mode of the subarray consisting of blocks $i, i+1, \ldots, j$. They also construct, for each integer $\alpha \in \{1, 2, \ldots, \Delta\}$, a sorted array Q_α of the positions of the occurrences of α in array A. All these structures occupy $\mathcal{O}(n + s^2)$ words and can be built in $\mathcal{O}(ns)$ time. With them, the mode in $A[a, b]$ can be computed by decomposing $[a, b]$ into up to three subranges: the *span* consists of all the blocks that are entirely contained in $[a, b]$, while the *prefix* and the *suffix* are the two subranges of $[a, b]$ before and after the span, respectively. The mode, c, of the span can be retrieved from S in $\mathcal{O}(1)$ time. The answer to the query is either c, or an element in the prefix or the suffix. We call each of these up to $2\lceil \frac{n}{s} \rceil - 1$ elements a candidate, and the frequency of each candidate $A[x]$ in the query range is computed by a binary search in $Q_{A[x]}$. Then the total query time is $\mathcal{O}((n/s) \lg n)$. Hence, setting $s = \lceil \sqrt{n} \rceil$ yields a linear-word structure with $\mathcal{O}(\sqrt{n} \lg n)$ query time and $\mathcal{O}(n^{3/2})$ preprocessing time.

2.2 Exact Range Mode in Linear Space and $\mathcal{O}(\sqrt{n})$ Time

Chan et al. [7] improved the solution of Krizanc et al. [22] by constructing two additional data structures: A rank array A' in which $A'[i]$ is the index of the entry of $Q_{A[i]}$ that stores i, and an additional $s \times s$ table S' in which $S'[i, j]$ stores the frequency of the mode in blocks $i, i+1, \ldots, j$. With the addition of A', we can determine, in constant time, whether $A[i]$ occurs at least q times in $A[i..j]$ for any given i, j and q, by checking if $Q_{A[i]}[A'[i] + q - 1] \leq j$.

The query algorithm again decomposes the query range $[a, b]$ into the span, the prefix and the suffix. Using S and S', we can find the mode, c, of the span and its frequency, f_c, in the span in $\mathcal{O}(1)$ time. This is one candidate of the mode in $A[a, b]$. We then look for the elements in the prefix or the suffix whose frequencies in $A[a, b]$ are greater than f_c: We scan the prefix, and for each element $A[x]$ in it, we find out whether we have seen it before by checking whether $Q_{A[x]}(A'[x] - 1)$ is at least a. If not, we determine whether $A[x]$ occurs more than f_c times in $A[x, b]$ in $O(1)$ time by the approach discussed before. If it does, then $A[x]$ is a candidate, and we compute its frequency in $A[a, b]$ by skipping the next $f_c - 1$ occurrences in $Q_A[x]$ and then continuing the scan of $Q_A[x]$ to find its remaining occurrences in $A[a, b]$. Since the number of times that $A[x]$ occurs in the span is at most f_c, the number of scanned entries of $Q_{A[x]}$ is at most the number of occurrences of $A[x]$ in the prefix and the suffix. Therefore, the frequencies of all candidates can be computed in time linear in the lengths of the prefix and the suffix, which is $O(n/s)$. We scan the suffix in a similar manner, and the candidate with the highest frequency in $A[a, b]$ is the answer. This way the query time is improved to $O(n/s)$, implying a linear-word tradeoff with $\mathcal{O}(\sqrt{n})$ query time.

2.3 Exact Range Mode in Linear Space and $\mathcal{O}(\sqrt{n/w})$ Time

The final solution of Chan et al. [7] divides the input array A into two subsequences B_1 and B_2 as follows: We scan A. If the current element appears at most s times in A, we append it to B_1. Otherwise, it is appended to B_2. Additionally, we define two $2 \times n$ tables $I_\beta[i]$ and $J_\beta[i]$, in which, for every $\beta \in [1, 2]$ and each $i \in [1, n]$, $I_\beta[i]$ (or $J_\beta[i]$) stores the index in B_β of the closest element in A to the left (or right) of $A[i]$ that lies in B_β. Then a range mode query in A can be answered by querying both B_1 and B_2.

A compact version of the structure in Section 2.2 is built over B_1 which consists of Q_α for each α and a compact encoding of S' in $\mathcal{O}(s^2)$ bits, or $\mathcal{O}(s^2/w)$ words. The latter uses Lemma 1 to encode each row of S' in $\mathcal{O}(s)$ bits, as it contains at most s positive integers upper bounded by s. Furthermore, Chan et al. use this structure to infer any entry of S in $\mathcal{O}(n/s)$ time without storing S. This decreases storage to $\mathcal{O}(n + s^2/w)$ words and can answer range mode over B_1 in $\mathcal{O}(n/s)$ time. As for B_2, since each element occurs more than s times, the number of distinct elements, Δ', is at most n/s. They mark every Δ' positions in B_2 and use n words to encode the number of occurrences of each distinct element from the start of B_2 to each marked position, so that the frequency of any element between two marked positions can be computed in $\mathcal{O}(1)$ time. Together with a walk from each endpoint of the query range $[a, b]$ to the nearest marked position inside $[a, b]$, we can compute the frequencies of all Δ' distinct elements in $[a, b]$ in $O(\Delta')$ time, thus answering range mode over B_2. Combing the structures for B_1 and B_2, we have an $\mathcal{O}(n + s^2/w)$-word structure with $\mathcal{O}(n/s)$ query time and $\mathcal{O}(ns + n\lg(n/s))$ preprocessing time. Setting $s = \lceil\sqrt{nw}\rceil$ yields a linear word structure with $\mathcal{O}(\sqrt{n/w})$ query time and $\mathcal{O}(n^{3/2}\sqrt{w})$ preprocessing time.

Remarks. We can further decrease the space overhead by replacing I_β and J_β, where $\beta \in \{1, 2\}$, with a bit vector F, in which $F[i] = 0$ if $A[i]$ is stored in B_1 and $F[i] = 1$ otherwise. Then, the elements in a query range $[a, b]$ are in $B_1[\texttt{rank}_0(a - 1) + 1, \texttt{rank}_0(b)]$ and $B_2[\texttt{rank}_1(a - 1) + 1, \texttt{rank}_1(b)]$. This decreases the space cost to $n + o(n) + \mathcal{O}(s^2/w)$ words. We will study both the original approach and our variant experimentally.

2.4 $(1+\varepsilon)$-Approximation in $O(\frac{n}{\varepsilon})$ Words and $\mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ Time

To design approximate solutions, Bose et al. [6] first presented a simple approach: For each $i \in \{1, 2, \ldots, n\}$, build a table T_i in which $T_i[r]$ stores the smallest index $j \geq i$ such that $A[j]$ occurs $\lceil (1+\varepsilon)^r \rceil$ times in $A[i, j]$. Given a query range $[a, b]$, they perform a binary search in T_a to find the entry $T_a[k]$ with $T_a[k] \leq b < T_a[k+1]$, and $A[T_a[k]]$ is a $(1+\varepsilon)$-approximate answer. This algorithm uses $\mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ time, and the space cost is $\mathcal{O}(\frac{n \lg n}{\varepsilon})$ words.

In a more advanced solution, Bose et al. define two number series, f_{low} and f_{high} by the recurrence $f_{low_1} = f_{high_1} = 1$, $f_{low_{r+1}} = f_{high_r} + 1$ and $f_{high_{r+1}} = \lfloor (1+\varepsilon) f_{low_r} \rfloor + 1$. They then construct a table T_i for each $i = 1, 2, \ldots, n$ as follows. In T_1, an entry $T_1[r]$ stores the smallest index $j \geq i$ such that $A[j]$ occurs f_{high_r} times in $A[1, j]$. To compute an entry $T_i[r]$ for any $i \geq 2$, we first determine whether $T_{i-1}[r]$ occurs at least f_{low_r} times in $A[i, T_{i-1}[r]]$. If it does, then we set $T_i[r] = T_{i-1}[r]$. Otherwise, $T_i[r]$ stores the smallest index $j \geq i$ such that $A[j]$ occurs f_{high_r} times in $A[i, j]$. To answer a query, observe that, the frequency of the mode of any query range $[a, b]$ with $T_a[r] \leq b < T_a[r+1]$ is at most $f_{high_{r+1}} - 1$. Since $A[T_a[r]]$ occurs at least f_{low_r} times in $A[a, T_a[r]] \subseteq A[a, b]$, the ratio of its frequency in $A[a, b]$ to $F_{a,b}$ is at least $f_{low_r}/(f_{high_{r+1}} - 1) = f_{low_r}/\lfloor (1+\varepsilon) f_{low_r} \rfloor \leq 1/(1+\varepsilon)$. [1] Therefore, $A[T_a[r]]$ is a $(1+\varepsilon)$-approximate answer.

Each table has at most $2\lceil \lg_{1+\varepsilon} n \rceil$ entries. To reduce storage costs, Bose et al. view $T_1, T_2, \ldots, T_n$ as n different versions of the same table T, and, to obtain T_i from T_{i-1}, an update is needed for each r with $T_i[r] \neq T_{i-1}[r]$. They proved that the total number of updates over all versions is $\mathcal{O}(n/\varepsilon)$, so these tables can be stored in a persistent binary search tree [11] in $\mathcal{O}(n/\varepsilon)$ words while supporting the search in any table in $\mathcal{O}(\lg(2\lceil \lg_{1+\varepsilon} n \rceil)) = \mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ time. They also maintain frequency counters [10] to achieve $O(\frac{n \lg n}{\varepsilon})$ preprocessing time.

2.5 $(1+\varepsilon)$-Approximation in $\mathcal{O}(\frac{n}{\varepsilon})$ Words and $\mathcal{O}(\lg\frac{1}{\varepsilon})$ Time

Let $\varepsilon' = \sqrt{(1+\varepsilon)} - 1$. The structures of Greve et al. [18] consist of the following two parts.

Low Frequency. For each $i = 1, 2, \ldots, n$, we precompute a table Q_i of length $\lceil \frac{1}{\varepsilon'} \rceil$, in which $Q_i[r]$ stores the rightmost index j such that $F_{i,j} = r$. Given a query range $[a, b]$, we perform a binary search to look for the index, s, of the successor of b in Q_a. If s does not exist, then $F_{a,b} > \lceil \frac{1}{\varepsilon'} \rceil$, and we use the structures for high frequencies to compute an answer. Otherwise, $F_{i,j} = s$, and, as observed by El-Zein et al. [12], $A[Q_a[s-1]+1]$ is the answer. [2]

High Frequency. For each $i = 1, 2, \ldots, n$, we precompute a table T_i of length at most $\lceil \lg_{1+\varepsilon'}(\varepsilon' n) \rceil$: For each $r \in [1, \lceil \lg_{1+\varepsilon'}(\varepsilon' n) \rceil$, if $i > 1$ and $F_{i,T_{i-1}[r]} \geq \lceil \frac{1}{\varepsilon'}(1+\varepsilon')^k \rceil + 1$, we set $T_i[r] = T_{i-1}[r]$. Otherwise, $T_i[r]$ stores the rightmost index j with $F_{i,j} \leqslant \lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{k+1} \rceil - 1$. We also build a table L_i for each i; $L_i[r]$ stores $A[i+j-1]$ where j is the smallest positive integer such that $T_{i+j}[r] \neq T_i[r]$. Then, $L_i[r]$ occurs at least $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^k \rceil + 1$ times in $A[i, T_i[r]]$. With these tables, given a query range $[a, b]$ with $F_{a,b} > \lceil \frac{1}{\varepsilon'} \rceil$, the query algorithm finds the successor, $T_a[s]$, of b in T_a. Then $F_{a,b} \leq F_{a,T_a[s]} \leqslant \lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{s+1} \rceil - 1$ and the frequency of $L_a[s-1]$ in $A[a, b]$ is at least $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{s-1} \rceil + 1$, so $L_a[s-1]$ is a $(1+\varepsilon)$-approximate mode.

---

[1] Bose et al. [6] originally defined $f_{high_{r+1}} = \lceil (1+\varepsilon) f_{low_r} \rceil + 1$. However, with their definition, the ratio of the frequency of $A[T_a[r]]$ in $A[a, b]$ to $F_{a,b}$ is at least $f_{low_r}/\lceil (1+\varepsilon) f_{low_r} \rceil$ which is not guaranteed to be at least $1/(1+\varepsilon)$. Therefore, we fix this issue by defining $f_{high_{r+1}} = \lfloor (1+\varepsilon) f_{low_r} \rfloor + 1$ instead.

[2] To return the mode, Greve et al. augments the low frequency structure by storing the mode in $A[i, Q_i[k]]$ with each $Q_i[k]$. This approach does not break asymptotic bounds, but, when implementing this data structure, we do not store these mode elements and use the observation in [12] to save space.

Hence, the total query time is $\mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$. To speed it up to $\mathcal{O}(\lg\frac{1}{\varepsilon})$, Greve et al. design a 3-approximate structure to narrow down the initial range of binary search over high frequency structures. This structure performs constant-time lowest common ancestor (LCA) queries over a tree of a small $O(\lg\lg n)$ height. Unfortunately, experiments [5] show that, for trees with small heights, structures with constant LCA queries in theory are outperformed by naive approaches. Hence, we implement their solution without this speedup.

Regarding space, the bottleneck is the high frequency structures. Greve et al. view the T_i tables as n version of the same table T as in [6] and bound the total number of updates to T by $\mathcal{O}(\frac{n}{\varepsilon})$. A similar argument apples to L_i's. It is possible to store T_i's and L_i's in a persistent search tree, but this does not allow the speedup. Instead, Greve et al. design an $\mathcal{O}(n/\varepsilon)$-word scheme which samples some table entries and encodes updates between them compactly. It supports the retrieval of an arbitrary entry in constant time. Here we sketch the scheme of storing T_i's; the entries of L_i's can be paired with those of T_i's and stored as additional fields in the same structures. In this scheme, we explicitly store T_l in an array S_l if $l \bmod t = 1$, i.e., we sample and store one out of every t versions of T. Let r be an arbitrary integer in $[1, \lceil\lg_{1+\varepsilon'}(\varepsilon' n)\rceil$. Between two consecutive sampled versions, $T_l[r]$ and $T_{l+t}[r]$, of $T[r]$, there may be updates to $T[r]$. If $r \geq 1 + \lceil\log_{1+\varepsilon'} t\rceil$, then there can only be at most one update to $T[r]$ between versions l and $l + t$. In this case, we store with each sampled entry $T_l[r]$ the next update to $T[r]$. If $r \leq \lceil\log_{1+\varepsilon'} t\rceil$, then, for each sampled entry $T_l[r]$, construct a bit vector of length t with constant-time support for `rank` which uses one bit for each of the next t versions to encode whether an update to $T[r]$ is performed. We also store the (distinct) values used to update $T[r]$ in an array.

Preprocessing. As Greve et al. did not provide information on preprocessing, we also also design an algorithm to construct their data structure in $\mathcal{O}((n\lg n)/\varepsilon)$ time.

The low frequency structure can be constructed in $\mathcal{O}(n/\varepsilon)$ time using frequency counters [10] as was done by Bose et al. [6] to compute similar tables. For the high frequency structure, if we have already computed the content of T_i's and L_i's, we can encode them in time linear in the total number of entries in T_i's and L_i's, and there are $\mathcal{O}(n\lceil\lg_{1+\varepsilon'}(\varepsilon' n)\rceil) = \mathcal{O}((n\lg n)/\varepsilon)$ entries.

What remains is to compute the entries of T_i's and L_i's, and for this we scan A $\lceil\lg_{1+\varepsilon'}(\varepsilon' n)\rceil$ times. In the r-th scan, we compute $T_i[r]$ and $L_i[r]$ for all $i \in [1, n]$ in increasing order of i as follows. We maintain an array $C[1..\Delta]$ of counters; initially all entries of C are $0s$. We use an integer m to keep track of the number of entries of C that are greater than or equal to $\lceil\frac{1}{\varepsilon'}(1+\varepsilon')^k\rceil + 1$; m can be updated each time an entry of C is updated. During the scan, we maintain the following invariant: immediately after computing $T_i[r]$, each counter $C[j]$ stores the number of occurrences of j in $A[i, T_i[r]]$. To compute $T_1[r]$, we retrieve $A[k]$ for $k = 1, 2, ...$, and for each k, we increment $C[A[k]]$. We repeat until $C[A[k]]$ is the first counter in C that reaches $\lceil\frac{1}{\varepsilon'}(1+\varepsilon')^{k+1}\rceil$. This means $A[1..k-1]$ is the longest prefix of A whose mode has frequency $\lceil\frac{1}{\varepsilon'}(1+\varepsilon')^{k+1}\rceil - 1$ in it. Therefore, we set $T_1[r] = k - 1$. Then we put the entry $A[k]$ back to the unscanned portion of A by decrementing $C[A[k]]$ and then k. To compute $T_i[r]$ for any $i > 1$, we first decrement $C[A[i-1]]$ and then check whether m is still greater than 0. If it is, then there is at least one element whose frequency in $A[i, T_{i-1}[r]]$ is $\lceil\frac{1}{\varepsilon'}(1+\varepsilon')^k\rceil + 1$, and we set $T_i[r] = T_{i-1}[r]$. Otherwise, we resume the scanning of A to compute $T_i[r]$ using the approach used to compute $T_1[r]$. We also store $A[i-1]$ in $L_r[u], L_r[u+1], \ldots, L_r[r-1]$, where u is the smallest integer such that $T_u[r] = T_{r-1}[r]$. With this implementation, we need to scan the input array A $\mathcal{O}(\lceil\lg_{1+\varepsilon'}(\varepsilon' n)\rceil)$ times, and hence the total preprocessing time is $\mathcal{O}(n\lceil\lg_{1+\varepsilon'}(\varepsilon' n)\rceil) = \mathcal{O}((n\lg n)/\varepsilon)$.

2.6 $(1+\varepsilon)$-Approximation in $\mathcal{O}(\frac{n}{\varepsilon})$ Bits and $\mathcal{O}(\lg \frac{1}{\varepsilon})$ Time

The encoding data structure of El-Zein et al. [12] also consists of two parts: The low frequency structure contains, for each integer $k \in [1, \lceil \frac{1}{\varepsilon} \rceil]$, a table Q_k of length n, in which $Q_k[i]$ stores the rightmost index j such that $F_{i,j} = k$. Q_k can be encoded by Lemma 1 in $2n + o(n)$ bits, so all tables use $\mathcal{O}(\frac{n}{\varepsilon})$ bits. Then, for a range $[a, b]$, we perform a binary search in $Q_1[a], Q_2[a], \ldots, Q_{\lceil \frac{1}{\varepsilon} \rceil}[a]$ to check whether $F_{a,b} \le \lceil \frac{1}{\varepsilon} \rceil$ and compute the index of a mode if so.

The high frequency structure contains, for each integer $k \in [1, \lfloor \log_{1+\varepsilon}(\varepsilon n) \rfloor]$, a data structure that can find in $\mathcal{O}(1)$ time one of these inequalities that holds for query range $[a, b]$: 1) $F_{a,b} < (1+\varepsilon)^k/\varepsilon$, 2) $F_{a,b} > (1+\varepsilon)^k/\varepsilon$, or 3) $(1+\varepsilon)^{k-1/2}/\varepsilon < F_{a,b} < (1+\varepsilon)^{k+1/2}/\varepsilon$. It finds in case 2 an element that occurs more than $(1+\varepsilon)^k/\varepsilon$ times in $A[a, b]$, and, in case 3, an element that occurs more than $(1+\varepsilon)^{k-1/2}/\varepsilon$ times in $A[a, b]$.

Let $\varepsilon' = \sqrt{1+\varepsilon} - 1$ and $f_j = (\varepsilon'/\varepsilon) \times (1+\varepsilon')^j$. This structure is designed based on four sequences s, s', r and r': For each integer $i \in [0, n/\lceil f_{2k-1} \rceil]$, s_i, the i-the element in s, is $i\lceil f_{2k-1} \rceil + 1$, and r_i is the smallest index such that $F_{s_i, r_i} \geqslant (1+\varepsilon')^{2k}/\varepsilon$. similarly, for each integer $i \in [0, n/\lceil f_{2k} \rceil]$, define $s'_j = i\lceil f_{2k} \rceil + 1$, and r'_j is the smallest index such that $F_{s'_j, r'_j} \geqslant (1+\varepsilon')^{2k+1}/\varepsilon$. Then, given a query range $[a, b]$, El-Zein et al. determine which case applies by comparing b to the entries of r and r' that correspond to the predecessors of a in s and s'. The high frequency structure can be encoded in $\mathcal{O}(\frac{n}{\varepsilon})$ bits by Lemma 1.

To use this trichotomy to answer queries in the high frequency case, perform a binary search in $\mathcal{O}(\lg \lg n + \lg \frac{1}{\varepsilon})$ time to compute a k such that either case 3 applies for the query range, or case 2 applies for k and case 1 applies for $k+1$. The element found by either case 3 or case 2 is a $(1+\varepsilon)$-approximate mode. Finally, to speed up the query time to $\mathcal{O}(\lg \frac{1}{\varepsilon})$, El-Zein et al. designed an $\mathcal{O}(n)$-bit structure that answers 4-approximate range mode queries in constant time, and used it to narrow down the initial range of binary search.

Remarks. The $\mathcal{O}(n)$-bit 4-approximate structure contains a network of fusion trees [16] and is not practical. Hence, our implementation does not include this speedup. El-Zein et al. did not discuss preprocessing, but we can build their structures using frequency counters [10] in $\mathcal{O}(n \lg n/\varepsilon)$ time. Finally, storing all structures in integer arrays without using Lemma 1 would yield a simple $\mathcal{O}(n/\varepsilon)$-word solution, which we also conduct experimental studies on.

3 Experimental Results

3.1 Experimental Setup

Table 1 gives an outline of the data structures we implemented. Among them, the first naive approach, $\texttt{nv}_1$, sorts the elements in the given range to answer a query, while the second one, $\texttt{nv}_2$, scans the elements in the range and uses an array of length Δ to count element frequencies. Four data structures, $\texttt{subsr}_1$, $\texttt{subsr}_2$, `sample` and `succ`, use succinct bit vectors, for which we use the implementation in the succinct data structures library, `sdsl-lite`, of Gog et al. [17]. Two types of bit vectors are used: a plain bit vector, `sdsl::bit_vector` and a compressed bit vector [30], `sdsl::rrr_vector`. To distinguish them, we combine $\texttt{subsr}_1$, $\texttt{subsr}_2$, `sample` or `succ` with superscripts `p` or `c`, e.g., $\texttt{succ}^{\texttt{p}}$ and $\texttt{succ}^{\texttt{c}}$, to respectively indicate whether plain or compressed bit vectors are used. Note that, even though $\texttt{subsr}_2^{\texttt{c}}$ uses compressed bit vectors to encode the table S', a plain bit vector is still used to represent F: we found that, due to the small space cost of F (n bits), compressing it would achieve negligible space savings at the cost of increasing query times by 4.5% to 25%. Finally, for a fair comparison, we modified the implementation of persistent search trees by Jansens [21] to remove the space overhead for generic programming and used it to implement `pst`.

Table 1 The data structures we implemented. The first half of the table present exact solutions, while the second half are $(1+\varepsilon)$-approximate structures with $\mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ query time.

abbr.	description
`nv`$_1$,`nv`$_2$	two naive solutions in Section 3.1
`supsr`	$\mathcal{O}(n)$-word, $\mathcal{O}(\sqrt{n}\lg n)$ query time structure for exact range mode in Section 2.1
`sqrt`	$\mathcal{O}(n)$-word, $\mathcal{O}(\sqrt{n})$ query time structure for exact range mode in Section 2.2
`subsr`$_1$	$\mathcal{O}(n)$-word, $\mathcal{O}(\sqrt{n/w})$ query time structure for exact range mode in Section 2.3
`subsr`$_2$	modifying `subsr`$_1$ with more succinct data structures; see the remarks in Section 2.3
`simple`	simple $\mathcal{O}(\frac{n\lg n}{\varepsilon})$-word approximate solution in Section 2.4
`pst`	$\mathcal{O}(\frac{n}{\varepsilon})$-word approximate solution with persistent search trees in Section 2.4
`sample`	$\mathcal{O}(\frac{n}{\varepsilon})$-word approximate solution with sampling in Section 2.5
`tri`	$\mathcal{O}(\frac{n}{\varepsilon})$-word approximate solution with the trichotomy in Section 2.6
`succ`	$\mathcal{O}(\frac{n}{\varepsilon})$-bit approximate solution with the trichotomy in Section 2.6

Table 2 The data sets used in our experiments, each stored as an array of n integers in $[1,\Delta]$.

data	n	Δ	$\lg\Delta$	H_0	
`reviews`	10,000,000	1,367,909	20.38	18.46	books of the first 10^8 book reviews by Amazon customers in 2018 [25]
`IPs`	8,571,089	135,542	17.04	7.96	source IP addresses of DDoS attacks [14]
`words`	6,715,122	127,886	16.96	12.74	words in a text string containing the 100 most frequently downloaded Project Gutenberg [1] e-books in July 2021, with stop words removed
`library`	10,000,000	314,358	18.26	15.75	first 10^8 call numbers in the 2016/17 Seattle Public Library checkout records [23]
`tickets`	10,000,000	79,027	16.27	11.10	street names of the first 10^8 parking tickets issued in New York in 2017 [26]

Five publicly available datasets are used; see Table 2. This table also shows the zeroth-order empirical entropy, H_0, of each dataset. Due to page limit, sometimes we only show figures and tables created for typical datasets, and a full set of tables/figures for all datasets is available in the second author's thesis [24]. To convert raw data into an integer array, we encode each element as an integer in $[1,\Delta]$. To generate a query range $[a,b]$, we adopt the method in [8, 20]: we pick an integer from $[1,n]$ uniformly at random (u.a.r.) and assign it to a, and b is chosen u.a.r. from $[a, a+\lceil\frac{n-a}{K}\rceil]$ for a parameter K. We generate three categories of queries, `large`, `medium` and `small`, by setting $K = 1$, 10 and 100, respectively. To justify that this approach of generating queries is appropriate, Appendix C shows additional studies, including those performed over query ranges even smaller than `small` queries.

Our platform is a server with an Intel(R) Xeon(R) Gold 6234 CPU and 128GB of RAM, running Ubuntu 18.04.2. We complied programs using `g++` 7.4.0 with `-O2` flags.

3.2 An Initial Performance Study on Exact Mode

For exact range mode, we initially set $s = \sqrt{n}$ for `supsr` and `sqrt` and set $s = \sqrt{nw}$ for `subsr`$_1$ and `subsr`$_2$ to achieve linear space as in [22, 7]. Tables 3 and 4 present the query time, space usage and construction time of exact query structures. We measure space costs

Table 3 Average time to answer an exact range mode query, measured in micro seconds. Queries are categorized into small, medium and large, and each category has 10^6 queries.

	Query	nv$_1$	nv$_2$	supsr	sqrt	subsr$_1^p$	subsr$_1^c$	subsr$_2^p$	subsr$_2^c$
reviews	small	1134	442	338.93	51.70	10.74	11.56	10.74	11.56
	medium	13262	870	366.90	51.00	9.94	10.82	9.94	10.82
	large	144642	5686	363.42	50.85	9.39	10.20	9.39	10.20
IPs	small	532	51	218.75	15.58	3.93	4.40	3.94	4.48
	medium	5938	186	240.03	15.19	3.86	4.37	3.91	4.46
	large	66121	1531	239.35	14.48	3.53	4.01	3.60	4.07
words	small	678	45	298.83	31.49	8.01	8.75	8.14	9.06
	medium	7094	149	334.53	31.27	7.60	8.41	7.82	8.63
	large	73401	1235	349.22	28.28	6.53	7.24	6.67	7.38
library	small	1160	125	384.07	49.54	11.90	13.14	12.13	13.37
	medium	12960	408	422.32	47.11	10.66	11.87	10.71	11.98
	large	132605	3407	444.30	43.68	9.32	10.42	9.40	10.53
tickets	small	990	37	362.47	43.16	9.99	10.67	10.19	10.99
	medium	9931	187	414.76	42.39	9.92	10.65	10.15	10.97
	large	101281	1756	436.93	37.44	8.56	9.35	8.80	9.60

Table 4 Space (bits per symbol) and construction time (minutes) of exact range mode structures.

	Dataset	supsr	sqrt	subsr$_1^p$	subsr$_1^c$	subsr$_2^p$	subsr$_2^c$
space	reviews	109.1	173.2	174.3	144.1	174.3	144.1
	IPs	97.5	161.5	332.6	255.9	205.8	129.0
	words	97.8	161.9	329.1	284.1	202.2	157.2
	library	99.0	163.0	315.2	294.5	188.3	167.6
	tickets	96.7	160.8	311.0	289.9	184.1	163.0
construct time	reviews	0.911	0.911	7.205	7.460	7.205	7.460
	IPs	0.695	0.695	1.865	1.867	1.890	1.892
	words	0.438	0.438	2.755	2.760	2.762	2.765
	library	0.806	0.806	5.923	5.933	5.971	5.974
	tickets	0.720	0.720	4.251	4.275	4.756	4.809

in bits per symbol (bps), which is the space usage in bits divided by the length of the input array. Furthermore, the cost of the input array A (32 bps) is included in the space usage of supsr and sqrt but excluded for subsr$_1$ and subsr$_2$, because supsr and sqrt scan A when answering a query but subsr$_1$ and subsr$_2$ do not. Nevertheless, the space cost of A is not significant enough to affect our conclusions. These tables show that most data structures have much faster query time than both naive approaches, and supsr is the only exception in some cases. Between two naive approaches, nv$_2$ is faster because the number of distinct elements is relatively small compared to input array length.

Before comparing the performance of data structure solutions, we discuss how the distributions of the datasets affect subsr$_1$ and subsr$_2$, for which the array entries are stored in two subsequences B_1 and B_2 (see Section 2.3). Since B_2 stores elements of higher frequency, the lower the entropy of a dataset is, the larger the ratio of the length of B_2 to n tends to be. Indeed, for reviews, words and library, the ratios are 0, 0.037 and 0.010, respectively, while for IPs and tickets, the ratios are 0.58 and 0.14, respectively, which are higher. These

ratios are consistent with the values of H_0 in Table 2. This immediately explains why, for reviews, there is no difference in costs between $\texttt{subsr}_1$ and $\texttt{subsr}_2$: These two solutions differ in the components used to map the query range to ranges in B_1 and B_2. Since $|B_2| = 0$ for reviews, no mapping is needed.

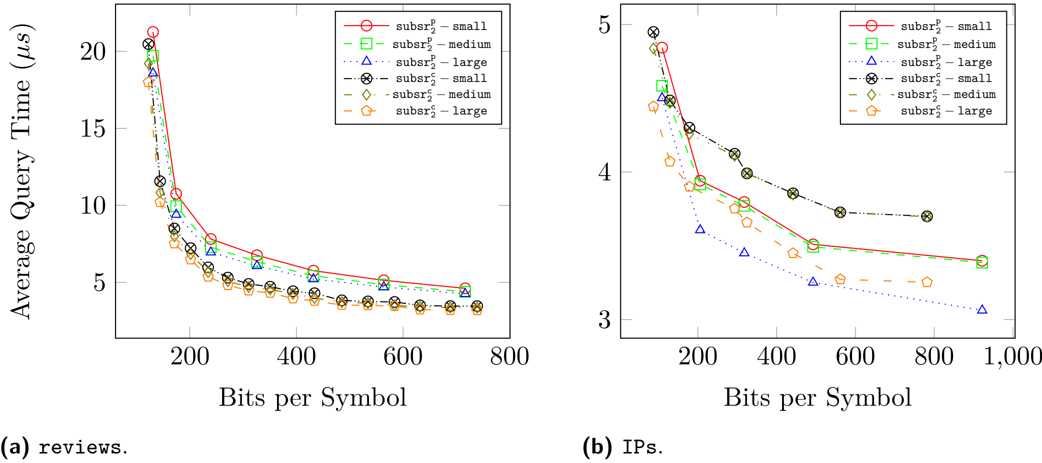
With this in mind, we now compare data structure solutions. We first find that the query time of sqrt is 6.0% to 15.3% of that of supsr, which is consistent with theoretical bounds. Then we observe that, by using a succinct bit vector to replace multiple arrays, $\texttt{subsr}_2$ saves much space compared to $\texttt{subsr}_1$ over all datasets except reviews (which does not require these components as discussed before). At the same time, there is almost no sacrifice in query performance. This is because we only perform rank over this bit vector a constant number of times to map query ranges to ranges in B_1 and B_2, and this cost is dominated by subsequent steps which use $O(\sqrt{n/w})$ time. The use of compressed bit vectors in $\texttt{subsr}_1^\texttt{c}$ and $\texttt{subsr}_2^\texttt{c}$ saves more space, albeit at the cost of a small increase in query time. Theoretical analysis indicates that, when we double s, the query time halves but tables S and S' use four times as much space. Hence, we predict that $\texttt{subsr}_2^\texttt{p}$ and $\texttt{subsr}_2^\texttt{c}$ achieve the best query-space tradeoffs, and more experiments will be run in Section 3.3 to confirm this.

The sizes of query ranges affect query times greatly for the naive approaches since they either sort or scan the elements in the range. On the other hand, these sizes only affect the query times of supsr, sqrt, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ slightly. For sqrt, $\texttt{subsr}_1$ and $\texttt{subsr}_2$, larger queries even tend to take less time to answer. This is because the query algorithm of sqrt (which is also performed over B_2 in $\texttt{subsr}_1$ and $\texttt{subsr}_2$) keeps updating a candidate by a new candidate with higher frequency in the query range, until the mode of the range is found. The initial candidate is the mode of the span of the query. When the query range is larger, the span is also longer, and hence its mode tends to be a better candidate, thus decreasing the query time.

Regarding construction time, observe that the processing times of supsr and sqrt are about same. For reviews, words, library and tickets, the preprocessing time of supsr and sqrt is 12.2% to 16.9% of that of $\texttt{subsr}_1$ and $\texttt{subsr}_2$. This is because, with the choices of parameters, it takes $\mathcal{O}(n^{3/2}\sqrt{w})$ time to build $\texttt{subsr}_1$ and $\texttt{subsr}_2$, but the preprocessing time of supsr and sqrt is $\mathcal{O}(n^{3/2})$. However, the difference is much smaller for IPs. This is because, when constructing $\texttt{subsr}_1$ and $\texttt{subsr}_2$ for this dataset, 58% of array entries are in B_2, whose query structure can be built in linear time.

3.3 Different Parameter Values

We now choose different values of s to compare these structures thoroughly. First, we compare $\texttt{subsr}_2^\texttt{p}$ and $\texttt{subsr}_2^\texttt{c}$. The experimental results over reviews and IPs are shown in Figure 1, while the results over words, library and tickets are shown in Figure 4 in Appendix A. To draw the subfigure for either dataset, we initially set s to be $0.5\sqrt{nw}$ to construct $\texttt{subsr}_2^\texttt{p}$ or $\texttt{subsr}_2^\texttt{c}$, and each time we increase s by $0.5\sqrt{nw}$ until the space usage exceeds 640 bps. Each point in the figure represents a tradeoff achieved between space and the average query time of a category (small, medium or large) of queries. We then connect the points for the same data structure and query category into a polyline. Hence, over either dataset, we show how the query time changes when more space is used for either data structure using three plotted polylines, one for each query category. In Figure 1 (a), for the same category of queries, the polyline plotted for $\texttt{subsr}_2^\texttt{p}$ is always above that for $\texttt{subsr}_2^\texttt{c}$. This means, with the same space cost, $\texttt{subsr}_2^\texttt{c}$ uses less time to answer a query on average. Hence, $\texttt{subsr}_2^\texttt{c}$ outperforms $\texttt{subsr}_2^\texttt{p}$ over reviews. It is however the opposite for IPs, and there is no discernible differences over the three other datasets.



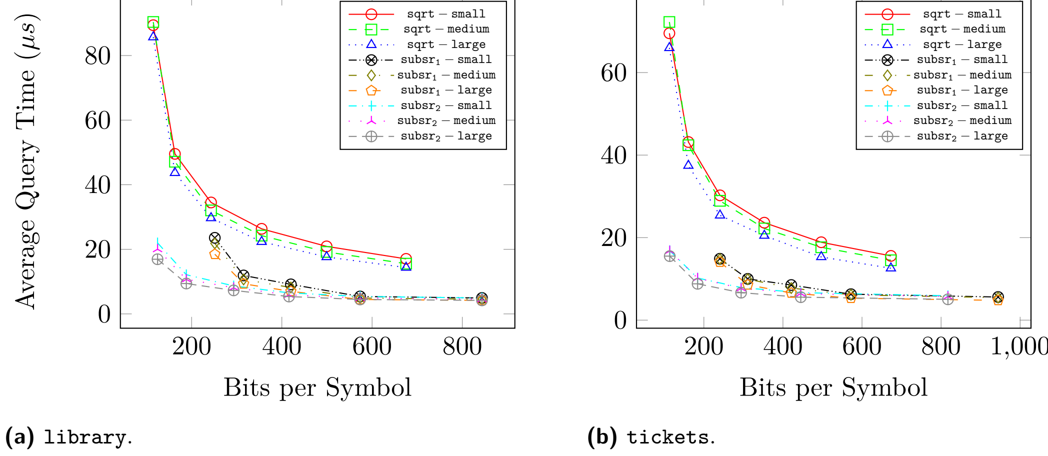
(a) `reviews`. **(b)** `IPs`.

Figure 1 Time-space tradeoffs of $\texttt{subsr}_2^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{c}}$ over `reviews` and `IPs`.

To discuss why they compare differently for different datasets, observe that whether to use plain or compressed bit vectors to encode S' in $\texttt{subsr}_2$ affects the structures built over B_1 only. Furthermore, when s increases, the block size decreases, and more adjacent entries of S' tend to store the same values, making S' more compressible. The dataset `reviews` has the largest entropy, which means the table S' constructed over it is less compressible than that over any other dataset for small s, so the increase of s makes it more compressible rapidly. All arrays entries of `reviews` are also stored in B_1 for all the values of s that we have used, making the compression more sensitive to the choice of the value of s. Hence, for `reviews`, the increase of s improves the compression ratio of $\texttt{subsr}_2^{\texttt{c}}$ at a faster rate than what it does for any other dataset. This allows S' to store much more precomputed information for $\texttt{subsr}_2^{\texttt{c}}$, speeding up the queries despite the increased operation time over compressed bit vectors. Other datasets perform differently when s changes, due to their smaller entropy which also affects the number of array entries distributed into B_1. In the extreme case of `IPs`, $\texttt{subsr}_2^{\texttt{p}}$ performs better, while for the rest, compression does not make a significant or consistent difference. Since it takes less time to construct plain bit vectors, we decide that $\texttt{subsr}_2^{\texttt{p}}$ is also a better solution for `words`, `library` and `tickets`.

We further conducted similar experiments to compare $\texttt{subsr}_1^{\texttt{p}}$ and $\texttt{subsr}_1^{\texttt{c}}$ and made the same observations. See Figure 5 in Appendix A for details. Hence, in the rest of this paper, when the context is clear, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ respectively represent $\texttt{subsr}_1^{\texttt{c}}$ and $\texttt{subsr}_2^{\texttt{c}}$ for `reviews`, while they represent $\texttt{subsr}_1^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{p}}$ for all other datasets.

After deciding on bit vector implementations, we compare `sqrt`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$. We continue with the same parameters for $\texttt{subsr}_1$ and $\texttt{subsr}_2$, while for `sqrt`, the initial value of s is $0.5\sqrt{n}$, and each time we increase s by $0.5\sqrt{n}$ until the space usage exceeds 640 bps. Figure 2 shows the results over `library` and `tickets`, and the results are similar for the other three data sets (see Figure 6 in Appendix A), except that for `reviews`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ have the same performance because no structures are used to map query ranges as discussed before. Our results show that $\texttt{subsr}_1$ and $\texttt{subsr}_2$ have much better query performance than `sqrt` when the same storage costs are incurred. This matches the discussions and prediction in Section 3.2. Between $\texttt{subsr}_1$ and $\texttt{subsr}_2$, for `IPs`, `words`, `library` and `tickets`, our results show that $\texttt{subsr}_2$ achieves better time-space tradeoffs than $\texttt{subsr}_1$ does. The difference is significant for smaller values of s, but as s grows much larger, the plotted lines start to converge. This is because, for large enough s, the space savings by replacing four integer



(a) `library`. **(b)** `tickets`.

Figure 2 Time-space tradeoffs of `sqrt`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ over `library` and `tickets`.

arrays with a bit vector is dominated by the $\mathcal{O}(s^2)$-bit cost of S'. Nevertheless, when we require a reasonable space cost for data structures in practice, $\texttt{subsr}_2$ still improves $\texttt{subsr}_1$ significantly. Finally, we conducted similar experiments to confirm that `sqrt` outperforms `supsr` significantly; see Appendix A. Therefore, we conclude that $\texttt{subsr}_2$ preforms the best among all exact solutions.

3.4 Performance of Approximate Range Mode

Tables 5 and 6 present the query time, space usage and construction time of approximate range mode structures when $\varepsilon = 1/2$. Space costs do not include the cost of array A, since these structures can compute the indexes of approximate range modes without accessing A.

To measure accuracy, we compute the approximation ratio of each answer as the frequency of the actual mode in the query range divided by the frequency of the reported approximate mode in the range. Then, for each solution, we compute the average and the maximum of the approximation ratios of the answers for each query category over each dataset. We find that the average ratios range between 1.00001 and 1.02630, and the maximum ratios are closer to 1.5. To see why the average quality of the answers is high, recall that the approximate mode computed is the actual mode of a range having a significant overlap with the query range, so the probability of it being the mode of the query range is high. Since these results are consistent across datasets and query categories, we use Table 7 in Section 3.5 to provide a summary by reporting, for each data structure, the average and maximum ratios over all queries, together with results for some subsequent experiments. Since these structures have slower query support and higher space usage for smaller ε, setting $\varepsilon = 1/2$ is attractive to applications for which a high average approximation ratio is sufficient.

Another phenomenon is that larger queries tend to be faster with approximate solutions. This is because all query algorithms are essentially based on binary searches in lists of possible candidates, and in each list, the farther it is away from the list head, the larger the gaps between the indexes (in A) of two consecutive candidates are, benefiting larger query ranges.

We also observe that the space cost of `pst` can vary greatly among datasets, with the space cost of `library` being about 3.6% of that of `IPs`. Recall that in this solution, we view n different tables as versions of the same table T to store them in a persistent search tree, and each tree node corresponds to an update to the table (the initial version of the table is not

Table 5 Average time to answer an approximate query for $\varepsilon = 1/2$, measured in microseconds. Queries are categorized into small, medium and large, and each category has 10^8 queries.

	Query	simple	pst	sample^p	sample^c	tri	succ^p	succ^c
reviews	small	0.098	0.861	0.869	1.016	0.191	1.122	2.970
	medium	0.095	0.714	0.598	0.610	0.135	1.009	3.178
	large	0.089	0.556	0.440	0.453	0.116	0.864	3.703
IPs	small	0.110	1.561	0.545	0.796	0.138	1.003	4.003
	medium	0.113	1.343	0.358	0.430	0.105	0.696	3.198
	large	0.120	1.120	0.285	0.304	0.091	0.581	3.030
words	small	0.102	0.986	0.809	1.166	0.168	1.126	3.642
	medium	0.098	0.780	0.486	0.585	0.127	0.967	3.754
	large	0.105	0.546	0.281	0.309	0.095	0.595	2.547
library	small	0.099	0.760	1.017	1.164	0.200	1.230	3.508
	medium	0.099	0.581	0.603	0.629	0.144	1.152	3.809
	large	0.106	0.434	0.360	0.370	0.112	0.766	3.023
tickets	small	0.112	1.072	0.773	1.108	0.172	1.281	3.861
	medium	0.109	0.817	0.460	0.585	0.129	0.997	3.371
	large	0.119	0.580	0.300	0.327	0.105	0.634	2.669

Table 6 Space (bits per symbol) and construction time (minutes) of approximate structures when $\varepsilon = 1/2$.

	Dataset	simple	pst	sample^p	sample^c	tri	succ^p	succ^c
space	reviews	680.0	100.6	225.4	204.7	291.2	56.9	11.4
	IPs	1038.6	1051.5	327.9	311.3	291.5	82.9	30.0
	words	787.8	146.3	240.7	220.5	291.4	67.1	21.5
	library	769.6	37.6	231.6	210.8	291.3	65.6	13.9
	tickets	896.6	115.8	248.2	228.1	291.5	74.2	24.2
construct time	reviews	0.084	0.142	0.655	0.668	0.050	0.082	0.085
	IPs	0.075	0.172	0.564	0.568	0.031	0.063	0.067
	words	0.050	0.082	0.412	0.418	0.018	0.038	0.040
	library	0.084	0.136	0.663	0.673	0.042	0.065	0.067
	tickets	0.081	0.122	0.648	0.649	0.027	0.068	0.070

stored explicitly since $T[i] = i$ for all $i \in [1, n]$). Thus, we recorded the number of updates to T for each dataset, and it is $1,380,391$ for reviews, $10,773,911$ for IPs, $1,232,046$ for words, $485,498$ for library and $1,386,886$ for tickets. The difference in the numbers of updates is consistent with the difference in space costs. To see why there is such a difference in updates, recall that an update to T happens when the frequency of a candidate within a certain range $A[i, j]$ drops below a threshold when we increment i. This happens more often when the entropy of the dataset is lower or when the locality of reference is higher, since a lower entropy or higher locality of references means we are more likely to decrease the frequency of this candidate each time we increment i. Indeed, IPs has the lowest entropy by Table 2, and since the same subset of IPs occur frequently in a DDoS attack event, it has high locality of reference. This explains the high space cost of pst over IPs. On the other hand, library has the second highest entropy, and unlike reviews whose entropy is higher, due to the limited number of copies that a library has for each book, the borrowing records

tend to be less affected by trends such as "best sellers of the month" than Amazon reviews are. This explains the low space usage for `library`. The space cost of `sample` also fluctuates among datasets for similar reasons, but due to sampling, the difference is small.

We now compare approximate structures. Among them, `simple` has the fastest query time due to its simplicity, but its space cost is high. Among more sophisticated, $O(n/\varepsilon)$-word solutions which are not succinct, `tri` stands out as its query time is comparable to that of `simple` (it even beats `simple` in some cases), but its space cost is only 28.1% to 42.8% of that of `simple`. Compared to `pst` and `sample`, it has the smallest worst-case space cost; it is not based on persistence and is thus not sensitive to entropy or locality of reference. On the other hand, for most datasets, `pst` and `sample` provide useful tradeoffs with lower space usage but slower query time, with `pst` especially attractive for datasets of high entropy but low locality of reference such as `library`. Finally, $\texttt{succ}^{\texttt{p}}$ and $\texttt{succ}^{\texttt{c}}$ provide compact solutions; $\texttt{succ}^{\texttt{p}}$ uses $0.89n$ to $1.30n$ words, with query slightly slower than `pst` and `sample` in most cases, while $\texttt{succ}^{\texttt{c}}$ is highly compact, with space costs only 35.6% to 93.8% of the array of 32-bit integers (in most cases, the space cost is closer to the lower end, and the average is 20.2 bps over all datasets), while the query time is 265% to 522% of that of $\texttt{succ}^{\texttt{p}}$. Regarding preprocessing, we observed that the construction of `tri` is the fastest while that of `sample` is the slowest.

3.5 Different Values of ε and Comparisons to Exact Queries Structures

We further conduct experiments by setting ε to 1/4, 1/8 and 1/16. Table 7 shows that average approximation ratios decrease when ε decreases, though they are already close to 1 for $\varepsilon = 1/2$. Maximum approximation ratios are close to $1 + \varepsilon$.

Table 7 Average and max approximation ratios for different ε.

ε	Average				Maximum			
	`simple`	`pst`	`sample`	`tri/succ`	`simple`	`pst`	`sample`	`tri/succ`
1/2	1.00644	1.00464	1.00644	1.00192	1.49977	1.5	1.48879	1.47826
1/4	1.00218	1.00164	1.00188	1.00085	1.24952	1.25	1.24701	1.25
1/8	1.00075	1.00068	1.00055	1.00019	1.12474	1.125	1.12148	1.11765
1/16	1.00020	1.00017	1.00016	1.00006	1.06240	1.0625	1.06107	1.05882

We also measure the performance of each solution for different ε. Tables 8 and 9 present the performance and accuracy of approximate query structures for different values of ε over the `words` dataset. We observe that query times increase slowly as ε decreases, fitting the growth of the function of $\lg \frac{1}{\varepsilon} + \lg \lg n$. The space costs, however, grows at a much faster rate, proportional to $1/\varepsilon$. For different values of ε, how different solutions compare to each other is similar to the case where $\varepsilon = 1/2$. The main notable difference is that, due to persistence or compression, the space costs of `pst`, `sample`, and $\texttt{succ}^{\texttt{c}}$ grow more slowly than other data structures.

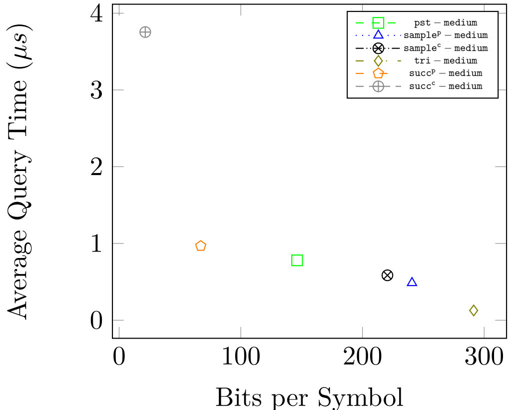
Finally, for $\varepsilon = 1/2$, we plotted figures to compare approximate structures to the best exact structure, `subsr2`. Due to its high space costs, `simple` is not included. To better compare approximate solutions, we plot a subfigure without `subsr2`, before plotting another one with `subsr2`. As a typical example, Figure 3 shows the tradeoffs achieved for medium queries over `words`, while Figure 8 in Appendix B shows the tradeoffs for all three types of queries over `reviews`. From them, we can tell approximate structures outperform exact structures greatly, making them suitable for applications that require good average approximations. They still achieve better time/space tradeoffs over `subsr2` for $\varepsilon = 1/4$, but may lose the appeals when we keep decreasing ε due to the increase in space costs.

Table 8 Average time to answer an approximate query over the `words` datasets for $\varepsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$, measured in microseconds. Queries are categorized into `small`, `medium` and `large`, and each category has 10^8 queries.

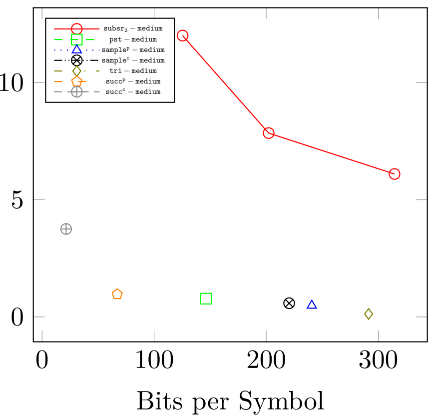
ε	Query	`simple`	`pst`	`sample`^{p}	`sample`^{c}	`tri`	`succ`^{p}	`succ`^{c}
1/2	`small`	0.102	0.986	0.809	1.166	0.168	1.126	3.642
	`medium`	0.098	0.780	0.486	0.585	0.127	0.967	3.754
	`large`	0.105	0.546	0.281	0.309	0.095	0.595	2.547
1/4	`small`	0.122	1.178	1.069	1.486	0.248	1.568	4.537
	`medium`	0.119	0.924	0.738	0.841	0.184	1.371	4.597
	`large`	0.119	0.639	0.357	0.386	0.142	0.906	3.512
1/8	`small`	0.148	1.637	1.277	1.809	0.349	2.231	5.778
	`medium`	0.138	1.586	1.253	1.280	0.269	2.128	5.940
	`large`	0.133	1.090	0.543	0.563	0.194	1.402	4.598
1/16	`small`	0.178	1.704	1.439	2.061	0.468	2.993	6.849
	`medium`	0.170	1.479	1.459	1.690	0.383	3.104	7.230
	`large`	0.161	0.935	1.025	1.077	0.261	2.095	5.894

Table 9 Space (bits per symbol) and construction time (minutes) when answering approximate queries over the `words` datasets for $\varepsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

	ε	`simple`	`pst`	`sample`^{p}	`sample`^{c}	`tri`	`succ`^{p}	`succ`^{c}
space	1/2	787.8	146.3	240.7	220.5	291.4	67.1	21.5
	1/4	1418.9	264.7	393.0	352.4	547.8	117.4	35.9
	1/8	2677.9	657.9	704.3	619.5	1063.9	212.2	62.6
	1/16	5185.2	753.6	1337.7	1157.9	2074.5	389.5	111.7
construc-tion	1/2	0.050	0.082	0.412	0.418	0.018	0.038	0.040
	1/4	0.091	0.166	0.744	0.746	0.032	0.066	0.077
	1/8	0.171	0.318	1.610	1.636	0.058	0.148	0.150
	1/16	0.335	0.554	3.224	3.247	0.108	0.229	0.238



(a) `words` – `medium` without `subsr`$_2$.



(b) `words` – `medium` with `subsr`$_2$.

Figure 3 Time-space tradeoffs of different data structures for medium queries over `words`.

References

1. Project Gutenberg. (n.d.), retrieved in July 2021. Available from `https://www.gutenberg.org/`.
2. Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 522–539. SIAM, 2021. `doi:10.1137/1.9781611976465.32`.
3. Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In Guy E. Blelloch and Dan Halperin, editors, *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010*, pages 84–97. SIAM, 2010. `doi:10.1137/1.9781611972900.9`.
4. Nikhil Bansal and Ryan Williams. Regularity lemmas and combinatorial algorithms. *Theory of Computing*, 8:69–94, 2012.
5. Michael A Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.
6. Prosenjit Bose, Evangelos Kranakis, Pat Morin, and Yihui Tang. Approximate range mode and range median queries. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 377–388. Springer, 2005.
7. Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T. Wilkinson. Linear-space data structures for range mode query in arrays. *Theory of Computing Systems*, 55(4):719–741, March 2013.
8. Francisco Claude, J Ian Munro, and Patrick K Nicholson. Range queries over untangled chains. In *International Symposium on String Processing and Information Retrieval*, pages 82–93. Springer, 2010.
9. O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Engineering the LOUDS succinct tree representation. In Carme Àlvarez and Maria J. Serna, editors, *Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings*, volume 4007 of *Lecture Notes in Computer Science*, pages 134–145. Springer, 2006. `doi:10.1007/11764298_12`.
10. Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms*, pages 348–360. Springer, 2002.
11. James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.
12. Hicham El-Zein, Meng He, J Ian Munro, Yakov Nekrich, and Bryce Sandlund. On approximate range mode and range selection. In *30th International Symposium on Algorithms and Computation (ISAAC 2019)*, volume 149, page 57. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.
13. Hicham El-Zein, Meng He, J Ian Munro, and Bryce Sandlund. Improved time and space bounds for dynamic range mode. In *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112, page 25. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.
14. Derya Erhan. Boğaziçi university DDoS dataset, 2019. Available from `https://dx.doi.org/10.21227/45m9-9p82`.
15. Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Trans. Algorithms*, 2(4):611–639, 2006. `doi:10.1145/1198513.1198521`.
16. Michael L Fredman and Dan E Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, pages 1–7, 1990.
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)*, pages 326–337, 2014. `doi:10.1007/978-3-319-07959-2_28`.

18 Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, and Jakob Truelsen. Cell probe lower bounds and approximations for range mode. In *International Colloquium on Automata, Languages, and Programming*, pages 605–616. Springer, 2010.

19 Yuzhou Gu, Adam Polak, Virginia Vassilevska Williams, and Yinzhan Xu. Faster monotone min-plus product, range mode, and single source replacement paths. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 75:1–75:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ICALP.2021.75`.

20 Meng He and Serikzhan Kazi. Path query data structures in practice. In *18th International Symposium on Experimental Algorithms*, volume 160, pages 27:1–27:16, 2020.

21 D. Jansens. *Persistent Binary Search Trees.* `https://cglab.ca/~dana/pbst/`.

22 Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nordic Journal of Computing*, 12(1):1–17, 2005.

23 Seattle Public Library. Seattle library checkout records, 2017. Available from `https://www.kaggle.com/seattle-public-library/seattle-library-checkout-records`.

24 Zhen Liu. Exact and approximate range mode query data structures in practice. Master's thesis, Dalhousie University, 2023. URL: `http://hdl.handle.net/10222/81772`.

25 Jianmo Ni, Jiacheng Li, and Julian McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 188–197, 2019.

26 City of New York. NYC parking tickets, 2017. Available from `https://www.kaggle.com/datasets/new-york-city/nyc-parking-tickets`.

27 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007. `doi:10.1137/1.9781611972870.6`.

28 Mihai Patrascu. Succincter. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 305–313, 2008.

29 Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Information Processing Letters*, 109(4):225–228, 2009.

30 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 (TALG)*, 3(4):43–es, 2007.

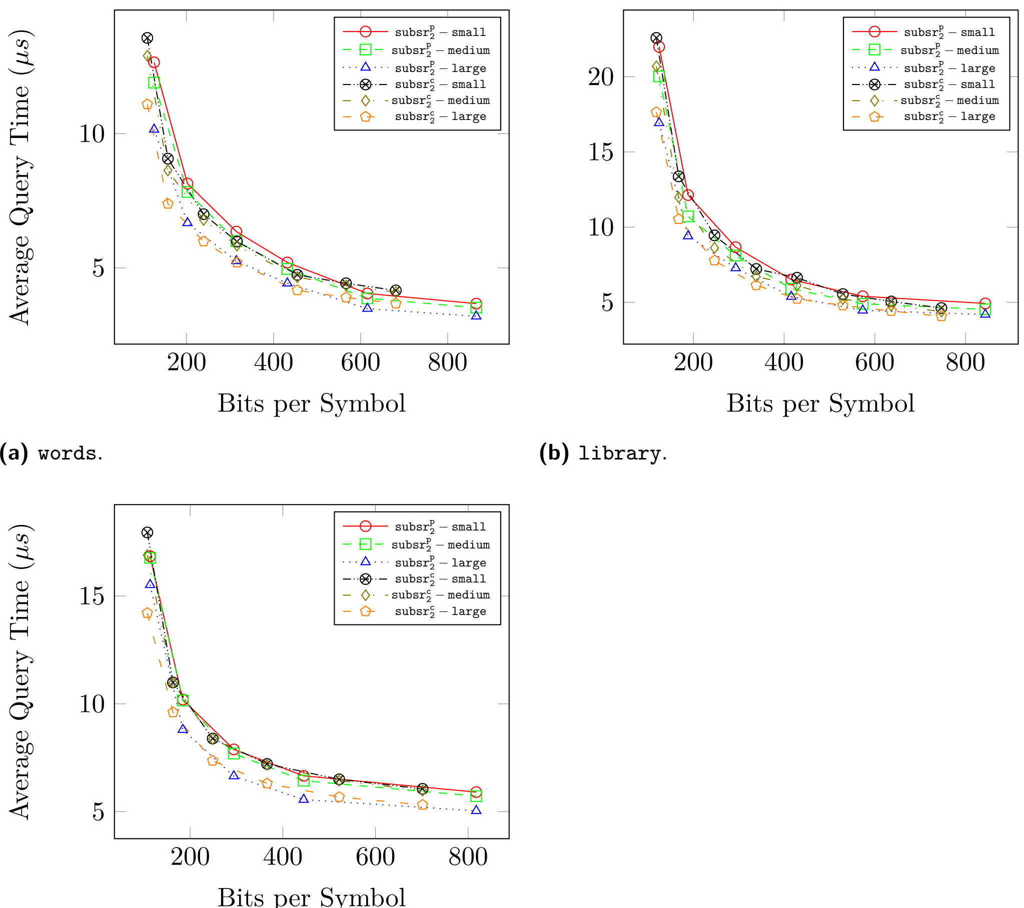
31 Bryce Sandlund and Yinzhan Xu. Faster dynamic range mode. In *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

32 Virginia Vassilevska Williams and Yinzhan Xu. Truly subcubic min-plus product for less structured matrices, with applications. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 12–29. SIAM, 2020.

A Details Omitted from Section 3.3

Figures 4, 5 and 6 are omitted figures from Section 3.3.

We also compare the time-space tradeoffs that can be achieved by `supsr` and `sqrt` with different parameters. Figure 7 shows our experimental results over `reviews` and `IPs`, in which a subfigure is used for either dataset. The results for other datasets are similar. To draw each subfigure, we construct `supsr` (and similarly `sqrt`) over each dataset for different values of s. The initial value of s is $0.5\sqrt{n}$, and each time we increase s by $0.5\sqrt{n}$ until the space usage of the data structure exceeds 640 bits per symbol. In Figure 7, our experimental study shows that `sqrt` use less query time than `supsr` when these data structures use the same space. Therefore, `sqrt` outperforms `supsr`.

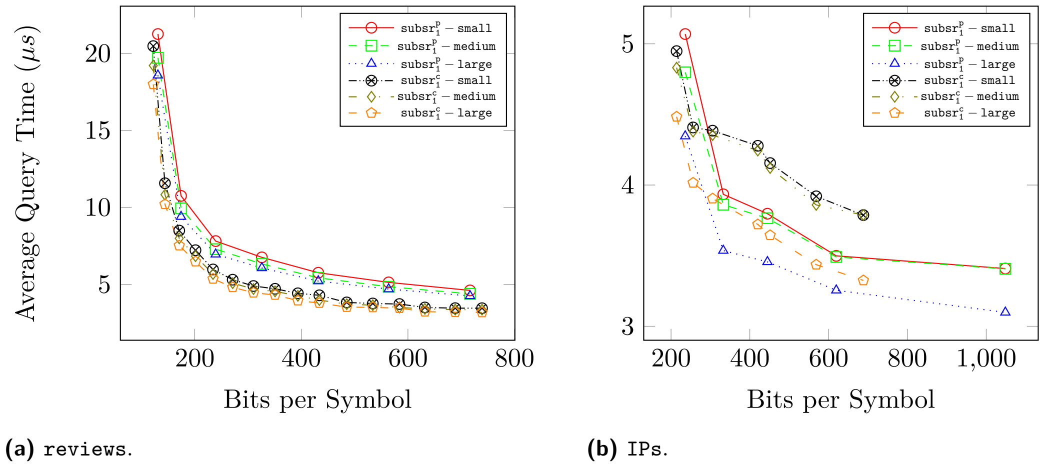


(a) `words`.

(b) `library`.

(c) `tickets`.

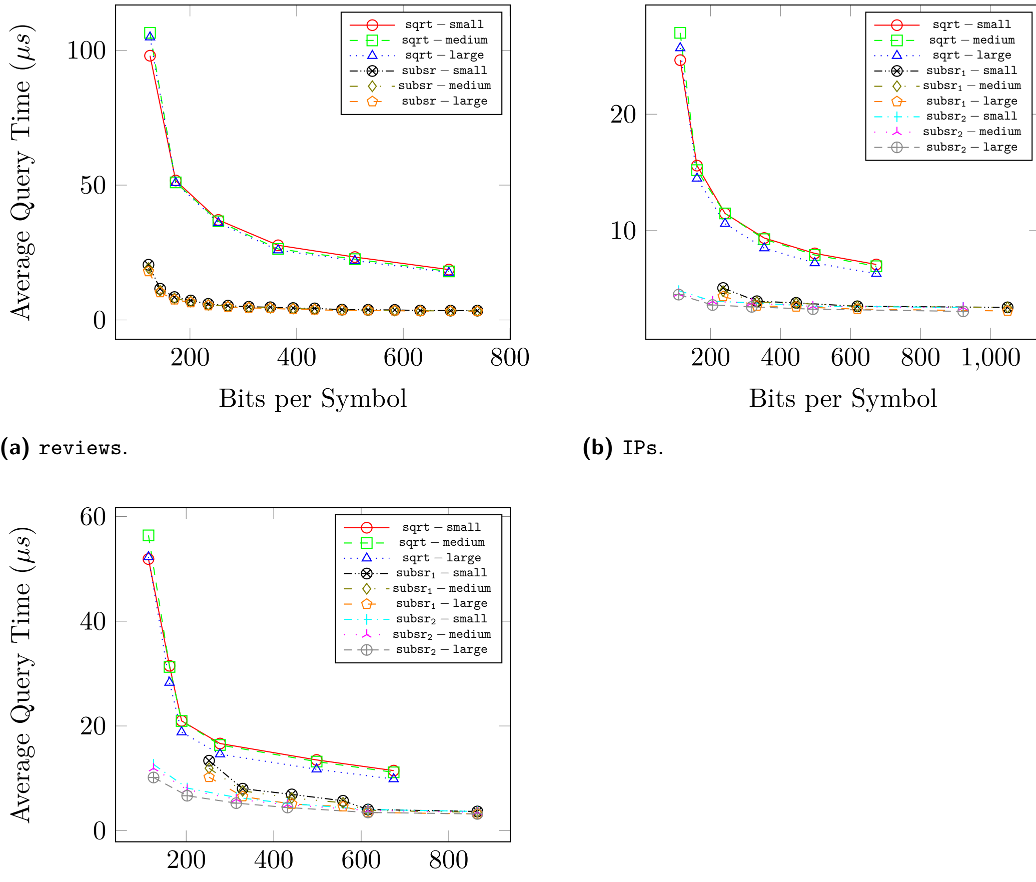
Figure 4 Time-space tradeoffs achieved by $\texttt{subsr}_2^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{c}}$ over `words`, `library` and `tickets`.



(a) `reviews`.

(b) `IPs`.

Figure 5 Time-space tradeoffs achieved by $\texttt{subsr}_1^{\texttt{p}}$ and $\texttt{subsr}_1^{\texttt{c}}$ over `reviews` and `IPs`

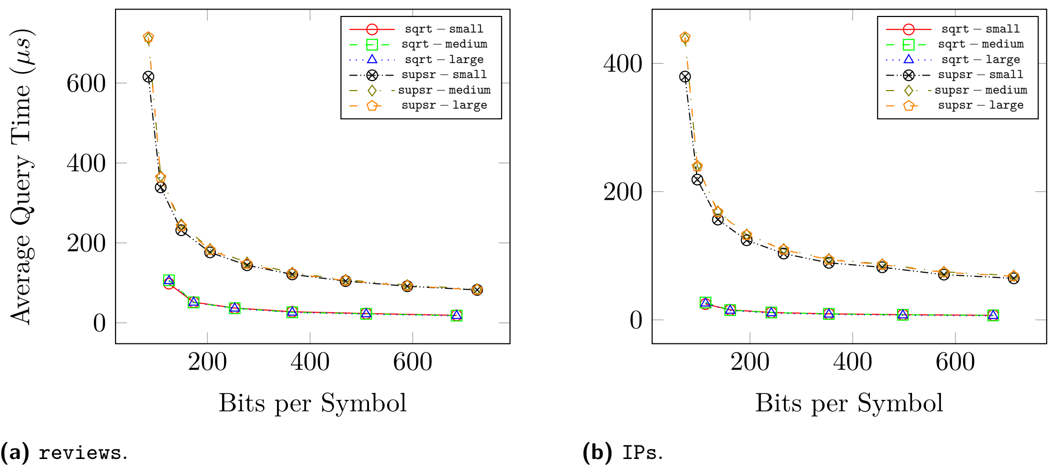


(a) `reviews`.

(b) `IPs`.

(c) `words`.

Figure 6 Time-space tradeoffs achieved by `sqrt`, `subsr`$_1$ and `subsr`$_2$ over `reviews`, `IPs` and `words`.



(a) `reviews`.

(b) `IPs`.

Figure 7 Time-space tradeoffs achieved by `supsr` and `sqrt` over `reviews` and `IPs`.

B Comparing exact range mode and approximate range mode data structures on reviews

Figure 8 compares tradeoffs achieved by exact and approximate range mode structures over reviews. Due to its high space costs, the figures do not show simple. We also omit some tradeoffs with low space cost that can be achieved using subsr2, because their query times are so large that, with them, it would not be possible to tell how other tradeoffs compare to each other in the same figure.

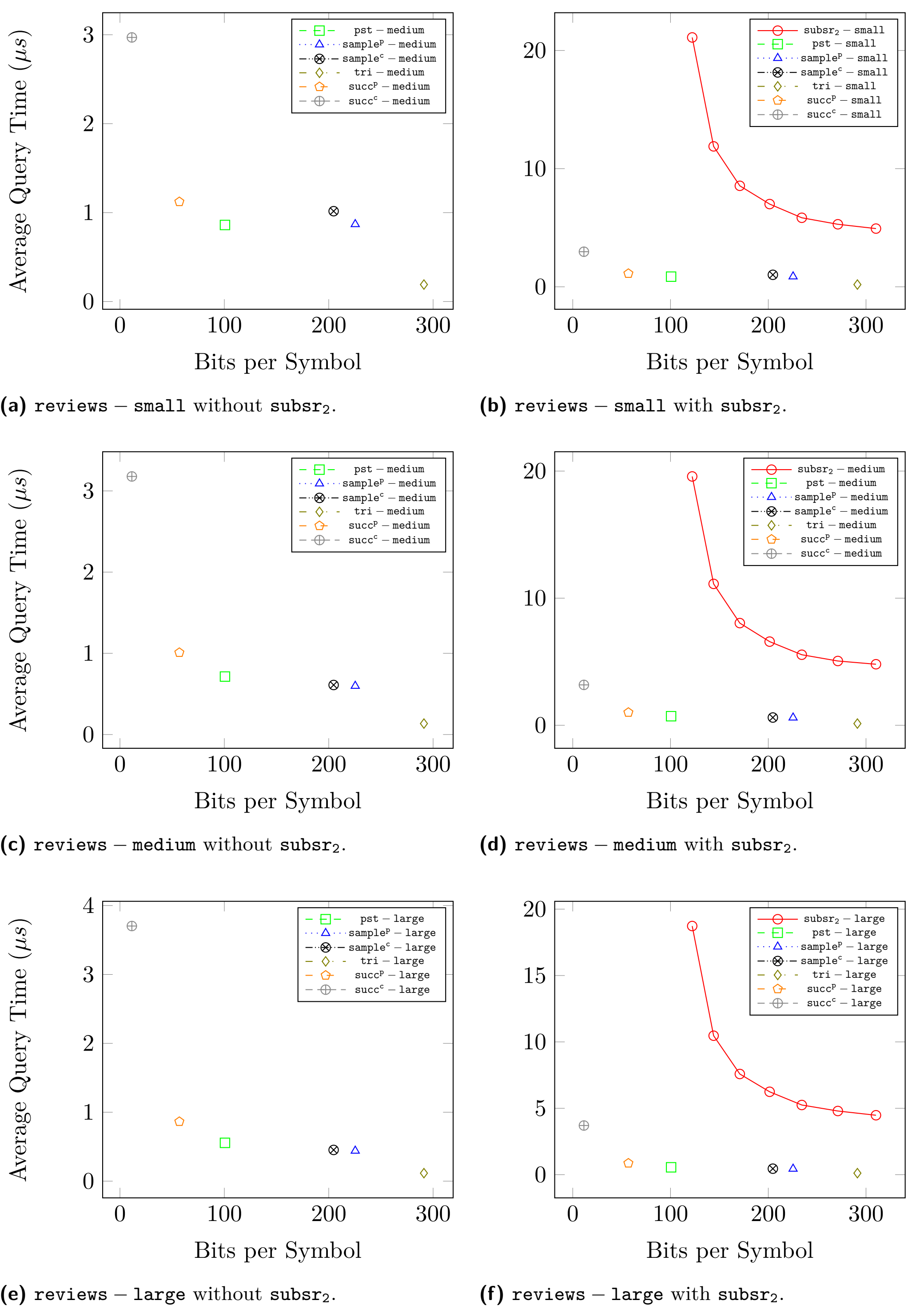
C Even Smaller Queries Ranges

In the experimental studies reported in Section 3, we adopt the method in [8, 20] to generate small, medium and large queries. To confirm whether this is appropriate for our experimental studies, we further perform additional studies using query ranges of sizes 10^1, 10^2, 10^3, 10^4 and 10^5, most of which are even smaller than the average size of our small queries, to see whether exact and approximate solutions still compare similarly for these query ranges. To run these experiments, for each $i \in \{1, 2, 3, 4, 5\}$, we generate 10^6 query ranges of size 10^i by choosing the starting positions of the ranges uniformly at random.

Exact query structures can achieve different time-space tradeoffs when setting the parameter s to different values. For a fair comparison, we binary search on s to make space costs as close to 300bps as possible. For example, for words, we set s to 4613, 16792 and 29748 for sqrt, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ to achieve space costs of 300.7bps, 300.3bps and 300.3bps, respectively. For library, we set s to 5614, 22853 and 38859 for sqrt, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ to achieve space costs of 300.7bps, 297.3bps and 300.0bps, respectively. We also include $\texttt{nv}_1$ to find out when data structure solutions outperform this naive solution. The other naive solution, $\texttt{nv}_2$, is not included; since it uses an array of size Δ, smaller query sizes will make it compare more poorly to others. Figure 9 presents our experimental results on words and library, and the results on other datasets are similar. These figures show that, for small query sizes under 100, the query times of all solutions including $\texttt{nv}_1$ are close, but after query sizes exceed 100 or so, data structure solutions start to outperform $\texttt{nv}_1$ significantly, and they compare to each other similarly as they did during the studies in Section 3.3. We also observe that, when query sizes increase, all data structure query times first increase due to the scan of more entries of A. Later, when query ranges are big enough (starting from somewhere between 10^3 and 10^4) to include multiple blocks of A, the table S is used, so the query algorithms need not scan more array entries. Instead, the query times decrease slowly when query sizes increase due to the reasons discussed in Section 3.2.

Figure 10 shows the results for approximate range mode structures over words and library, and the results on other datasets are similar. It again shows that the conclusions in Section 3.4 apply to these query sizes. A new observation is that the query times of sample and succ decrease rapidly when query sizes drops below 10^3 and 10^2, respectively. This is because each of these solutions consists of a low frequency structure and a high frequency structure, and when query sizes are smaller, it is more likely that only the former is used which has much faster query time than the latter.

These experiments show that, when query sizes are big enough to justify the use of data structures (instead of merely using a naive solution), the same conclusions in Section 3 apply here. Hence, we conclude that it is appropriate to generate small, medium and large queries and use them throughout our studies.



(a) `reviews` – `small` without `subsr`$_2$.

(b) `reviews` – `small` with `subsr`$_2$.

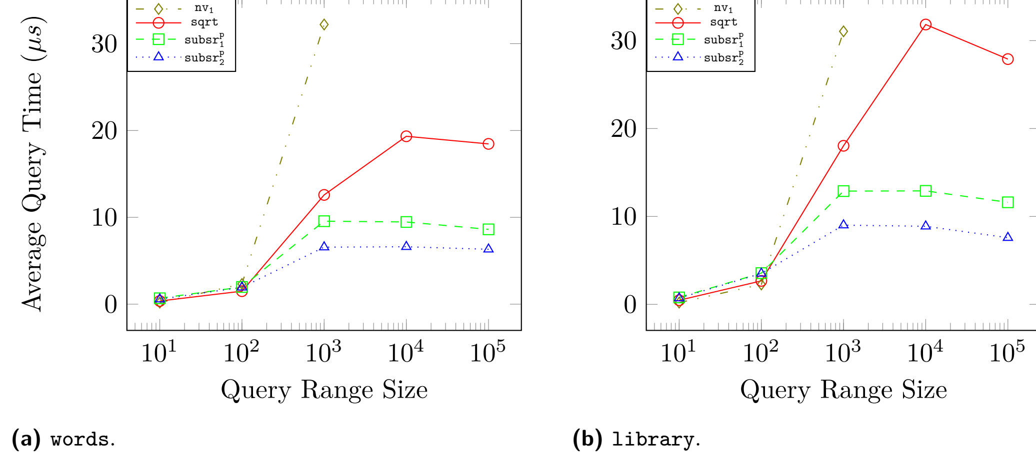
(c) `reviews` – `medium` without `subsr`$_2$.

(d) `reviews` – `medium` with `subsr`$_2$.

(e) `reviews` – `large` without `subsr`$_2$.

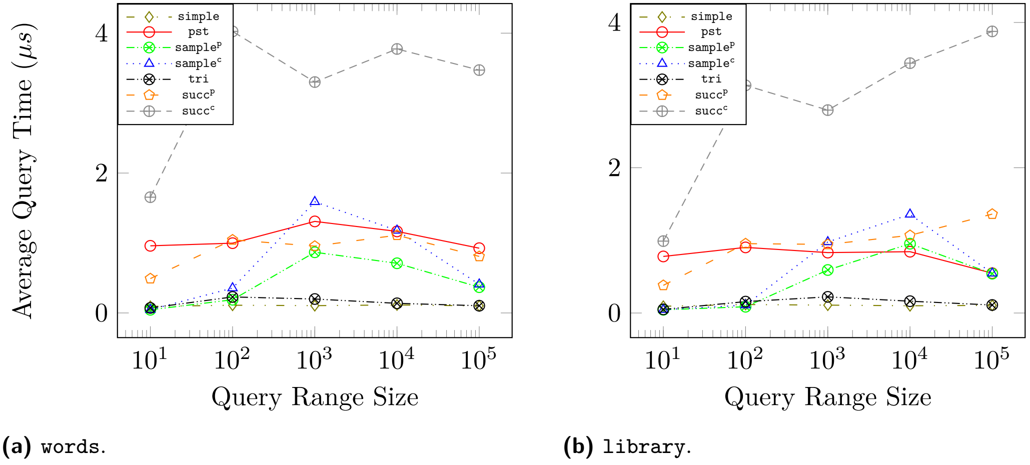
(f) `reviews` – `large` with `subsr`$_2$.

Figure 8 Time-space tradeoffs achieved by `subsr`$_2$, `pst`, `sample`^{p}, `sample`^{c}, `tri`, `succ`^{p}, and `succ`^{c} on `reviews`.



(a) `words`. **(b)** `library`.

Figure 9 Query time of exact range mode query, for query ranges of sizes from 10^1 to 10^5.



(a) `words`. **(b)** `library`.

Figure 10 Query time of approximate range mode, for query ranges of sizes from 10^1 to 10^5.

Efficient Yao Graph Construction

Daniel Funke ✉
Karlsruhe Institute of Technology, Germany

Peter Sanders ✉
Karlsruhe Institute of Technology, Germany

Abstract

Yao graphs are geometric spanners that connect each point of a given point set to its nearest neighbor in each of k cones drawn around it. Yao graphs were introduced to construct minimum spanning trees in d dimensional spaces. Moreover, they are used for instance in topology control in wireless networks. An optimal $\mathcal{O}(n \log n)$-time algorithm to construct Yao graphs for a given point set has been proposed in the literature but – to the best of our knowledge – never been implemented. Instead, algorithms with a quadratic complexity are used in popular packages to construct these graphs. In this paper we present the first implementation of the optimal Yao graph algorithm. We engineer the data structures required to achieve the $\mathcal{O}(n \log n)$ time bound and detail algorithmic adaptations necessary to take the original algorithm from theory to practice. We propose a priority queue data structure that separates static and dynamic events and might be of independent interest for other sweepline algorithms. Additionally, we propose a new Yao graph algorithm based on a uniform grid data structure that performs well for medium-sized inputs. We evaluate our implementations on a wide variety of synthetic and real-world datasets and show that our implementation outperforms current publicly available implementations by at least an order of magnitude.

2012 ACM Subject Classification Theory of computation → Sparsification and spanners

Keywords and phrases computational geometry, geometric spanners, Yao graphs, sweepline algorithms, optimal algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.20

Related Version *Technical report*: `arXiv:2303.07858` [11]

Supplementary Material *Software*: `https://github.com/dfunke/YaoGraph`
archived at `swh:1:dir:c9682e5265d00eef03757c009b3aaeefdeaa5288`

1 Introduction

Yao graphs are geometric spanners that connect each point of a given point set to its nearest neighbor in each of k cones, refer to Figure 1 for an example. A t-spanner is a weighted graph, where for any pair of vertices there exists a t-path between them, which is a path with weight at most t times their spatial distance. The parameter t is known as the *stretch factor* of the spanner. Upper bounds on the stretch factor of Yao graphs have been the subject of extensive research. While the stretch factor of Yao graphs with $k \leq 3$ cones is proofed to be unbounded, bounds have been established for all graphs with $k \geq 4$ cones [3]. Whereas for $k \geq 7$ cones the stretch factor is proofed to be bounded by the general formula $\left(1+\sqrt{2-2\cos(2\pi/k)}\right)/(2\cos(2\pi/k)-1)$, bounds on Yao graphs with 4 to 6 cones require complex individual arguments [3, 8].

Yao introduced this kind of graphs to construct minimum spanning trees in d-dimensional space [17]. Moreover, they are used for instance in topology control in wireless networks [14, 18]. Chang et al. [6] present an optimal algorithm to construct these graphs in $\mathcal{O}(n \log n)$ time. Due to the intricate nature of their algorithm and the reliance on expensive geometric constructions, to the best of our knowledge, there is no implementation of their algorithm available. Instead, an algorithm with an inferior $\mathcal{O}(n^2)$ time bound is used in the cone-based spanners package of the popular CGAL library [15].

© Daniel Funke and Peter Sanders;
licensed under Creative Commons License CC-BY 4.0

21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis; Article No. 20; pp. 20:1–20:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

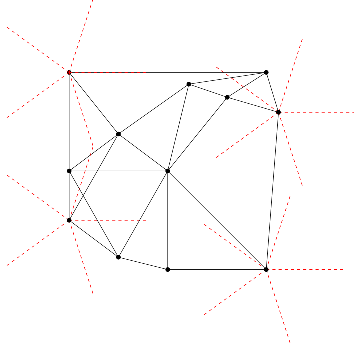


Figure 1 Yao graph for ten points and $k = 5$ cones. The five cones are illustrated as red dashed lines around four example points.

Contribution. In this paper we present the first publicly available implementation of Chang et al.'s optimal algorithm for Yao graph construction. We take their algorithm from theory to practice by engineering the data structures required to achieve the $\mathcal{O}(n \log n)$ time bound and provide detailed descriptions of all operations of the algorithm that are missing in the original paper, such as input point ordering, handling of composite boundaries and enclosing region search. In our event queue, we separate static (input point) events and dynamic (intersection point) events. This greatly improves the efficiency of priority queue operations and might be a useful technique for other sweepline algorithms. We test our algorithm on a wide range of synthetic and real-world datasets. We show that, despite the intricate nature of the algorithm and the use of expensive geometric constructions, our implementation achieves a speedup of an order of magnitude over other currently available implementations. Additionally, we develop a new Yao graph algorithm based on a uniform grid data structure that is simple to implement, easy to parallelize, and performs well for medium-sized inputs.

Outline. In Section 2 we review related work on the construction of Yao graphs. Section 3 presents the optimal algorithm of Chang et al. and algorithmic adaptions necessary for its implementation. Further implementation details such as data structures and geometric operations are described in Section 4. We evaluate our implementation and compare against its competitors in Section 5. Section 6 summarizes our paper and presents an outlook on future work.

Definitions. Given a set $\mathbf{P}$ of points in two-dimensional Euclidean space and an integer parameter $k > 1$, the Yao graph $G_k = (\mathbf{P}, \mathbf{E})$ is a directed graph, connecting every point $p \in \mathbf{P}$ with its nearest neighbor in each of k cones [17]. Every cone $\mathcal{C}_i = (\theta_L, \theta_R)$, $0 \leq i < k$, is defined by its two limiting rays with angles $\theta_L = \frac{2i\pi}{k}$ and $\theta_R = \frac{2(i+1)\pi}{k}$. We denote the cone $\mathcal{C}_i$ with apex at point p as $\mathcal{C}_i^p$. We furthermore define, that the *left* – or counterclockwise – boundary ray with angle θ_L belongs to a cone $\mathcal{C}$, whereas the *right* one does not, i. e. for a given point $p \in \mathbf{P}$ and cone $\mathcal{C}_i^p$ we define the set of points $\mathbf{P} \cap \mathcal{C}_i^p := \{q \in \mathbf{P} : \sphericalangle(p, q) \in [\theta_L, \theta_R)\}$, with $\sphericalangle(p, q)$ denoting the angle between p and q. Then the edge set $\mathbf{E}$ of the Yao graph $G_k = (\mathbf{P}, \mathbf{E})$ can be formally defined as $\mathbf{E} := \{(p, q) : \forall i \in [0, k), \forall p \in \mathbf{P}, q = \arg\min_{v \in \mathbf{P} \cap \mathcal{C}_i^p} (d(p, v))\}$, with $d(\cdot, \cdot)$ denoting the Euclidean distance function.

2 Related Work

Yao presents an $\mathcal{O}(n^{5/3}\log n)$-time algorithm to compute a solution to the *Eight-Neighbors Problem* – a Yao graph with $k = 8$. It is based on a tessellation of the Euclidean space into cells. For a given point and cone, each cell of the tessellation is characterized whether it can contain nearest neighbor candidates to reduce the number of necessary distance computations. The problem is solved optimally by Chang et al., who present a $\mathcal{O}(n\log n)$-time algorithm for constructing the Yao graph of a given point set and parameter k [6]. Their algorithm follows the same structure as Fortune's algorithm for constructing the Voronoi diagram of a point set [10], using the sweepline technique originally introduced by Bentley and Ottmann for computing line-segement intersections [4]. However, even though there are many implementations of Fortune's algorithm available, there is no implementation of Chang et al.'s Yao graph algorithm that we are aware of. Instead, for instance the CGAL library's cone-based spanners package implements a less efficient $\mathcal{O}(n^2)$-time algorithm [15]. Their algorithm is an adaption of a sweepline algorithm for constructing Θ-graphs [13]. Θ-graphs are defined similarly to Yao graphs, except that the nearest neighbor in each cone is not defined by Euclidean distance but by the projection distance onto the cone's internal angle bisector. This allows for a $\mathcal{O}(n\log n)$-time sweepline algorithm, that uses a balanced search tree as sweepline data structure to answer one-dimensional range queries [13]. For Yao graphs, such a reduction in dimensionality is not possible, thus, CGAL's algorithm employs linear search within the sweepline data structure to find the nearest neighbor, leading to the $\mathcal{O}(n^2)$ time bound. However, CGAL's algorithm is much simpler to implement than the optimal algorithm proposed by [6] and does not require geometric constructions, just predicates. Table 1 in Section 4 provides an overview of the required geometric operations by both algorithms.

3 Algorithm

In their 1990 paper, Chang et al. present an $\mathcal{O}(n\log n)$-time sweepline algorithm to compute the *oriented Voronoi diagram* (OVD) of a point set. Through a small modification, their algorithm can compute the *geographic neighborhood graph* – or Yao graph – of a point set within the same, optimal, bound [6, Theorem 3.2, Theorem 4.1].[1] To construct the Yao graph $G_k = (\mathbf{P}, \mathbf{E})$ with k cones for point set $\mathbf{P}$, k sweepline passes are required, each considering a specific cone $\mathcal{C} = (\theta_L, \theta_R)$. The sweepline for a cone $\mathcal{C}$ proceeds in direction $\tau = \frac{\theta_L+\theta_R}{2} + \pi$, i. e. opposite to the cone's internal angle bisector. Input points are swept in the order of their projection onto τ – represented as blue dashed line in Figure 2 – given by sorting

$$\rho_\tau(p) := \begin{pmatrix} x \\ y \end{pmatrix} \cdot \begin{pmatrix} \cos\tau \\ \sin\tau \end{pmatrix} \quad \forall p = \begin{pmatrix} x & y \end{pmatrix}^T \in \mathbf{P}. \tag{1}$$

All input points are inserted into an event priority queue Q with priority $\rho_\tau(p)$ and event type *input point*. Each input point p is the origin of a cone $\mathcal{C}^p$ with boundary rays B_L and B_R. Cone $\mathcal{C}^p$ defines the *region* R_p of the plane, where point p is the nearest neighbor with respect to cone $\mathcal{C}^p$ for any point being swept after p. The invariant of the algorithm is that once a point has been swept, its nearest neighbor in cone $\mathcal{C}$ has been determined. For instance, in Figure 2 p_2 is in the region of p_1, therefore p_1 is the nearest neighbor for p_2 with respect to cone $\mathcal{C}$. A boundary ray $B_\square$ always separates the regions of two input

[1] Our implementation computes the Yao graph but can easily be modified to compute the OVD.

Algorithm 1 Sweepline algorithm for cone defined by (θ_L, θ_R). $L(p, \theta, R_a, R_b)$ and $L(\overleftrightarrow{pv}, R_a, R_b)$ denote the ray originating at p with angle θ and the line segment from p to v, respectively, both separating regions R_a and R_b.

Input: points $\mathbf{P} = \{p_1, \ldots, p_n\}$ with $p_i \in \mathbb{R}^2$, cone (θ_L, θ_R)
Output: GNG $G = (V, E)$

1: $\tau \leftarrow \frac{\theta_L + \theta_R}{2} + \pi$ ▷ opposite of cone's internal angle bisector
2: $Q \leftarrow \{(\rho_\tau(p), p, I) : p \in \mathbf{P}\}$ ▷ initialize PQ with input points
3: $SL \leftarrow \emptyset$
4: $G = (V, E) \leftarrow (\mathbf{P}, \emptyset)$
5: **while** $p \leftarrow \text{popMin}(Q)$ **do**
6: **if** p is input point **then**
7: $B_L, B_R \leftarrow \text{findRegion}(p, SL)$ ▷ B_L and B_R enclose p
8: $E \cup= (p, B_L^r)$ ▷ assert $\left(B_L^r == B_R^l\right)$
9: **if** $B_L \cap B_R = v$ **then** delete v from Q
10: $B_L^* \leftarrow L(p, \theta_L + \pi, B_L^r, R_p)$
11: $B_R^* \leftarrow L\left(p, \theta_R + \pi, R_p, B_R^l\right)$
12: insert $[B_L^*, B_R^*]$ into SL between B_L and B_R
13: **if** $B_L \cap B_L^* = v$ **then** $Q \cup= (\rho_\tau(v), v)$
14: **if** $B_R \cap B_R^* = v$ **then** $Q \cup= (\rho_\tau(v), v)$
15: **if** p is intersection **then**
16: $B_L, B_R \leftarrow$ intersecting rays at p
17: $a \leftarrow B_L^l$ $b \leftarrow B_R^r$
18: **if** $B_L \cap \text{prev}(B_L) = v$ **then** delete v from Q ▷ left neighbor boundary on SL
19: **if** $B_R \cap \text{succ}(B_R) = v$ **then** delete v from Q ▷ right neighbor boundary on SL
20: $L_{BS} \leftarrow L(\frac{\vec{a}+\vec{b}}{2}, \sphericalangle(a, b) + \frac{\pi}{2}, R_a, R_b)$ ▷ bisector of line segment $\overleftrightarrow{ab}$
21: $L_L \leftarrow L(p, \theta_L + \pi, R_a, R_b)$
22: $L_R \leftarrow L(p, \theta_R + \pi, R_a, R_b)$
23: **if** $L_L \cap L_{BS} = \emptyset = L_{BS} \cap L_R$ **then** ▷ bisector intersects no line from p
24: $B^* = L\left(p, \pi + \begin{cases} \theta_L & \text{if } \rho_\tau(a) < \rho_\tau(b) \\ \theta_R & \text{else} \end{cases}, R_a, R_b\right)$
25: **if** $L_L \cap L_{BS} = p = L_{BS} \cap L_R$ **then** ▷ bisector intersects both lines in p
26: $B^* = L(p, \sphericalangle(a, b) + \frac{\pi}{2}, R_a, R_b)$
27: **if** $L_L \cap L_{BS} = v$ or $L_{BS} \cap L_R = v$ **then** ▷ bisector intersects one line in v
28: $B^* = L(\overleftrightarrow{pv}, R_a, R_b) + L(v, \sphericalangle(a, b) + \frac{\pi}{2}, R_a, R_b)$
29: $Q \cup= (\rho_\tau(v), v)$ ▷ deletion event
30: replace $[B_L, B_R]$ in SL with B^*
31: **if** $B^* \cap \text{prev}(B^*) = v$ **then** $Q \cup= (\rho_\tau(v), v)$
32: **if** $B^* \cap \text{succ}(B^*) = v$ **then** $Q \cup= (\rho_\tau(v), v)$
33: **if** p is deletion point **then**
34: $B \leftarrow$ ray belonging to p ▷ $B = L(\overleftrightarrow{pv}, R_a, R_b) + L(v, \sphericalangle(a, b) + \frac{\pi}{2}, R_a, R_b)$
35: replace B in SL with $L(v, \sphericalangle(a, b) + \frac{\pi}{2}, R_a, R_b)$
36: **return** G

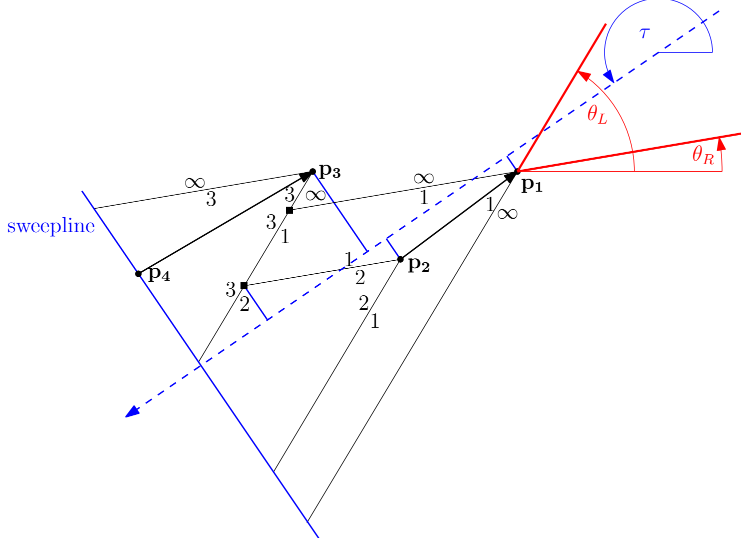


Figure 2 Example state of the sweepline algorithm for cone $\mathcal{C} = (\theta_L, \theta_R)$ (marked in red). Input points (circles, bold labels) are numbered in the order they are swept by the sweepline, with their projections on the cone's internal angle bisector shown in blue. Rays are labeled with the regions they are separating. Intersection events are marked with a square. Already determined edges of the Yao graph are indicated by arrows.

points, thus it is defined by its point of origin $B^p_\square$ and angle $B^\theta_\square$ as well as its left and right region, $B^l_\square$ and $B^r_\square$ respectively. The region outside any point's cone is labeled with infinity. Due to intersecting boundary rays, boundaries between regions can also be the union of a line segment and ray as described in detail in Section 3.2. However, for simplicity of presentation we still refer to these composite boundaries as boundary rays, unless this distinction is of relevance. The algorithm maintains an ordered data structure SL of rays currently intersecting the sweepline. The rays are ordered left-to-right and the data structure needs to support insert, remove and find operations in $\mathcal{O}(\log n)$ time, as well as access to the left and right neighbors of a given ray. In Section 4.2 we describe a balanced binary search tree that supports these operations and is tuned for our application. Algorithm 1 presents our algorithm, which is described in detail in the following. An example execution of the algorithm is depicted in Figure 11 in the appendix.

3.1 Event Types

There are three *event types*: 1) input points, 2) intersection points, and 3) deletion points.In the following, we describe how each event is handled by the algorithm. Through the execution of the algorithm, priority queue Q contains all unprocessed input points, the intersection points of the boundaries of adjacent regions as well as deletion points for composite boundaries, ordered according to $\rho_\tau(p)$. If several events coincide, their processing order can be arbitrary.

1) Input points. All points of the given set $\mathbf{P}$ are inserted into the event priority queue at the beginning of the algorithm. For an input point event with associated point p, the sweepline data structure SL is searched for the region R_q containing p. This region is defined

by its two bounding rays B_L and B_R and their associated regions $B_L^r = B_R^l = R_q$.[2] We can then add edge (p, q) to the edge set $\mathbf{E}$ of G_k, as proven in [6, Lemma 3.1]. The point p is the apex of region R_p, bounded to the left by $B_L^* = (p, \theta_L + \pi)$, separating regions R_q and R_p, and bounded to the right by $B_R^* = (p, \theta_R + \pi)$, separating regions R_p and R_q. These new rays are inserted into SL between B_L and B_R, forming the sequence $[B_L, B_L^*, B_R^*, B_R]$. Lastly, intersection points of the considered rays need to be addressed. If B_L and B_R intersect in point v, its associated intersection point event needs to be removed from the priority queue Q, as B_L and B_R are no longer neighboring rays. Instead, possible intersection points between B_L and B_L^* as well as B_R^* and B_R are added to Q for future processing.

2) Intersection points. An intersection point v is associated with its two intersecting rays B_L and B_R. They separate regions $R_p := B_L^l$, $B_L^r = R_m = B_R^l$ and $B_R^r =: R_q$, refer to Figure 3. Region R_m terminates at intersection point v and a new boundary B^* between R_p and R_q originates at v. The shape of B^* depends on the configuration of points p and q and can either be a simple ray or a union of a line segment and a ray. Section 3.2 describes in more detail how B^* is determined. B^* then replaces the sequence $[B_L, B_R]$ in the sweepline data structure. Again, intersection points of the considered rays need to be addressed. If B_L has an intersection point with its left neighbor or if B_R has an intersection point with its right neighbor, the associated intersection point events need to be removed from the event queue Q. Correspondingly, if B^* intersects its left or right neighbor, the appropriate intersection point events are added to Q.

3) Deletion event. Deletion events are not part of the original algorithm described by Chang et al. [6], as the authors do not specify how to handle composite boundaries in their paper. We use them to implement boundaries consisting of a line segment and a ray. The deletion event marks the end of the line segment and the beginning of the ray. It does not change the actual state of the sweepline data structure.

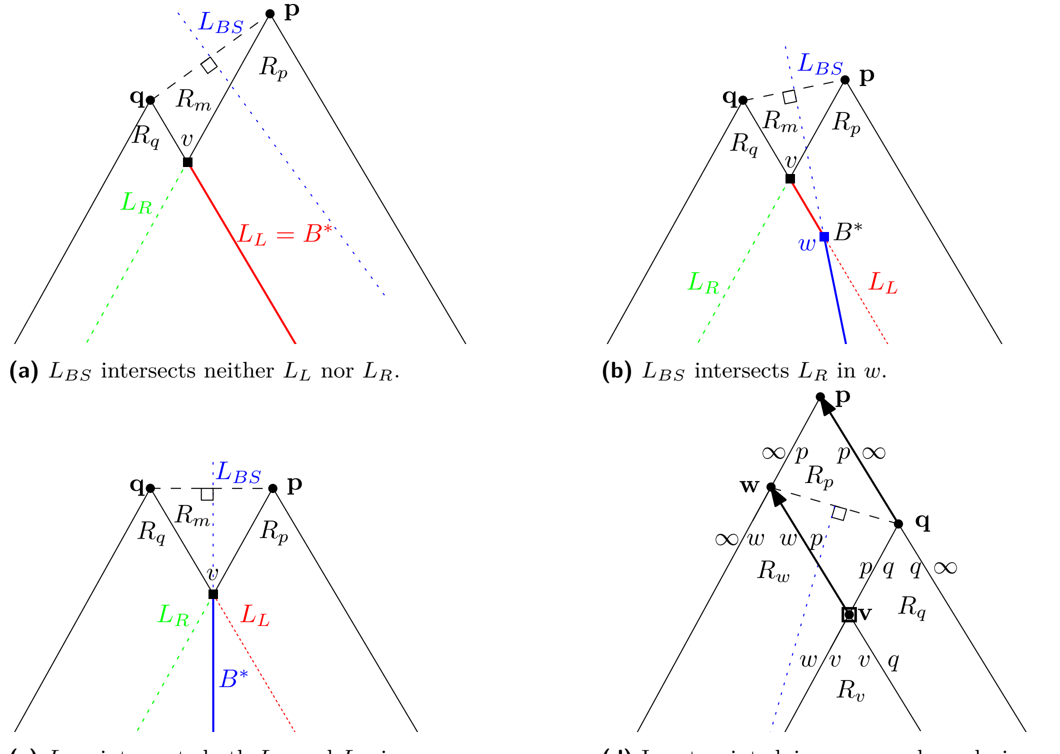
3.2 Boundary Determination

As described in the previous section, at an intersection point event v, the two intersecting rays B_L and B_R are merged into a new boundary B^*, separating regions $R_p := B_L^r$ and $R_q := B_R^l$. The shape of the boundary is determined by the configuration of points p and q. The determination is based on the number of intersection points between lines

- $L_L = (v, \theta_L + \pi)$ (green),
- $L_R = (v, \theta_R + \pi)$ (red), and
- the bisector L_{BS} of line $\vec{pq}$, $L_{BS} = (\frac{\vec{p}+\vec{q}}{2}, \sphericalangle(p, q) + \pi/2)$ (blue, dashed).

The colors refer to the lines in Figure 3 which illustrates the different cases. If L_{BS} intersects neither L_L nor L_R then $B^* = L_L = (v, \pi + \theta_R)$ if p was swept before q, otherwise $B^* = L_R = (v, \pi + \theta_R)$ (Figure 3a). Intuitively, the region of the lower point with respect to the sweepline direction continues, whereas the upper region stops at intersection point v. If L_{BS} intersects both lines L_L and L_R, then the two intersection points must coincide with v (Figure 3c). In this case, $B^* = (p, \sphericalangle(p, q) + \pi/2)$, i. e. the boundary continues with the angle of the bisector from point v. Otherwise, i. e. L_{BS} intersects either L_L or L_R in a point w, the resulting boundary B^* will be the union of the line segment $\vec{vw}$ and ray $(w, \sphericalangle(p, q) + \pi/2)$ (Figure 3b). In this case, a deletion point event is added to the priority queue at point w.

[2] Note that Chang et al. [6] explicitly store rays and regions in their sweepline data structure. However, since a region is identifiable by its two bounding rays, we choose this simpler representation of the sweepline state.



(a) L_{BS} intersects neither L_L nor L_R.
(b) L_{BS} intersects L_R in w.
(c) L_{BS} intersects both L_L and L_R in v.
(d) Input points lying on cone boundaries.

Figure 3 Illustration of the three possible configurations for boundary B^* following an intersection point event. In all examples input point p is swept before q. Lines L_L, L_R and L_{BS} are dotted, the resulting boundary B^* is denoted in bold. In Figure d), Yao graph edges are shown as bold arrows.

Input Points Colinear on Cone Boundaries. Chang et al. make the assumption that no line between two input points is with angle θ_L or θ_R. In the following we shall lift this requirement. Input points sharing a common line with angle $\theta_\square$ become *visible* from each other. This impacts the regions the passing boundary rays are separating. Refer to Figure 3d for a graphical representation of the following discussion. Recall, that the *left* – or counterclockwise – boundary ray with angle θ_L belongs to a cone $\mathcal{C}$, whereas the *right* one does not. If a boundary ray $B_\square = (p, \theta_\square, B^l_\square, B^r_\square)$ intersects an input point q then $B_\square$ terminates at q and a new boundary ray $B'_\square$ is formed. If $B_\square$ is a left boundary, i. e. $\theta_\square = \theta_L$, then edge (p, q) is added to G_K and $B'_\square$ separates regions $B^l_\square$ and R_q. If it is a right boundary then no edge is added to G_k and $B'_\square$ separates R_q and $B^r_\square$. Refer to example points q and w in Figure 3d.

3.3 Analysis

The total number of events processed by the sweepline algorithm is the sum of input point events N_{input}, intersection point events N_{IE} and deletion events N_{DE}. In order to bound the number of events processed by the sweepline algorithm, we consider the number of rays that can be present in the sweepline data structure during the execution of the algorithm.

Every input point event adds two rays to the sweepline data structure SL, resulting in a total of $2n$ rays. It possibly removes one intersection event from the event queue Q and may add up to two new such events. Every intersection point event removes two rays from SL and adds one new ray, thus reducing the sweepline size by one. Therefore, at most $2n$

Table 1 Comparison of geometric predicates used in algorithms for Yao graph construction.

	Chang et al.	CGAL	Naive	Grid
Complexity	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Predicates				
Eucl. distance comp.	X	X	X	X
dist. to line comp.	X	X		
oriented side of line	X		X	X
Constructions				
cone boundaries	X	X	X	X
box construction				X
line projection	X			
ray intersection	X			

intersection events can be processed before all rays are removed from the sweepline. An intersection event possibly removes two additional intersection events aside from itself from Q and may add one new intersection event.[3] Additionally, one deletion event may be added to Q. Therefore, at most $2n$ deletion events may be processed, each of which leaves the number of rays and intersections unchanged. In total,

$$
\begin{aligned}
N_{\text{events}} &\leq N_{\text{input}} + N_{\text{IE}} + N_{\text{DE}} \\
&\leq n + 2n + 2n = 5n
\end{aligned}
\tag{2}
$$

With a balanced binary search tree as sweepline data structures each event can be processed in $\mathcal{O}(\log n)$ time, yielding the bound of $\mathcal{O}(n \log n)$.

4 Implementation Details

In this section we highlight some of the design decisions of our implementation of Chang et al.'s Yao graph algorithm.

4.1 Geometric Kernels

The algorithm by Chang et al. [6] requires many different geometric predicates and constructions. We implement our own version of the required predicates and constructions in an *inexact* manner. Additionally, the user can employ kernels provided by the CGAL library. The EPIC kernel provides exact predicates and inexact constructions, whereas the EPEC kernel features exact predicates and exact constructions [5].

Table 1 lists the geometric predicates used by the different algorithms for Yao graph construction presented in this paper, refer to Section 5 for details on the naive and grid algorithm. Only the sweepline algorithm requires the computation of particularly costly intersections. The naive as well as grid-based Yao graph algorithm require an oriented side of line predicate only if cone boundaries are constructed exactly, in order to determine the cone $\mathcal{C}_I^p$ a point q lies in with respect to point p. Additionally, the grid algorithm could construct

[3] Technically, B^* can intersect both its neighbors, leading to two intersection events. However, when the first – as defined by $\rho_\mathcal{T}(\cdot)$ – intersection event is processed, it will delete the second event from Q, as B^* is removed from SL and a new boundary ray is inserted by the first event.

the grid data structure using exact computations. However, in our implementation we only use inexact computations to place the input points into grid cells, as we did not encounter the need for exact computation in any of our experiments. Note that the determination whether a grid cell could hold a closer neighbor than the currently found one is done using the (exact) Euclidean distance comparison predicate.

In order to reduce the number of expensive ray-ray intersection calculations, we store all found intersection points in a linear probing hash table, with the two intersecting rays as key, see e. g. Algorithm 1::13. If we need to check whether two rays are intersecting – e. g. Algorithm 1::9 – we merely require a hash table lookup.

4.2 Sweepline Data Structure

Chang et al. prove an $\mathcal{O}(n \log n)$-time complexity for the algorithm [6, Theorem 3.2]. In order to achieve this bound, the data structure maintaining the rays currently intersected by the sweepline must provide the following operations: insert, remove and predecessor search in $\mathcal{O}(\log n)$ time. Our data structure furthermore provides neighbor access in $\mathcal{O}(1)$ time.

In order to support the these operations, we use a doubly linked list of rays, with an AVL search tree on top [1]. Figure 4 shows a graphical representation of our data structure. As the order of the rays along the sweepline is known at the time of insertion – see Algorithm 1::12 – the $\mathcal{O}(\log n)$-time search phase of a traditional AVL data structure can be omitted and new rays can be inserted in a bottom-up manner. However, this optimization requires the need for parent pointers in the tree. As always two neighboring rays are inserted into the sweepline data structure at the same time, we implement a special insert operation for this case that only requires one rebalancing operation for both rays. For removal operations, the position of the ray within the sweepline is known as well – refer to Algorithm 1::30. Thus, similar to insert operations, no search phase is required for removals and the operation can be performed in a bottom-up manner. The algorithm always removes the two neighboring rays B_L and B_R and and replaces them with B^*. B^* has the same left neighbor as B_L and the same right neighbor as B_R. Therefore, we can simply replace B_L with B^* in the data structure and just need to remove B_R, leading to merely one rebalancing operation.

The search for the enclosing region of a point p – Algorithm 1::7 – is performed by finding the first ray B_R, currently intersecting the sweepline, that has p to its right. This requires the evaluation of a oriented side of line predicate at each level of the tree. The left neighbor B_L of B_R, must have p to its left or on it. Therefore B_L and B_R enclose p and $B_L^r = B_R^l = R_q$ gives the the region p is contained in. To facilitate searching, each internal node of the tree needs to refer to the rightmost ray in its subtree. As rays are complex objects, we use pointers to the corresponding leaf to save memory. Given the expensive search operations, AVL trees – as strictly height-balancing trees – are preferable to data structures with weaker balancing guarantees, such as red-black trees [16].

4.3 Priority Queue

The priority queue (PQ) Q is initialized with all input points at the beginning of the algorithm. During event processing, intersection and deletion events may be added and removed from Q, therefore requiring an addressable priority queue. The objects are ordered according to Equation (1), thus keys are (exact) numerical values. Our experiments show that, typically, for n input points, only about $\mathcal{O}(\sqrt{n})$ intersection and deletion events are in Q at any given step. Using the same PQ for all events would result in expensive dynamic PQ operations. As input point events are static in Q, we can use a two part data structure as shown in

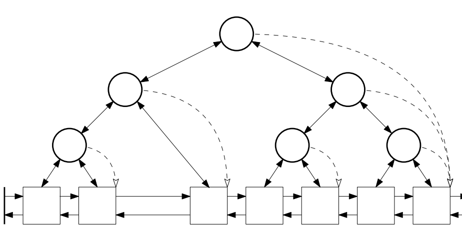


Figure 4 The sweepline data structure is a doubly linked list of rays with an AVL search tree on top. Additionally, each node has a pointer to the rightmost ray in its subtree (dashed).

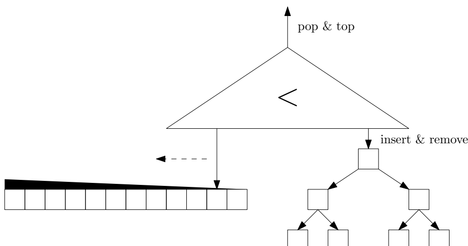


Figure 5 The priority queue consists of a *static*, sorted array of input points and a *dynamic*, addressable PQ for intersection and deletion events.

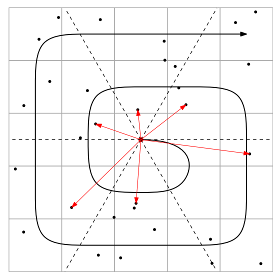


Figure 6 Grid-based Yao graph construction algorithm. The cone boundaries are represented by dashed lines. The algorithm visits grid cells in order of the thick curve. Found edges of the Yao graph are labeled in red.

Figure 5. Input point events are stored in an array – sorted by priority in Q – with a pointer to the smallest unprocessed element. Intersection and deletion events are stored in an actual addressable priority queue. We use an addressable binary heap for this part of the data structure. The TOP operation needs to compare the minimum element of the PQ with the element pointed to in the array and return the minimum of both. POP either performs a regular pop on the PQ or moves the pointer of the array to the next larger element. INSERT and REMOVE operations can access the PQ directly, as only this part of the data structure is dynamic. Thus, the actual dynamic PQ is much smaller resulting in more efficient SIFTDOWN and SIFTUP operations in the binary heap. The smaller heap size not only reduces the tree height but also makes the heap more cache friendly.

This optimization might be of interest for other algorithms that initialize their priority queue with all input points and only have a small number of dynamically added events in their priority queue at any given time. Note that the *total* number of processed intersection and deletion events surpasses the number of input points by far, however only a small number of these events are in the PQ at the same time.

5 Evaluation

In this section we evaluate our implementation on a variety of datasets against other algorithms for Yao graph generation.

Competing Algorithms. As mentioned before, we are not aware of any previous implementations of Chang et al.'s Yao graph algorithm. Therefore, we evaluate our implementation against other algorithms to construct the Yao graph of a given point set. Our main competitor

is the Yao graph algorithm from the CGAL library's cone-based spanners package [15]. As we are not aware of any other tuned implementations to construct Yao graphs, we implement two other algorithms ourselves as competition.

First, we implement a *naive* $\mathcal{O}(n^2)$-time algorithm that serves as a trivial baseline. The algorithm compares the distance of all point pairs and determines the closest neighbor in each of the k cones for a point. It requires only two geometric predicates: distance comparison and oriented side of line test. For a point pair p and q, the cone $\mathcal{C}_i^p$ that q lies in with respect to p can first be approximated by $i = {\sphericalangle(p,q)}/{k}$. Then, two oriented side of line tests suffice to exactly determine the cone q lies in, independent of the number of cones k.

Second, we implement a *grid*-based algorithm. The algorithm places all points in a uniform grid data structure [2] that splits the bounding box of all input points in $\mathcal{O}(n)$ equal-sized cells. For each input point p, the algorithm first visits p's own grid cell and computes for each point q in the cell its distance to p and the cone q lies in with respect to p. The algorithm then visits the grid cells surrounding p's cell in a spiraling manner, refer to Figure 6. For each visited cell, the algorithm computes the distances and cones for the points contained in it with respect to p until all cones of point p are settled. A cone is settled, if a neighbor v has been found within that cone and no point in adjacent grid cells can be closer to p than v. Note that some cones may remain unsettled until all grid cells have been visited if no other input points lie within that cone for point p. While the algorithm still has a $\mathcal{O}(n^2)$ worst case time complexity, it performs much better in practice.

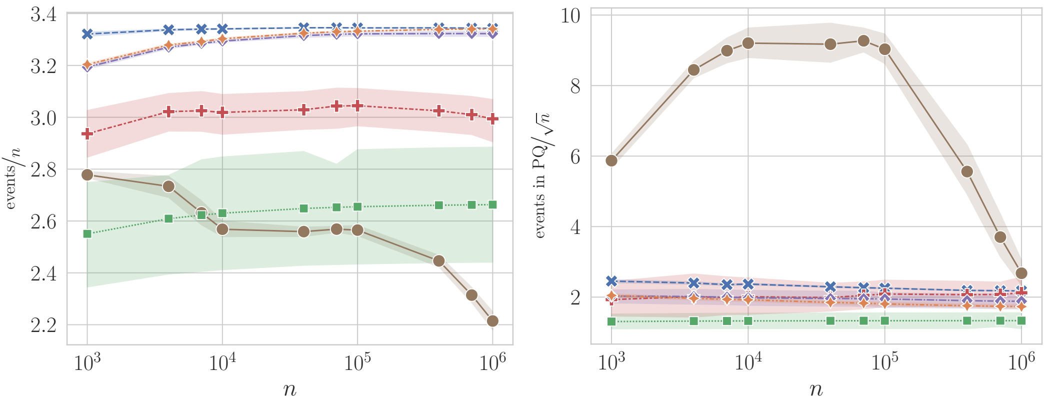
Experimental Setup. We test all algorithms on a variety of synthetic and real-world datasets. We use input point sets distributed uniformly and normally in the unit square, as well as points lying on the circumference of a circle and at the intersections of a grid [12] – the former being a worst case input for the grid algorithm, the latter being a bad case for numerical stability. We furthermore use two real-world datasets: intersections in road networks and star catalogs. As road networks we use graphs from the 9th DIMACS implementation challenge [9]. To generate a road network of a desired size n from the Full USA graph, we use a random location and grow a rectangular area around it until at least n nodes are within the area. US cities feature many points on a grid and therefore present a challenge for numerical stability. We furthermore use the Gaia DR2 star catalog [7], which contains celestial positions for approximately 1.3 billion stars. We use a similar technique as for road networks to generate subgraphs of a desired size. Here, we grow a cube around a random starting location until the desired number of stars fall within it. We then project all stars onto the xy-plane as 2D input for our experiments. Figure 10 in the appendix shows examples of our input datasets and resulting Yao graphs.

We implemented all algorithms in our C++ framework YaoGraph, available on Github.[4] Our code was compiled using GCC 12.1.0 with CGAL version 5.0.2. All experiments were run on a server with an Intel Xeon Gold 6314U CPU with 32 cores and 512 GiB of RAM. For experiments we used three different random seeds and $k = 6$ unless otherwise specified.

5.1 Algorithmic Metrics

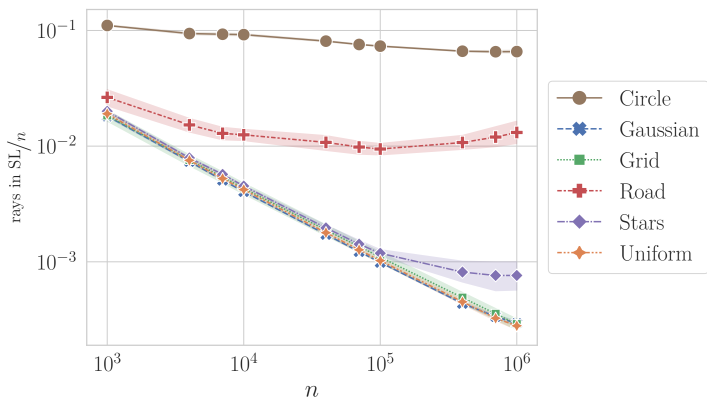
Firstly, we discuss relevant properties of the sweepline algorithm. Figure 7a shows the number of events processed per input point by the algorithm. Each input point has one input point event and generates 2.3 intersection and/or deletion events on average, with very little

[4] https://github.com/dfunke/YaoGraph



(a) Total number of events processed.

(b) Maximum number of intersection and deletion events concurrently in PQ.



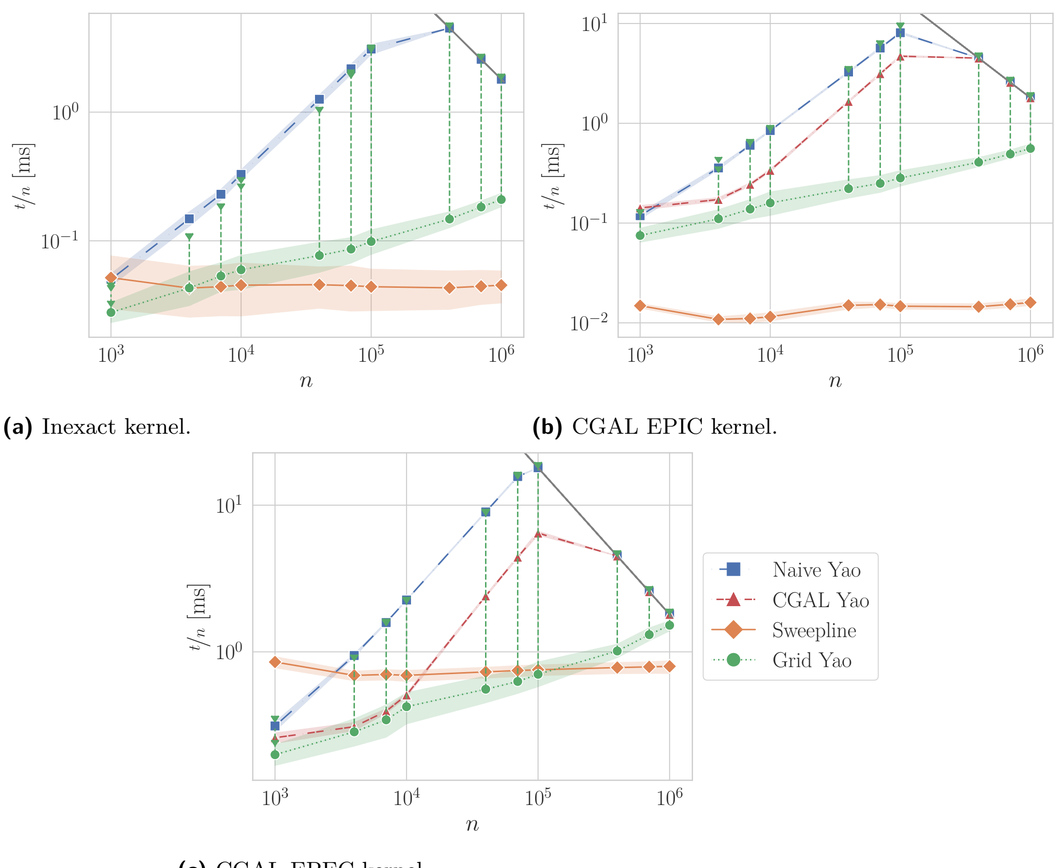
(c) Maximum number of rays in sweepline data structure.

Figure 7 Statistics for varying input point distribution and point set sizes for $k = 6$ cones. Error bands give the variation over the different cones being calculated.

variance with regard to input size and distribution – except for the grid distribution and road graphs. Both exhibit larger variance, depending on whether the cone's boundaries coincide with grid lines or not. Figure 7b shows the maximum number of intersection and deletion events that are in the priority queue at any given time during the algorithm execution. This number scales with $\mathcal{O}(\sqrt{n})$ for most studied inputs, which motivates our choice of the two-part priority queue as discussed in Section 4.3. The behavior of the circle distribution requires further investigation. The maximum number of rays in the sweepline data structure at any point during algorithm execution shows no clear scaling behavior, refer to Figure 7c. It scales with $\mathcal{O}(\sqrt{n})$ for most synthetic input sets, but approaches a constant fraction of the input size for the circle (≈10 %), road (≈1 %) and star (≈0.1 %) datasets.

5.2 Runtime Evaluation

Figure 8 shows the results of our runtime experiments. Plots Figure 8a to Figure 8c show the (scaled) running time of the algorithms, displaying variations due to input distributions as error bands. Figure 12 in the appendix gives a more detailed picture of the runtime for the different distributions. Note that only the grid and the sweepline algorithm are sensitive



(a) Inexact kernel.

(b) CGAL EPIC kernel.

(c) CGAL EPEC kernel.

Figure 8 Algorithm runtime experiments. Experiments over varying input sizes are performed with $k = 6$ cones. Error bands give the runtime variation over the different input point distributions. For the grid algorithm, the circle distribution is plotted separately with triangular markers. The gray line represents the time limit of 30 min per algorithm. Experiments over varying number of cones are with $n = 1 \times 10^5$ uniformly distributed input points.

to the input point distribution. As previously seen in Figure 7a, the number of processed events by the sweepline algorithm is relatively stable for all distributions. Therefore only little variation is seen in the runtime of the algorithm. This also shows, that the size of the sweepline data structure has only negligible influence on the algorithm runtime, as no higher runtime is observed for the road or circle datasets. Our inexact kernel shows more runtime variation than CGAL's highly optimized kernels, mainly due to the grid distribution with its many points directly on cone boundaries. The sweepline algorithm clearly outperforms CGAL's Yao graph implementation. Furthermore, even though the sweepline algorithm requires much more involved computations, it is superior to the simple grid algorithm for non-exact constructions. Only for exact constructions, large inputs are required to negate the more expensive operations of the sweepline algorithm. The exact construction kernel leads to runtime overhead of 100 compared to the EPIC kernel. However, if points lie directly on cone boundaries, exact constructions are necessary to obtain correct results, as seen in Figure 10c in the appendix. The data dependency is more pronounced for the grid algorithm, which performs well for most datasets but degenerates to the naive algorithm for the circle distribution, due to the many empty grid cells in the circle's interior.

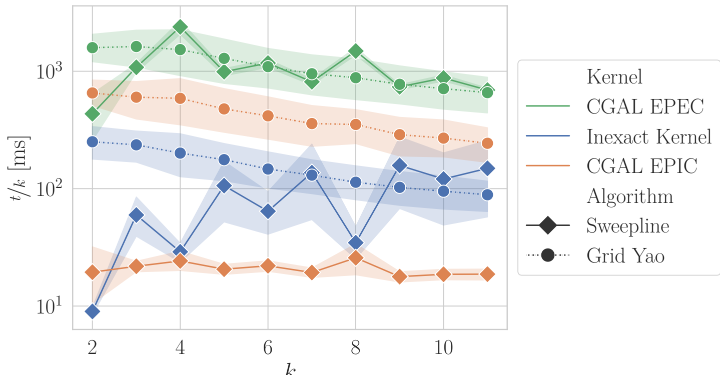


Figure 9 Algorithm runtime experiments. Experiments over varying number of cones are with $n = 1 \times 10^5$ uniformly distributed input points.

To compute a Yao graph with k cones, the sweepline algorithm requires k passes. This linear relationship can be seen in Figure 9. The grid algorithm has no dependency on k – except for the size of the neighborhood of a point. However, our experiments show that the runtime of the algorithm increases with increasing k. We attribute this to the fact that more grid cells need to be visited in order to settle all cones of a point p, since with narrower cones, chances are higher that no points lying in a specific cone of p are within a visited grid cell. We did not perform these experiments with the naive algorithm or the CGAL algorithm, due to their long runtimes. CGAL's algorithm also requires one pass per cone, whereas the naive algorithm's runtime dependency on k is negligible.

6 Conclusion

We present the – to the best of our knowledge – first implementation of Chang et al.'s optimal $\mathcal{O}(n \log n)$-time Yao graph algorithm. Our implementation uses carefully engineered data structures and algorithmic operations and outperforms current publicly available Yao graph implementations – particularly CGAL's cone-based spanners package – by at least an order of magnitude. We furthermore present a very simple grid-based Yao graph algorithm that also outperforms CGAL's implementation, but is inferior to Chang et al.'s algorithm for larger input. However, the algorithm could be further improved by using a precomputed mapping of the grid neighborhood to cones, in order to only visit grid cells that can contain points in hitherto unsettled cones. Moreover, the algorithm is trivially parallelizable over the input points, whereas Chang et al.'s algorithm can only be easily parallelized over the k cones. The parallelization within one sweepline pass remains for future work.

References

1 Georgy Maksimovich Adelson-Velsky and Evgeny Mikhailovich Landis. An algorithm for organization of information. In *Doklady Akademii Nauk*, volume 146, pages 263–266. Russian Academy of Sciences, 1962.

2 V. Akman, W.R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and uniform grid technique. *Computer-Aided Design*, 21(7):410–420, 1989. doi:10.1016/0010-4485(89)90125-5.

3 Luis Barba, Prosenjit Bose, Mirela Damian, Rolf Fagerberg, Wah Loon Keng, Joseph O'Rourke, André Van Renssen, Perouz Taslakian, Sander Verdonschot, and Ge Xia. New and improved spanning ratios for Yao graphs. *Journal of Computational Geometry*, 6(2):19–53, 2015. doi:10.20382/JOCG.V6I2A3.

4 J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, pages 643–647, 1979.

5 Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra. 2D and 3D Linear Geometry Kernel. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.5.1 edition, 2022. URL: https://doc.cgal.org/5.5.1/Manual/packages.html#PkgKernel23.

6 Maw Shang Chang, Nen-Fu Huang, and Chuan-Yi Tang. An optimal algorithm for constructing oriented Voronoi diagrams and geographic neighborhood graphs. *Information Processing Letters*, 35(5):255–260, 1990. doi:10.1016/0020-0190(90)90054-2.

7 Gaia Collaboration. Gaia Data Release 2 - Summary of the contents and survey properties. *Astronomy & Astrophysics*, 616:A1, 2018. doi:10.1051/0004-6361/201833051.

8 Mirela Damian and Naresh Nelavalli. Improved bounds on the stretch factor of Y4. *Computational Geometry*, 62:14–24, April 2017. doi:10.1016/j.comgeo.2016.12.001.

9 Camil Demetrescu, Andrew V Goldberg, and David S Johnson. *The shortest path problem: Ninth DIMACS implementation challenge*, volume 74. American Mathematical Soc., 2009.

10 Steven Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(1):153–174, 1987.

11 Daniel Funke and Peter Sanders. Efficient Yao Graph Construction, 2023. arXiv:2303.07858.

12 Daniel Funke, Peter Sanders, and Vincent Winkler. Load-Balancing for Parallel Delaunay Triangulations. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 156–169, Cham, 2019. Springer International Publishing.

13 Giri Narasimhan and Michiel Smid. *Geometric spanner networks*. Cambridge University Press, 2007.

14 Christian Schindelhauer, Klaus Volbert, and Martin Ziegler. Geometric spanners with applications in wireless networks. *Computational Geometry*, 36(3):197–214, 2007.

15 Weisheng Si, Quincy Tse, and Frédérik Paradis. Cone-Based Spanners. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.4 edition, 2022. URL: https://doc.cgal.org/5.4/Manual/packages.html#PkgConeSpanners2.

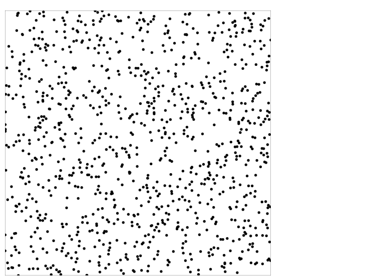
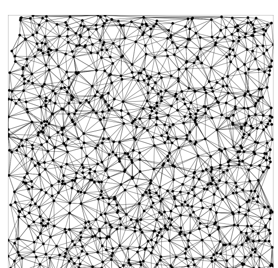
16 Svetlana Štrbac-Savić and Milo Tomašević. Comparative performance evaluation of the AVL and red-black trees. In *Proceedings of the Fifth Balkan Conference in Informatics*. ACM, 2012. doi:10.1145/2371316.2371320.

17 Andrew Chi-Chih Yao. On Constructing Minimum Spanning Trees in k-Dimensional Spaces and Related Problems. *SIAM Journal on Computing*, 11(4):721–736, 1982. doi:10.1137/0211059.

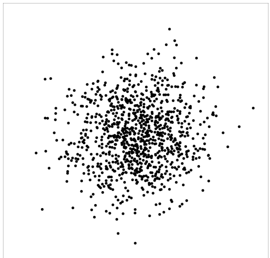
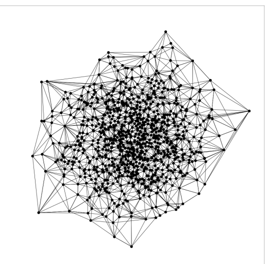
18 Xiujuan Zhang, Jiguo Yu, Wei Li, Xiuzhen Cheng, Dongxiao Yu, and Feng Zhao. Localized Algorithms for Yao Graph-Based Spanner Construction in Wireless Networks Under SINR. *IEEE/ACM Transactions on Networking*, 25(4):2459–2472, 2017. doi:10.1109/TNET.2017.2688484.

A Evaluation

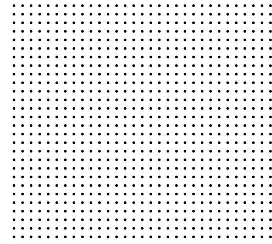
A.1 Input Point Distributions

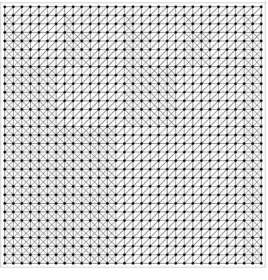
(a) Uniform distribution.

(b) Gaussian distribution.

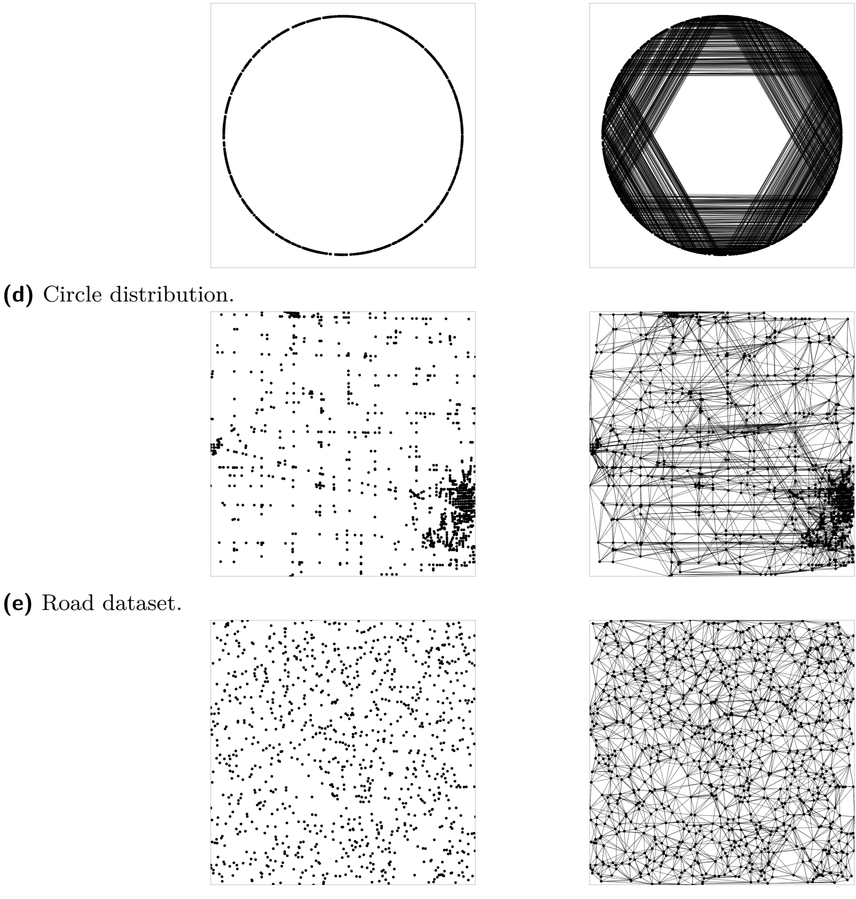


EPEC kernel



EPIC kernel

(c) Grid distribution.



(d) Circle distribution.

(e) Road dataset.

(f) Star dataset.

Figure 10 Input distributions for $n = 1000$ points and resulting Yao graph for $k = 6$. For the grid distribution, the resulting graphs from exact constructions and inexact constructions are shown.

A.2 Example Execution

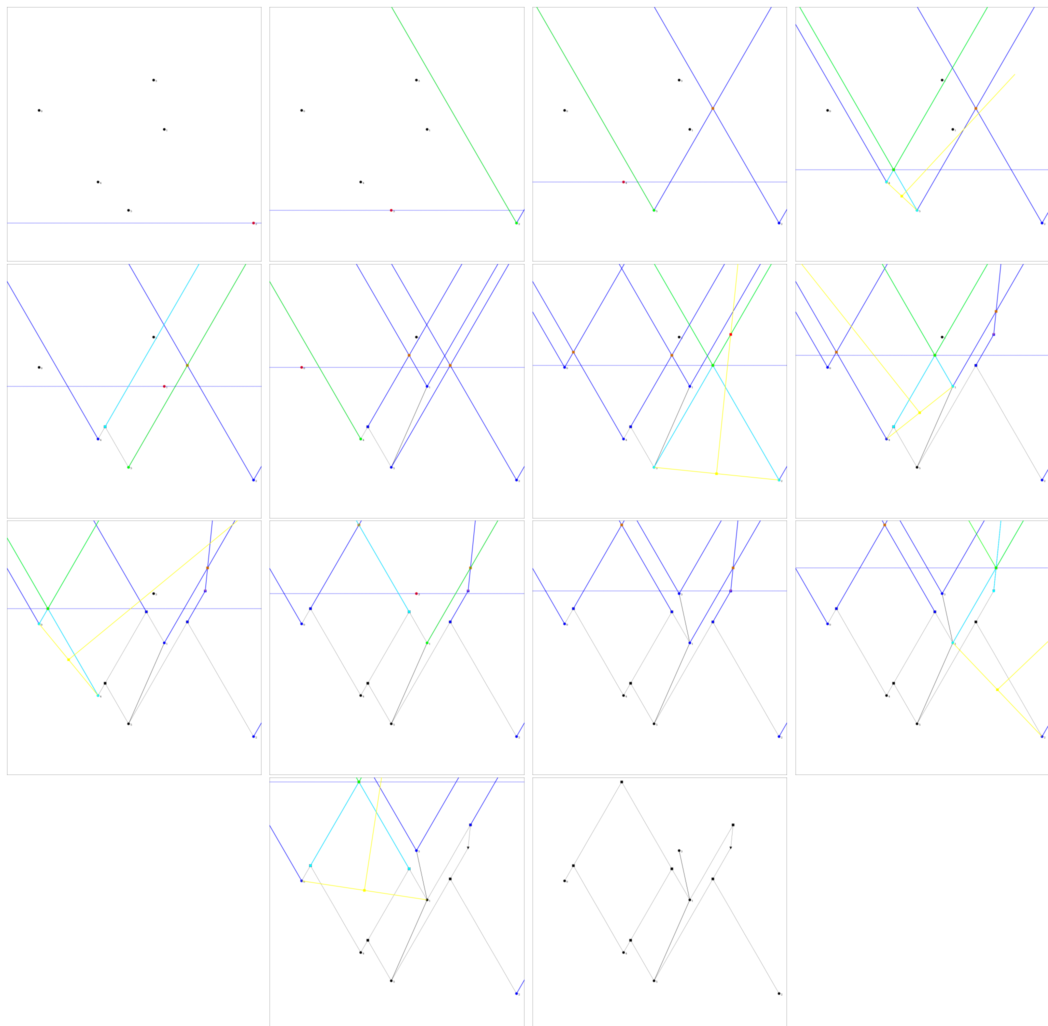
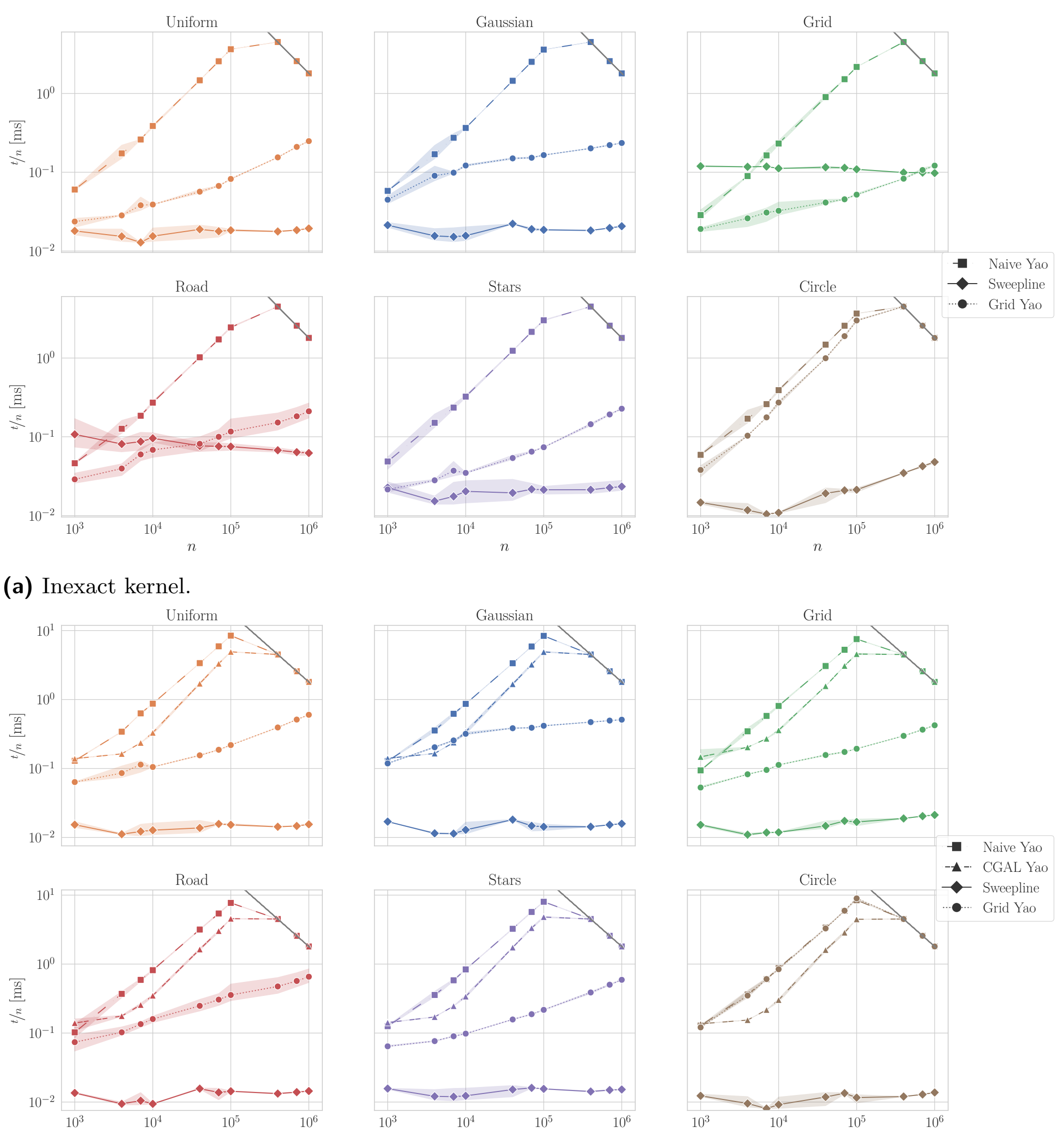


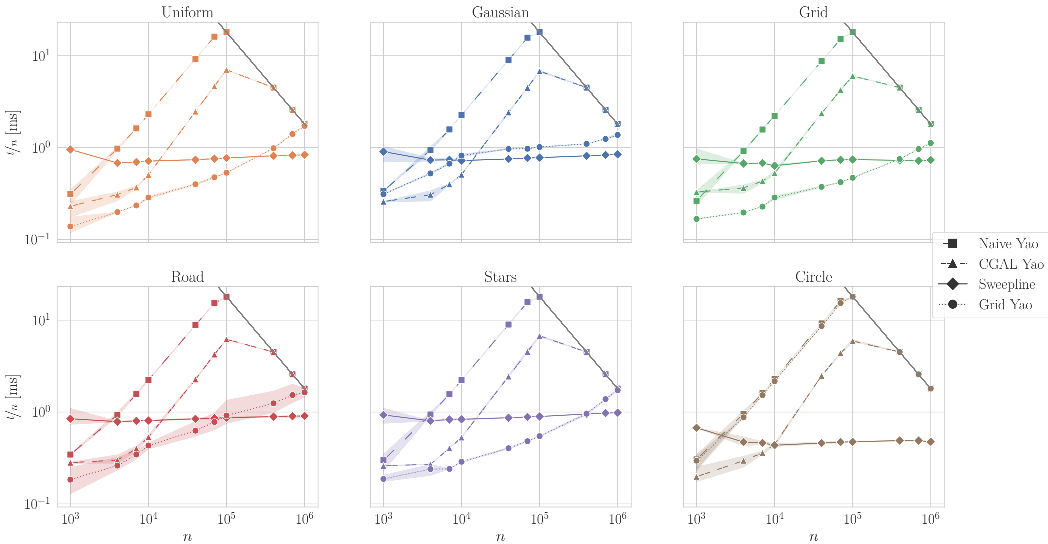
Figure 11 Sample execution for $n = 6$ points and cone $\mathcal{C} = ({}^{4\pi}/_{3}, {}^{5\pi}/_{3})$, resulting in $\tau = {}^{\pi}/_{2}$ (upward). The currently processed point is marked in red, the sweepline is a dashed, blue line. All rays currently intersecting the sweepline are blue, except for the left boundary ray B_L (cyan) and right boundary ray B_R (green). Intersection points are marked as squares, deletion points as triangles. For intersection events, the intersecting rays are cyan and the the bisector line is yellow. Edges of the Yao graph are solid black lines and settled cone boundaries are dashed.

A.3 Results



(a) Inexact kernel.

(b) CGAL EPIC kernel.



(c) CGAL EPEC kernel.

Figure 12 Algorithm runtime experiments. Experiments over varying input sizes are performed with $k = 6$ cones. The gray line represents the time limit of 30 min per algorithm.

Maximum Coverage in Sublinear Space, Faster

Stephen Jaud
School of Computing and Information Systems, The University of Melbourne, Australia

Anthony Wirth
School of Computing and Information Systems, The University of Melbourne, Australia

Farhana Choudhury
School of Computing and Information Systems, The University of Melbourne, Australia

Abstract

Given a collection of m sets from a universe $\mathcal{U}$, the *Maximum Set Coverage* problem consists of finding k sets whose union has largest cardinality. This problem is NP-Hard, but the solution can be approximated by a polynomial time algorithm up to a factor $1-1/e$. However, this algorithm does not scale well with the input size.

In a streaming context, practical high-quality solutions are found, but with space complexity that scales linearly with respect to the size of the universe $n = |\mathcal{U}|$. However, one randomized streaming algorithm has been shown to produce a $1-1/e-\varepsilon$ approximation of the optimal solution with a space complexity that scales only poly-logarithmically with respect to m and n. In order to achieve such a low space complexity, the authors used two techniques in their multi-pass approach:

- F_0-*sketching*, allows to determine with great accuracy the number of distinct elements in a set using less space than the set itself.
- *Subsampling*, consists of only solving the problem on a subspace of the universe. It is implemented using γ-independent hash functions.

This article focuses on the sublinear-space algorithm and highlights the time cost of these two techniques, especially subsampling. We present optimizations that significantly reduce the time complexity of the algorithm. Firstly, we give some optimizations that do not alter the space complexity, number of passes and approximation quality of the original algorithm. In particular, we reanalyze the error bounds to show that the original independence factor of $\Omega(\varepsilon^{-2}k\log m)$ can be fine-tuned to $\Omega(k\log m)$; we also show how F_0-*sketching* can be removed. Secondly, we derive a new lower bound for the probability of producing a $1-1/e-\varepsilon$ approximation using only pairwise independence: $1-\frac{4}{ck\log m}$ compared to $1-\frac{2e}{m^{ck/6}}$ with $\Omega(k\log m)$-independence.

Although the theoretical guarantees are weaker, suggesting the approximation quality would suffer, for large streams, our algorithms perform well in practice. Finally, our experimental results show that even a pairwise-independent hash-function sampler does not produce worse solution than the original algorithm, while running significantly faster by several orders of magnitude.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases streaming algorithms, subsampling, maximum set cover, k-wise independent hash functions

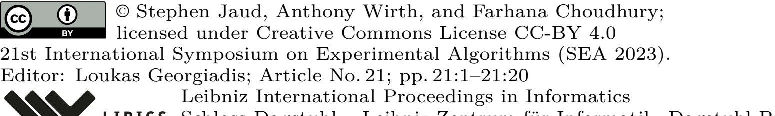
Digital Object Identifier 10.4230/LIPIcs.SEA.2023.21

Related Version *Full Version*: https://arxiv.org/abs/2302.06137

Supplementary Material
Software (Source Code): https://github.com/caesiumCode/streaming-maximum-cover
archived at swh:1:dir:1012da79a9177f4dc0ae4e5851608b597e79fa8d

Funding This research was supported by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (project DP190102078).
Farhana Choudhury: Farhana Choudhury is a recipient of the ECR22 grant from The University of Melbourne.

Acknowledgements Rowan Warneke, for reading and advising on an earlier version.


© Stephen Jaud, Anthony Wirth, and Farhana Choudhury;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Maximum Coverage, also known as *Maximum-k-Coverage* is a classic problem in computer science. Unless P = NP, the decision version is unsolvable in polynomial time. The input is a **family** of m sets, $\mathcal{F}$, each a subset of universe $\mathcal{U}$, comprising n elements, and a positive **integer**, k. The task is to find a subfamily of k sets in $\mathcal{F}$ whose union has largest cardinality. The best-known polynomial-time approximation algorithm for *Max Coverage* and the "dual" *Set Cover* problem[1], is a greedy approach. For Max Coverage, the greedy algorithm has been shown to return a solution whose coverage is at least a $1 - 1/e$ approximation of the optimal solution. This is known to be asymptotically optimal [11].

In practice, the greedy algorithm is much more effective than its theoretical guarantee would suggest, and typically produces a near-optimal solution on realistic inputs [14]. However, the greedy algorithm does not scale well with the size of the input. In the last 15 years, there has been increasing interest in efficient implementation of greedy and greedy-like approaches for Set Cover and Maximum Coverage [7, 10, 20]. In the streaming setting, there have been several innovative algorithms, as detailed below in Table 1. We focus in this paper on engineering the only sublinear-space Set Streaming algorithm [17] so that it runs much faster, and sacrifices no space.

In the Set Streaming model [20], the input stream, $\mathcal{S}$, comprises a sequence of the sets in $\mathcal{F}$, i.e., $\mathcal{S} = S_1, S_2, \ldots, S_m$. Each set S_i in $\mathcal{S}$ appears in full before the next set, S_{i+1}, appears. The design of a streaming algorithm trades off memory, throughput, query/solution time, and solution quality. Let I denote the indexes of the sets in the solution (so far). The coverage of (the sets in) I is $C = \cup_{i \in I} S_i$. Given I, and hence C, the *contribution* of each set S_j, for every $j \notin I$ is $S_j \setminus C$. In the greedy algorithm, we add a set to the solution whenever it has largest contribution, breaking ties arbitrarily. Additionally, another well-studied variant of this streaming model is *random set arrivals*, a reasonable assumption for many applications, and it makes the problem *easier*. Many results regarding trade-off between space complexity and approximation factor improve upon the classic *set arrival* setting [1, 18]. Another common model for Maximum Coverage, although not discussed in this paper, is the Edge-arrival Streaming model. Here the stream consists of pairs $(i, x) \in [m] \times \mathcal{U}$ to indicates that $x \in S_i$. In this more general context, Indyk and Vakilian [15] showed a space lower bound $\Omega(ma^2)$ and upper bound $\tilde{\mathcal{O}}(ma^2)$ for an arbitrary factor a-approximation factor in single pass.

1.1 Sublinear Space

Several of the greedy-like approaches for Set Cover in the Set Streaming model assume $\Omega(n)$ memory is available [7, 10, 16]: at least one bit per item, to record the coverage, and thus determine a set's contribution. Unlike Maximum Coverage, in Set Cover, we expect that the subfamily of sets returned, indexed by I, covers all of $\mathcal{U}$, so $n = |\mathcal{U}|$ bits seem necessary. In contrast, for Max Coverage, the minimum space requirement seems depend on m. For example, it is known that every one-pass $(1/2 + \varepsilon)$-approximation algorithm must work in $\Omega(\varepsilon m/k^3)$ space [12]. Also, $\Omega(m)$ space is necessary to achieve better approximation factors than $1 - 1/e$ [17]. Regarding $(1 - \varepsilon)$-approximation algorithms, Assadi [2] showed that $\tilde{\Omega}(m/\varepsilon^2)$ space is required. It should be noted that all these lower bounds are tight and several one-pass $\tilde{\mathcal{O}}(\varepsilon^d m)$-space algorithms do exist [4, 17].

[1] In Set Cover, the aim is to return a subfamily of *minimum cardinality* whose union is $\mathcal{U}$.

In this context, one algorithm for Maximum Coverage stands out. McGregor and Vu [17] introduced a family of streaming algorithms for Max Coverage. They describe, in §2.2 of their paper, an approximation algorithm that in $\mathcal{O}(\varepsilon^{-1})$ passes and in $\tilde{\mathcal{O}}(\varepsilon^{-2}k)$ space returns a $(1-1/e-\varepsilon)$-approximate solution[2]. This is the only reasonable approximation algorithm for Max Coverage that runs in $o(\min\{m,n\})$ space. For convenience[3], we name this algorithm MACH_*. Like some of the first streaming/external-memory algorithms for Set Cover, MACH_* takes multiple passes, achieving a near-greedy approach via a sequence of decreasing thresholds for the contribution of a set: further details of thresholding are in §1.4. And to save space, MACH_* has a randomized subsampling component achieved with multi-way independent hash functions. These hash functions are slow to evaluate, and it is this component that we accelerate.

1.2 Motivation

In terms of approximation quality and space complexity, MACH_* is the favored approach for Maximum Coverage. The space complexity of $\tilde{\mathcal{O}}(\varepsilon^{-2}k)$ is only a little more than the space required to store[4] solution I: $\tilde{\Omega}(k)$. However, in MACH_*, McGregor and Vu [17] invoke a γ-independent hash function, where $\gamma = \lceil 2c\varepsilon^{-2}k\log m\rceil$, with c a constant to be discussed in §4.1. At first glance, this seems to slow the algorithm down, as $\Omega(\gamma)$ operations are required for each component of the input. Our experiments (refer to Figure 1 below) confirm that the running time of MACH_* is particularly high compared to other alternatives. Our research motivation is:

> Can we accelerate this space-efficient Max Coverage algorithm, MACH_*, without significantly deteriorating space complexity or solution quality?

One promising direction is to *simplify* the subsampling process. McGregor and Vu show that, with $\gamma = \lceil 2c\varepsilon^{-2}k\log m\rceil$, an approximation factor of $1-1/e-\varepsilon$ is guaranteed with probability at least $1-1/m^{10k}$. In the original version of MACH_*, this γ parameter can easily exceed 10^3. So we would anticipate a thousand-fold reduction in throughput compared to a simpler, if theoretically less guaranteed, *sampling scheme*, such as pairwise independent hashing. Since we are designing a space-efficient algorithm, a pre-computed hash function table is infeasible.

1.3 Our contributions

Firstly, we show that the same space complexity and approximation quality can be achieved with $\Theta(k\log m)$ independence (Corollary 5) instead of the original $\Theta(\varepsilon^{-2}k\log m)$ and in fact without invoking F_0-sketching (Lemma 6). Removing F_0-sketching slightly reduces the probability of producing a $1-1/e-\varepsilon$ approximation from $1-e/m^{ck/6}$ to $1-2e/m^{ck/6}$.

---

[2] In this paper, the $\tilde{\mathcal{O}}(\cdot)$ notation hides polylogarithmic factors in m and n.

[3] MACH represents "Maximum Andrew Coverage Hoa": the $*$ represents their parameter choices, which we generalize in this paper.

[4] An approach that avoids storing at least one bit per index in I, as working space, is in principle possible. For example, I could be a size-k subset of $\{1,\ldots,m\}$ chosen uniformly at random; this is not an effective solution, but a valid one, generated in $\tilde{\mathcal{O}}(1)$ working space.

Secondly, if a weaker probabilistic guarantee on the approximation quality is allowed, we show that the algorithm still works with only pairwise independence (Proposition 8). This leads to a significant speed-up, from 10× to more than 1000× for $k \geq 100$ (Figure 1), while maintaining the same space complexity[5] as the original algorithm of McGregor and Vu [17].

Finally, our experimental results demonstrate the efficiency and quality of our generalized algorithm, MACH'_γ. In particular, reducing the independence factor does not lead to significantly worse solutions. We show that for reasonable values (< 0.27) of ε, our algorithm returns consistently better solutions than comparator streaming algorithms (Figure 3).

1.4 Related Work

Thresholding. Before surveying the algorithms for Max Coverage, we set out one of the important algorithmic frameworks. Several algorithms invoke a thresholding technique, first applied to Set Cover by Cormode et al. [10]. It *relaxes* the notion of greedy algorithm, and calculates a near-greedy solution. Instead of *searching* for the set whose contribution is $R^* = \max_j |S_j \setminus C|$, a thresholding algorithm might add a set S_i to the solution if its contribution is at least αR^*, where $\alpha \in [0, 1]$ describes the *greediness* of the thresholding algorithm. Applying this principle repeatedly results in a solution whose coverage is $\alpha(1-1/e)$ fraction of the optimum coverage.

Now the guarantee of αR^* contribution arises from a multi-pass approach to the stream. In pass j, all sets with contribution at least r are added, then in pass $j+1$, all sets with contribution at least αr are added. Since a set's contribution can only decrease as (other) sets are added to I, with this approach, we only add a set if its contribution is αR^*.

Prior art. There are several existing streaming algorithms for the Max Coverage problem, which we summarize in Table 1. Badanidiyuru et al. [3] presented a generic algorithm for maximizing submodular functions on a stream, which can be adapted to Max Coverage. This is a one-pass thresholding algorithm, somewhat similar to MACH_*, that *guesses* the optimal coverage size. Yu and Yuan [22] developed an algorithm that creates a specific ordering $(\tilde{S}_1, \ldots, \tilde{S}_m)$ of the entire collection of sets $\{S_1, \ldots, S_m\}$ such that for all k, $(\tilde{S}_1, \ldots, \tilde{S}_k)$ is a solution of the *Maximum-k-Coverage*. Saha and Getoor [20], who pioneered set streaming, took a *swapping* approach. A putative solution of k sets is stored, and sets in the putative solution can be replaced by new sets in the stream depending on the number of items uniquely covered by sets in the putative solution. More recently, Bateni et al. [4] used a sketching technique and they almost match the optimal approximation factor of $1 - 1/e$. This is an algorithm designed for the edge-arrival streaming model, but can be adapted to the set streaming model with a space complexity independent to the size of the universe. Norouzi-Fard et al. [18], in the continuation of Badanidiyuru et al. [3], presented a 2-pass and a multi-pass approach to maximize a submodular function on a stream. Developed at a similar time, McGregor and Vu [17] presented two polynomial-time algorithms that achieve the same approximation factor of $1 - 1/e - \varepsilon$: one taking a single pass, the other, MACH_*, taking multiple passes. The algorithms developed by McGregor and Vu [17] are *thresholding* algorithms.

[5] Actually, removing F_0-sketching and reducing the independence factor strictly reduces the space complexity, although not asymptotically.

Table 1 Streaming algorithms for Maximum Coverage. We focus on the $o(\min\{m,n\})$-space algorithm, MACH_*.

Author	Name	Passes	Space	Approx.
Badanidiyuru et al. [3]	BMKK	1	$\tilde{\mathcal{O}}(\varepsilon^{-1}n)$	$1/2-\varepsilon$
Yu and Yuan [22]		1	$\tilde{\mathcal{O}}(n)$	~ 0.3
Saha and Getoor [20]	SG	1	$\tilde{\mathcal{O}}(kn)$	$1/4$
Bateni et al. [4]		1	$\tilde{\mathcal{O}}(\varepsilon^{-3}m)$	$1-1/e-\varepsilon$
Norouzi-Fard et al. [18]	2P	2	$\tilde{\mathcal{O}}(\varepsilon^{-1}n)$	$5/9-\varepsilon$
Norouzi-Fard et al. [18]		$\mathcal{O}(\varepsilon^{-1})$	$\tilde{\mathcal{O}}(\varepsilon^{-1}n)$	$1-1/e-\varepsilon$
McGregor and Vu [17]	OP	1	$\tilde{\mathcal{O}}(\varepsilon^{-2}m)$	$1-1/e-\varepsilon$
McGregor and Vu [17]	MACH_*	$\mathcal{O}(\varepsilon^{-1})$	$\tilde{\mathcal{O}}(\varepsilon^{-2}k)$	$1-1/e-\varepsilon$

Sampling. Sampling via hashing is a key component of many streaming algorithms. Relaxing the independence requirement for hash functions was explored in the context of ℓ_0-samplers: Cormode and Firmani [9] invoked γ-independent hash functions. They showed theoretical bounds on γ to guarantee the probability of sampling a non-zero coordinate. In addition, their experimental results suggest that *constant*-independence hashing schemes produce similar successful sampling rate to *linear*-independent hash functions, while being significantly more efficient to compute.

Furthermore, some theoretical results [19] show that many strong guarantees generally associated with *high*-independence families of hash functions can be achieved with simpler hashing schemes. Tabulation hashing [19], for example, is not even 4-independent, but manages to implement γ-independent hash function based algorithms, such as *Cuckoo Hashing*. Pătraşcu and Thorup [19] also prove Chernoff-type inequalities with relaxed assumptions on the independence of the random variables.

2 Tools

The Introduction includes most of our notation; in addition, we let I_{OPT} be an optimal solution and OPT the size of the optimal coverage $\left|\bigcup_{i\in I_{\mathrm{OPT}}} S_i\right|$.

2.1 Subsampling

▶ **Definition 1** (subsampling). *Given $\mathcal{F}, \mathcal{U}$, and hash function $h:\mathcal{U}\to\{0,1\}$, the subsampled universe is $\mathcal{U}'=\{x\in\mathcal{U}\mid h(x)=1\}$, with subsampled sets $S'=S\cap\mathcal{U}'$ for every $S\in\mathcal{F}$.*

Instead of computing with respect to universe $\mathcal{U}$, algorithm MACH_* focuses on $\mathcal{U}'\subset\mathcal{U}$, and tracks only the subsampled coverage $C'=\cup_{i\in I}S'_i$. The size of the optimal coverage of $\mathcal{U}'$, by a subfamily of k sets from $\mathcal{F}$, is henceforth called OPT'.

▶ Remark. The value $\mathrm{OPT}'=\max_{|J|=k}\left|\bigcup_{i\in J} S'_i\right|$ is not necessarily the same as the size of the union of the subsampled sets in the optimal coverage of $\mathcal{U}$, i.e., $\left|\bigcup_{i\in I_{\mathrm{OPT}}} S'_i\right|$.

▶ **Definition 2.** *Let $\gamma, v, p\in\mathbb{N}$ such that $p>|\mathcal{U}|$:*

$$\mathcal{H}_{\gamma,v}=\left\{x\longmapsto\sum_{i=0}^{\gamma-1}a_ix^i \bmod p \mod v \mid 0\le a_i<p\right\}$$

is a family of hash functions $\mathcal{H}_{\gamma,v}\subset\{f:\mathcal{U}\to[v-1]\}$

Such a family has the property of being *γ-independent*[6]. Evaluating a hash function $f \in \mathcal{H}_{\gamma,v}$ takes $\Theta(\gamma)$ operations, including expensive *modulo* operations, but these can be accelerated using the overflow mechanism on unsigned integer types. $\mathcal{H}_{\gamma,v}$ are the families of hash functions used in MACH_*.

2.2 Sketching

To estimate the size of a set, McGregor and Vu invoke F_0 sketching.

▶ **Theorem 3** (F_0-sketching [8]). *Given a stream s, there exists a data structure, $\mathcal{M}(s)$, that requires $\mathcal{O}(\varepsilon^{-2}\log\delta^{-1})$ space and, with probability $1-\delta$, returns the number of distinct elements in s within multiplicative factor $1 \pm \varepsilon$. Processing each new element takes $\mathcal{O}(\varepsilon^{-2}\log\delta^{-1})$ time, the same time as finding the number of distinct elements.*

2.3 Thresholding on the sampled universe

The core of MACH_* is thresholding and subsampling. The solution, I, and the associated subsampled coverage $C' = \cup_{i\in I} S'_i$ are built incrementally, as new sets arrive in the stream and are selected. Given a threshold, r, the selection rule for set S_i is:

$$\text{If } |S'_i \setminus C'| \geq r, \text{ then } I \leftarrow I \cup \{i\} \text{ and } C' \leftarrow C' \cup S'_i\,, \tag{1}$$

where $|S'_i \setminus C'|$ is called the *contribution* of S'_i – from the context, it is clear this is in the sampled universe. In choosing the sequence of thresholds there is a trade-off [7]: the larger the threshold, the higher the solution quality, but the more passes.

3 Low-space Streaming Algorithm

In this section, we describe in detail MACH_* developed by McGregor and Vu [17], which solves Max Cover in sublinear space with a respectable approximation factor. Algorithm MACH_* depends on two variables:

- v, an estimate of the optimal coverage, OPT; and
- λ, an estimate of the optimal coverage on the subsampled universe, OPT′.

These variables determine the probability of subsampling an element, and the initial value of the threshold, r, as applied above (1). The subsampling hash function is implemented as $h(x) = \mathbb{1}_{f(x)<\lambda}$ where $f \in \mathcal{H}_{\lceil 2\lambda\rceil,v}$, hence the probability an item is subsampled is λ/v. The threshold, r, is initially set to $2(1+\varepsilon)\lambda/k$ and after each pass, r decreases by a factor $1+\varepsilon$. McGregor and Vu [17] showed that if

$$\lambda = c\varepsilon^{-2}k\log m\,, \qquad \text{with } c \geq 60\,, \text{ and} \qquad \mathrm{OPT}/2 \leq v \leq \mathrm{OPT}\,, \tag{2}$$

then this thresholding procedure, which we call TP, gives a $1-1/e-\varepsilon$ approximation using $\tilde{\mathcal{O}}(\varepsilon^{-2}k)$ space with probability at least $1-1/m^{10k}$.

3.1 Guessing

Algorithm MACH_* relies on a reasonable estimate of OPT: a v such that $\mathrm{OPT}/2 \leq v \leq \mathrm{OPT}$. Of course, we do not know OPT in advance! The algorithm naively finds the right value for v by executing TP_v for different values of v, called *guesses*, in parallel. Denote by v_g the

[6] Different definitions exist; our definition of γ-independent is stated in the Appendix.

Algorithm 1 Algorithm $\mathsf{MACH}_\gamma(\mathcal{S}, k, \varepsilon, ||S||_\infty)$.

```
begin
1    /* Initialise the guesses */
2    V ← {2^{g−1}||S||_∞ ≤ min(n, k||S||_∞),  g ∈ ℕ}
3    Duplicates each variable |V| times: h, I, C′, 𝓜 and active
4    r ← 2(1 + ε)λ/k
5    /* Multiple passes */
6    for p ← 1 to 1 + ⌈log_{1+ε}(4e)⌉ do
7        /* One pass */
8        for S_i ∈ 𝒮 (stream) do
9            /* Iterate over the guesses */
10           for g ← 0, ..., |V| − 1 do
11               S′_i ← Subsample S_i with h_g
12               R_i ← S′_i \ C′_g /* Contribution */
13               /* Check the bad guess condition */
14               if |C′_g| + |R′_i| > 2(1 + ε)λ then
15                   active_g ← false
16               /* Thresholding procedure */
17               if active_g and |I_g| < k and |R_i| ≥ r then
18                   update 𝓜_g with S_i
19                   C′_g ← C′_g ∪ R_i
20                   I_g ← I_g ∪ {i}
21       /* Update the threshold */
22       r ← r/(1 + ε)
23   /* Find the best coverage among the potentially correct guesses */
24   s ← argmax_{active_g} {|𝓜_g|}
25   return I_s
```

g^{th} guess. To reduce the number of guesses, we assume the maximum set size, which we call $||\mathcal{F}||_\infty$, is known. This assumption requires only one additional pass through the set stream, $\mathcal{S}$: the *asymptotic* number of passes is unchanged. Hence the guesses for v can be restricted to all the values $v_g = 2^{g-2}||\mathcal{S}||_\infty$, with $g \geq 1$, smaller than $k||\mathcal{S}||_\infty$. These *parallel* instantiations increase the running time and space by factor $\log_2 k$: there are separate copies of variables I, C' and h (the subsampling hash function) for each guess: these variables for guess v_g are I_g, C'_g and h_g. Now I, C and C' refer to the variable associated with the output of the algorithm.

Which is the right guess?

This guessing method begs the question: how do we detect the *right* guess? Also, MACH_* is only guaranteed to *work* under the condition $\mathrm{OPT}/2 \leq v \leq \mathrm{OPT}$. Some instances, with a *wrong* guess, might necessitate more space than the bound $\tilde{O}(\varepsilon^{-2}k)$. McGregor and Vu introduce two mechanisms to deal with these questions.

First, the right guess is found by estimating the (non-subsampled) coverage of $\mathcal{U}$associated with each guess: the biggest coverage is considered the right guess. However, only the subsampled coverages, of $\mathcal{U}'$, are calculated. To resolve this, McGregor and Vu adopt F_0-*sketching*, see Theorem 3, which approximates the number of distinct elements in a collection of sets using less space than the collection itself. More particularly, in addition to the subsampled coverage, C'_g, for each v_g, MACH_* maintains a *sketch* $\mathcal{M}_g$ of the coverage in

$\tilde{\mathcal{O}}(\varepsilon^{-2})$ space. Each time a set is selected, $\mathcal{M}_g$ is updated accordingly. Once all the instances TP_g are completed, the sketches $\{\mathcal{M}_g\}$ determine which guess produced the biggest coverage. We still need the subsampled coverages, $\{C'_g\}$, for calculating the set's contributions: $\{S_i \setminus C'_g\}$. The F_0-sketches are too inefficient to be queried that often.

Second, the space complexity never exceeds $\tilde{\mathcal{O}}(\varepsilon^{-2}k)$ due to a consequence of Corollary 9 of McGregor and Vu [17]: if v is the right guess then $\mathrm{OPT}' \leq 2(1+\varepsilon)\lambda$. Thus, if the subsample coverage C'_g of an instance TP_g exceeds $2(1+\varepsilon)\lambda$, the associated guess is necessarily wrong and this instance can be terminated (Line 14 in Algorithm 1). Hence every instance runs in space $\mathcal{O}(\lambda) = \tilde{\mathcal{O}}(\varepsilon^{-2}k)$. Thus, λ can be referred as the *space budget* of the algorithm.

3.2 Properties

Algorithm 1 is our generalisation of MACH_*, which we call MACH_γ. The independence factor, γ, is not fixed, but is instead a parameter that influences the implementation of the subsampling hash functions $\{h_g\}$. The original algorithm, MACH_*, has an independence factor of $\lceil 2\lambda \rceil$: **so $\mathsf{MACH}_* = \mathsf{MACH}_{\lceil 2\lambda \rceil}$ in our generalization**.

McGregor and Vu [17] showed that MACH_* has space complexity $\tilde{\mathcal{O}}(\varepsilon^{-2}k)$ and with probability at least $1 - \frac{1}{m^{10k}}$ produces a $1 - 1/e - \varepsilon$ approximation. As our focus is improving run time, with only a small trade-off in the other properties, we first dissect the run time. The coverages C'_g are implemented with hash tables in order to efficiently compute $R_i = S'_i \setminus C'_g$. If the gth guess is reading the ith set then:

Line 11: Subsampling S_i: $\mathcal{O}(\gamma)|S_i|$ time (evaluate degree-$\mathcal{O}(\gamma)$ polynomial for each element in S_i)

Line 12: Computing R_i: $\mathcal{O}(|S'_i|) \subset \mathcal{O}(|S_i|)$ time

Line 19: Updating C'_g: $\mathcal{O}(|R_i|) \subset \mathcal{O}(|S_i|)$ time

Line 20: Updating $\mathcal{M}_g$: $\mathcal{O}(\varepsilon^{-2}\log n)|S_i|$ time (Theorem 3)

Therefore, the expected time complexity, T_γ, of MACH_γ is

$$
\begin{aligned}
T_\gamma &= \underbrace{\mathcal{O}(\log k)}_{\text{guesses}} \left(\underbrace{\mathcal{O}(\varepsilon^{-1})}_{\text{passes}} \cdot \underbrace{\mathcal{O}\left(\sum_{i=1}^{m} \gamma |S_i| \right)}_{\text{subsampling}} + \underbrace{\mathcal{O}\left(\sum_{i \in I} \varepsilon^{-2} |S_i| \log n \right)}_{F_0\text{-sketching}} \right) \\
&= \mathcal{O}\left(\varepsilon^{-1}\gamma m |\overline{\mathcal{S}}| \log k + \varepsilon^{-2} k |\overline{\mathcal{C}}| \log n \log k \right) \qquad (3)
\end{aligned}
$$

▶ Note. $|\overline{\mathcal{S}}| = \frac{1}{m}\sum_i |S_i|$ is the average set size over the entire stream, $\mathcal{S}$, while $|\overline{\mathcal{C}}| = \frac{1}{k}\sum_{i \in I} |S_i|$ is the average set size over the selected sets (in I).

Therefore, $\mathsf{MACH}_* = \mathsf{MACH}_{\lceil 2\lambda \rceil}$ has expected time complexity of

$$
T_{\lceil 2\lambda \rceil} = \mathcal{O}\left(\varepsilon^{-3} k m |\overline{\mathcal{S}}| \log m \log k + \varepsilon^{-2} k |\overline{\mathcal{C}}| \log n \log k \right) . \qquad (4)
$$

Regarding the space complexity for $\gamma < \lceil 2\lambda \rceil$, it remains (asymptotically) the same. Indeed the cost for storing a γ-independent hash function is $\mathcal{O}(\gamma) \subset \tilde{\mathcal{O}}(\varepsilon^{-2}k)$.

Interestingly, MACH_γ does not guarantee the solution returned actually has k sets. Given that $|I| \leq k$, we can simply append $k - |I|$ random indices to the returned solution. However, the goal of this paper is to assess the probabilistic nature of the algorithm that arises from the γ-independent hash functions. Therefore, in our experiments in §5, we do not alter the returned solution given by MACH_γ.

4 Accelerating the Algorithm

Algorithm MACH_γ is a breakthrough: it runs in sublinear space. As we surmise from the expression (3) for T_γ, however, the high independence factor, γ, induces a significant bottleneck. Our experiments in §5 validate this conjecture. In this section we demonstrate how to improve the running time without too much sacrifice in the other properties. Indeed, when lowering the independence factor, γ, we maintain the space complexity and the number of passes.

McGregor and Vu [17] showed that if $\mathrm{OPT}/2 \le v$ and $\big|\,|C'| - p|C|\,\big| < \varepsilon vp$ then the solution produced by MACH_γ is a $1-1/e-\varepsilon$ approximation. Assuming condition $\mathrm{OPT}/2 \le v \le \mathrm{OPT}$ is met, i.e., we have the right guess for v, $\mathbb{P}(E_I)$ is a lower bound on the probability of producing a $1-1/e-\varepsilon$ approximation, where E_I is the event $\big\{\,\big|\,|C'| - p|C|\,\big| < \varepsilon vp\,\big\}$. Therefore, the goal is to estimate $\mathbb{P}(E_I)$ for independence factor γ smaller than $\lceil 2\lambda \rceil$.

We provide proofs of several of the following results in the Appendix.

4.1 Maintaining the Approximation Property

Recalculating *c*. The first optimisation is actually an observation about the constant c in the definition of $\lambda = c\varepsilon^{-2}k\log m$. Indeed, the independence factor $\gamma = \lceil 2\lambda \rceil$ depends on c: the authors state "Let c be some sufficiently large constant." At first glance, it seems that to get $\mathbb{P}(\overline{E_I})$, an upper bound on the failure probability, below e/m^{10k}, the constant c must be greater than 60 because the inequality in their Lemma 8 is $\mathbb{P}(\overline{E_I}) \le e/m^{ck/6}$. Indeed, with a bound on $\mathbb{P}(\overline{E_I})$, we apply a union bound to upper bound $\mathbb{P}\left(\cup_{|I|=k}\overline{E_I}\right)$. Unpacking this, we find the inequalities

$$\mathbb{P}\left(\bigcup_{|I|=k} \overline{E_I}\right) \le \sum_{|I|=k} \mathbb{P}(\overline{E_I}) \le \binom{m}{k}\frac{e}{m^{ck/6}} \le \frac{e}{k!\; m^{(c/6-1)k}},$$

whence we conclude that $c = 6$ is the smallest reasonable value to be sure the upper bound is $o(1)$. This reduction in c does not reduce the asymptotic time complexity of MACH_γ, but in practice it reduces the independence factor by a factor 10 so this is still a $10\times$ speed up compared to the original $c \ge 60$.

Reducing γ. In order to reduce further the independence factor γ, we express $\mathbb{P}(E_I)$ with respect to γ. To that end, we use the following concentration bound, from the same paper cited by McGregor and Vu.

▶ **Theorem 4** (Schmidt et al. [21]). *Let $X_1,\dots,X_n$ be γ-wise independent r.v.s, $X = \sum_{i=1}^n X_i$ and $\mu = \mathbb{E}(X)$. If $X_i \in [0,1]$ and $\gamma \le \lfloor \min(\delta,\delta^2)\mu e^{-1/3} \rfloor$ then $\mathbb{P}(|X-\mu| \ge \delta\mu) \le e^{-\lfloor \gamma/2 \rfloor}$.*

Notice that $|C'| = \sum_{i=1}^n X_i$ where $X_i = \mathbb{1}_{i\in C}\mathbb{1}_{h(i)=1} \in [0,1]$; since $p = \lambda/v$ is the probability of subsampling an element, $\mathbb{P}(h(i)=1)$, we have the following corollary:

▶ **Corollary 5.** *If $\gamma \le \lfloor \frac{c}{3}k\log m \rfloor$, with I, C, and C' defined accordingly, then:*

$$\mathbb{P}\left(\,\big|\,|C'| - p|C|\,\big| \ge \varepsilon vp\,\right) \le e^{-\lfloor \gamma/2 \rfloor}.$$

Consequently, by setting $\gamma = \lfloor \frac{c}{3}k\log m \rfloor = \mathcal{O}(\varepsilon^2\lambda)$, compared to the original $\mathcal{O}(\lambda)$, we keep the same approximation guarantees as McGregor and Vu [17]:

$$\mathbb{P}(\overline{E_I}) \le e^{-\left\lfloor \lfloor \frac{c}{3}k\log m \rfloor/2 \right\rfloor} = e^{-\left\lfloor \frac{c}{6}k\log m \right\rfloor} \le e^{-\frac{c}{6}k\log m+1} = \frac{e}{m^{ck/6}}.$$

Algorithm 2 Procedure `FindGuess`.

$s \leftarrow |V| - 1$
while $|C'_s| < (1-\varepsilon)(1-1/e-\varepsilon)\lambda$ ***or*** $\neg \mathtt{active}_s$ **do**
  $s \leftarrow s - 1$

Removing F_0-sketching. First, it should be noted that McGregor and Vu showed that MACH_* produces a $1 - 1/e - \delta(\varepsilon)$ approximation of the optimal coverage where $\delta(\varepsilon) = \varepsilon(3 - 1/e - \varepsilon) \leq 2.6\varepsilon$. Asymptotically, the statement of McGregor and Vu is right because the algorithm can simply start by dividing ε by 3 and it would indeed produce a $1 - 1/e - \varepsilon$ approximation. Nevertheless, such a modification would result in a significant slowdown ($\times 3$ to $\times 27$ depending on the independence factor). In the §5 experiments, we assess the approximation quality relatively to the actual theoretical bound $1 - 1/e - \delta(\varepsilon)$. Finally, thanks to the following result, we adapt MACH_γ so that F_0-sketching is not needed.

Let MACH'_γ be the algorithm that replaces line 24 in Algorithm 1 with Algorithm 2. The selected guess is the biggest active guess, s, such that $|C'_s| \geq (1-\varepsilon)(1-1/e-\varepsilon)\lambda$. We thus conclude MACH'_γ is correct for $\gamma \geq \lfloor \frac{c}{3} k \log m \rfloor$.

▶ **Lemma 6.** *Let v be some guess in MACH'_γ and let C' be the final subsampled coverage associated with guess v. If*

$$\big|\, |C'| - p|C| \,\big| < \varepsilon v p; \text{ and } v > OPT; \text{ and } (1-\varepsilon)(1-1/e-\varepsilon)\lambda \leq |C'|\,,$$

then $|C| > (1 - 1/e - \delta(\varepsilon))\,OPT$.

▶ **Proposition 7.** *For $\gamma \geq \lfloor \frac{c}{3} k \log m \rfloor$, MACH'_γ finds a $1 - 1/e - \delta(\varepsilon)$ approximation of the Maximum-k-Coverage problem with probability at least $1 - 2e/m^{ck/6}$.*

Since the F_0-sketch is omitted, the time complexity is $T'_\gamma = \mathcal{O}(\varepsilon^{-1}\gamma m|\overline{\mathcal{S}}|\log k)$, while the space complexity is unchanged. With $\gamma = \lfloor \frac{c}{3} k \log m \rfloor$, MACH'_γ has a time complexity of $T'_{\lfloor \varepsilon^2\lambda/3 \rfloor} = \mathcal{O}(\varepsilon^{-1} k m |\overline{\mathcal{S}}| \log k \log m)$, which is at least ε^{-2} faster than expression (4).

4.2 Pairwise Independence

We now consider the smallest independence factor possible, $\gamma = 2$.

▶ **Proposition 8.** *Let h be a 2-independent hash function, and I, C, C', be defined accordingly. We have $\mathbb{P}\left(\big|\, |C'| - p|C| \,\big| \geq \varepsilon v p \right) \leq 2/(ck \log m)$.*

Substituting the bound of Proposition 8 into Proposition 7, we conclude:

▶ **Corollary 9.** *With probability at least $1 - 4/(ck\log m)$, MACH'_2 returns a $1 - 1/e - \delta(\varepsilon)$ approximation to the Maximum-k-Coverage problem.*

The decrease in probabilistic guarantee is compensated by a significant speed-up. The time complexity of MACH'_2 is $T'_2 = \mathcal{O}(\varepsilon^{-1} m |\overline{\mathcal{S}}| \log k)$, which grows only logarithmically in k.

5 Experimentation

In this section, we assess the performance of the MACH'_γ algorithm family on real-world datasets. We focus on four datasets, summarized in Table 2:

- *SocialNet*[7] represents a collection of individuals linked by a friendship relation.
- *UKUnion* [5] combines snapshots of webpages in the .uk domain taken over a 12-month period between June 2006 and May 2007.
- *Webbase* [6] and *Webdocs* [13] each represent a collection of interlinked websites.

Table 2 Real-world datasets. Hapax Legomena (HL) refers to the number of sets that contains an element which appears only in this set. The minimum set size and element frequency is 1.

Dataset	n $\times 10^6$	m $\times 10^6$	Set size Max	Med	Avg	Element frequency Max	Med	Avg	HL
SocialNet	65.0	37.6	3,615	12	48.10	4,223	6	27.80	24.0%
UKUnion	126.5	74.1	22,429	25	45.56	4,714,511	2	26.71	16.7%
Webbase	112.2	57.0	3,841	6	11.81	618,957	2	6.00	23.5%
Webdocs	5.3	1.7	71,472	98	177.20	1,429,525	1	56.93	21.2%

MACH'_γ is implemented[8] in C++20 and executed on *Spartan* the high performance computing system of The University of Melbourne. The CPU model is the Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz with a maximum frequency of 4GHz. MACH'_γ naturally implies a parallel algorithm that consists of performing the computation related to each guess in parallel. Nonetheless, we do not implement an actual parallel algorithm as it would require substantial effort in order to fine tune. Also, compared to the original algorithm, this approach does not change the number of guesses. Therefore, the potential speed-up of a parallel implementation would be the same for our algorithm MACH'_γ and the original algorithm MACH_*.

Assessing coverage. With original independence factor $\gamma = \lceil 2\lambda \rceil$, MACH'_γ can still take tens of hours on the biggest datasets. We thus introduce a new variant of the algorithm, the *full sampling* variant, $\mathsf{MACH}'_{\text{fs}}$. *Full sampling* means there is no subsampling so $\mathsf{MACH}'_{\text{fs}}$ is a deterministic algorithm where $\mathbb{P}(E_I) = 1$. It means that $\mathsf{MACH}'_{\text{fs}}$ is fast and produces particularly good solutions (Figure 3). However, it has a space complexity of $\tilde{\mathcal{O}}(n)$ so $\mathsf{MACH}'_{\text{fs}}$ is just seen as a tool to assess the approximation quality of MACH'_2.

Setting c. To bound the failure probability of $\mathsf{MACH}'_{\lceil 2\lambda \rceil}$ and $\mathsf{MACH}'_{\lfloor \varepsilon^2\lambda/3 \rfloor}$, we set $c \leftarrow 6$. With $c < 6$, $\mathsf{MACH}'_{\lfloor \varepsilon^2\lambda/3 \rfloor}$ would be even more space efficient, while maintaining a high probability of success: $\mathbb{P}(E_I) \geq 1 - e/m^{ck/3}$, for $c \geq 1$, where m is expected to exceed several million. We therefore run $\mathsf{MACH}'_{\text{fs}}$ and MACH'_γ with $c = 1$.

Suite of experiments. Algorithms $\mathsf{MACH}'_{\text{fs}}$, $\mathsf{MACH}'_{\lceil 2\lambda \rceil}$, $\mathsf{MACH}'_{\lfloor \varepsilon^2\lambda/3 \rfloor}$ and MACH'_2 are executed on the four datasets, for $\varepsilon \in \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}\}$ and $k \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$. Although $\varepsilon = 1/2$ is out of the theoretical consideration, because $1 - 1/e - \delta(1/2) < 0$, it presents an opportunity to observe how the algorithm behaves outside its theoretical scope: $\varepsilon < 0.267$. For each Figure, in the main text, we only show the datasets representative of the variety of behaviors. The remaining components of each figure are in the Appendix.

[7] https://snap.stanford.edu/data/com-Friendster.html
[8] https://github.com/caesiumCode/streaming-maximum-cover

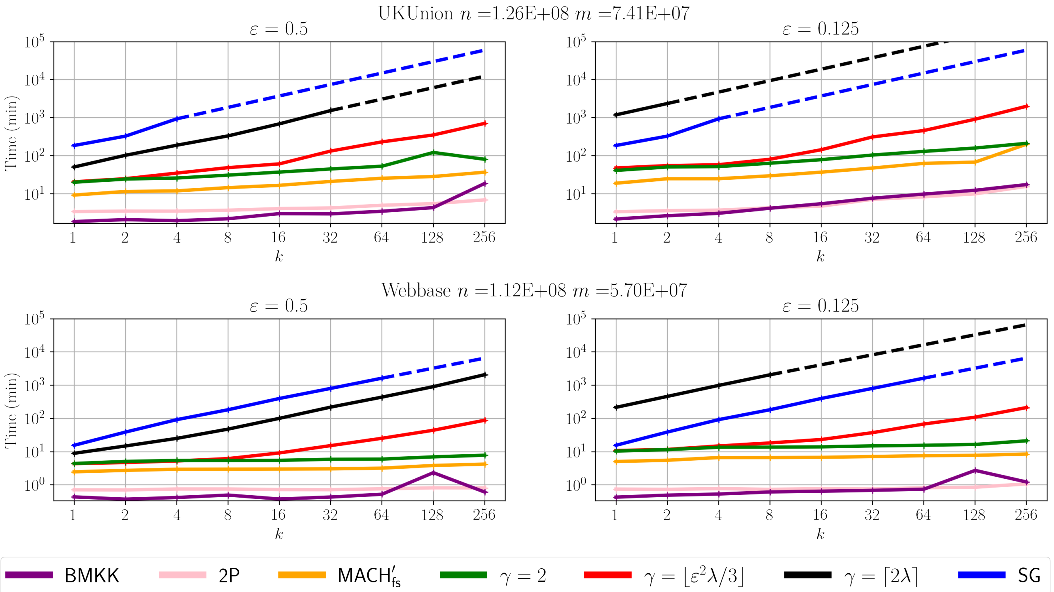


Figure 1 Running times of the algorithms, demonstrated on *Webbase*. Observe that for $\varepsilon = 1/8$, sticking with $\gamma = \lceil 2\lambda \rceil$ leads to a particularly slow algorithm. On the other hand, up to $k = 8$, both $\gamma = \lceil \varepsilon^2\lambda/3 \rceil$ and $\gamma = 2$ are only 2–3 times slower than full sampling. For larger values of k, $\gamma = \lceil \varepsilon^2\lambda/3 \rceil$ becomes 8–10 times slower. The missing values (dashed line) for $\gamma = \lceil 2\lambda \rceil$ and SG are extrapolated, they refer to a running time that exceeds the time limit of 48 hours.

Comparator algorithms (refer to Table 1). The goal is to assess the trade-off between time, space and approximation quality, therefore we aim to compare MACH'_γ with algorithms that perform relatively well in all three categories. For that reason, we implemented the comparator algorithms SG, BMKK, and 2P. The algorithm of Yu and Yuan [22] takes too much time and space, as it solves for all possible values of k at once. Also, as illustrated and explained in the Appendix (Figure 7), the $\tilde{O}(\varepsilon^{-d}m)$-space algorithms consume too much space in comparison to the $\tilde{O}(\varepsilon^{-d}n)$ space algorithms. We run Algorithm 2P instead of its $(1 - 1/e - \varepsilon)$-approximation cousin [18]: the latter is less effective than 2P, empirically, while consuming the same space and taking more passes.

5.1 Runtime Evaluation

Figure 1 demonstrates the time saved by reducing the independence factor. $\mathsf{MACH}'_{\lfloor \varepsilon^2\lambda/3 \rfloor}$ is consistently faster than $\mathsf{MACH}'_{\lceil 2\lambda \rceil}$ by an order of magnitude, while, as k increases, MACH'_2 widens its gap over $\mathsf{MACH}'_{\lfloor \varepsilon^2\lambda/3 \rfloor}$. In contrast to $\mathsf{MACH}'_{\mathrm{fs}}$, the time spent calculating hash function outputs for subsampling is clear. About half the running time of MACH'_2 is about subsampling, while this proportion easily exceeds 99% of the running time for $\mathsf{MACH}'_{\lceil 2\lambda \rceil}$. Considering comparators, BMKK and 2P are equally the fastest algorithms by a wide margin. Despite only one pass, SG is one of the slowest algorithms, along with $\mathsf{MACH}'_{\lceil 2\lambda \rceil}$. The precise running times can be consulted in Table 3.

5.2 Space Efficiency

To measure the space complexity of the different algorithms, we simply count the number of element instances stored by each algorithm. Figure 2 demonstrates how space efficient MACH'_γ is compared with alternatives, as predicted by the sublinear asymptotic bound: $\tilde{O}(\varepsilon^{-2}k)$, seemingly independent of the coverage size, in practice as well as theory.

Table 3 Summary of the running times in minutes of the algorithms (Figure 1). An empty cell means the time exceeds 48 hours (2880 minutes).

Dataset	*Webbase*				*UKUnion*			
k	4	16	64	256	4	16	64	256
$\varepsilon = 0.5$								
$\mathsf{MACH}'_{\mathrm{fs}}$	4.5	4.4	4.7	9.2	17.1	22.3	38.1	59.7
MACH'_2	8.5	8.8	9.3	10.8	35.7	48.3	77.2	134.5
$\mathsf{MACH}'_{\lfloor \varepsilon^2\lambda/3 \rfloor}$	8.0	14.5	38.3	135.9	40.8	90.3	338.3	1145.3
$\mathsf{MACH}'_{\lceil 2\lambda \rceil}$	143.4	588.6	2609.1	–	849.0	–	–	–
SG	92.1	399.3	1632.4	–	936.7	–	–	–
BMKK	0.5	0.5	0.6	0.8	2.9	3.8	5.5	11.6
2P	0.8	0.8	0.8	0.8	3.6	4.4	6.6	8.8
$\varepsilon = 0.125$								
$\mathsf{MACH}'_{\mathrm{fs}}$	6.7	6.8	7.6	8.5	24.9	37.1	63.3	201.3
MACH'_2	13.6	13.9	15.6	21.3	52.0	79.0	130.6	212.4
$\mathsf{MACH}'_{\lfloor \varepsilon^2\lambda/3 \rfloor}$	14.9	23.2	68.0	212.7	58.3	144.3	461.3	2002.5
$\mathsf{MACH}'_{\lceil 2\lambda \rceil}$	987.4	–	–	–	–	–	–	–
SG	92.1	399.3	1632.4	–	936.7	–	–	–
BMKK	0.5	0.6	0.7	1.2	3.1	5.5	9.9	17.5
2P	0.8	0.8	0.8	1.1	3.7	4.7	8.3	15.7

As stated earlier, the space complexity of $\mathsf{MACH}'_{\mathrm{fs}}$, SG and BMKK scales linearly with the coverage size of the solution. So when MACH'_γ does not look so advantageous for *SocialNet* when $\varepsilon = 0.125$, it is simply because the coverage is almost as small as the space budget of MACH'_γ. The coverage of *UKUnion* is about 10 times bigger than *SocialNet*, but the space consumption is about the same as *SocialNet*.

5.3 Estimating Approximation Quality

The maximum set coverage problem is NP-Hard. Comparing the coverage size produced by MACH'_γ with the optimal solution is infeasible at the scale of our datasets. Since the greedy algorithm guarantees a $1 - 1/e$ approximation, and can be implemented, its coverage is our *reference*. Moreover, even if the optimal solution, OPT, is unknown, the $1 - 1/e - \delta(\varepsilon)$ approximation of MACH'_γ can be verified using the greedy algorithm thanks to the following implication: $|C|/|G| \geq \beta \implies |C|/\mathrm{OPT} \geq \beta(1-1/e)$, where G returned by greedy and C an arbitrary coverage. In particular, if $|C|/|G| \geq 1-\delta(\varepsilon)/(1-1/e)$ then $|C|/\mathrm{OPT} \geq 1-1/e-\delta(\varepsilon)$.

Figure 3 demonstrates that for theoretically *admissible* values of ε, MACH'_γ produces a coverage close to greedy coverage, and always within the $1 - \delta(\varepsilon)/(1 - 1/e)$ limit. Regardless of γ value, it remains very close to the coverage produced by $\mathsf{MACH}'_{\mathrm{fs}}$. Even for pairwise independence, which is expected to produce worse solutions, there is no clear performance effectiveness difference compared higher independence. Additionally, even though MACH'_γ is a randomized algorithm, no coverage has been observed to beat greedy.

We observe some rare events ($< 1\%$) where the coverage is particularly small compared to $\mathsf{MACH}'_{\mathrm{fs}}$. Investigating these cases reveals that such solutions typically contain fewer than k sets. If MACH'_γ does not select the right guess, it tends to select a guess slightly bigger than the right one, which increases the threshold, therefore it does not have enough opportunity to select k sets in $\mathcal{O}(\varepsilon^{-1})$ passes.

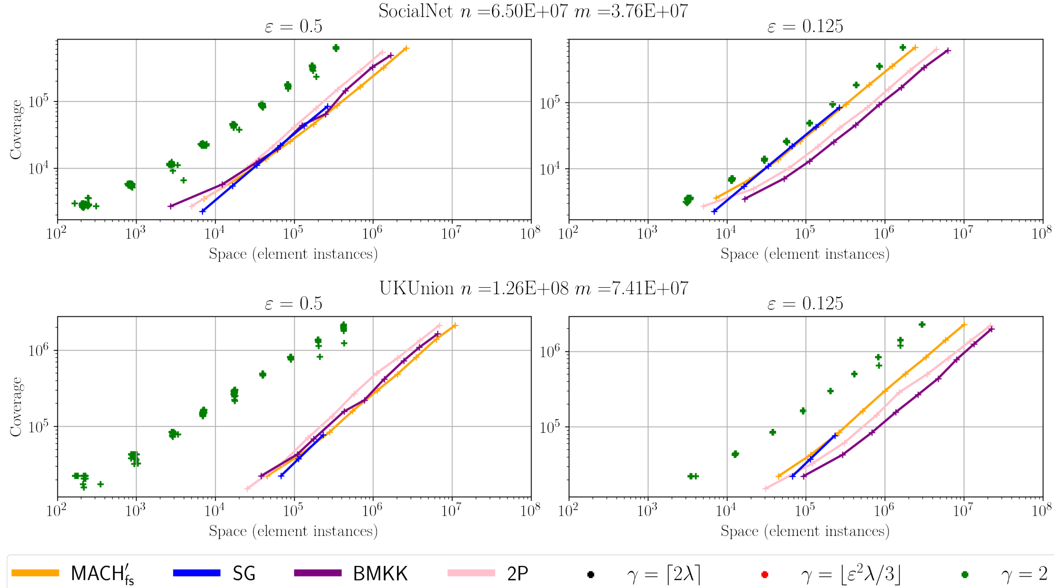


Figure 2 Coverage versus space, for $k \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$. Every panel shows the advantage of the MACH′ family for max coverage in streams. For $\varepsilon = 1/2$, the space advantage is at least ten-fold. Interestingly, as ε decreases, the space advantage drops for some datasets, but the coverage does not improve significantly, suggesting that a *lightweight* MACH′ approach, i.e., smaller ε^{-1}, might be the most effective time-space-performance trade-off.

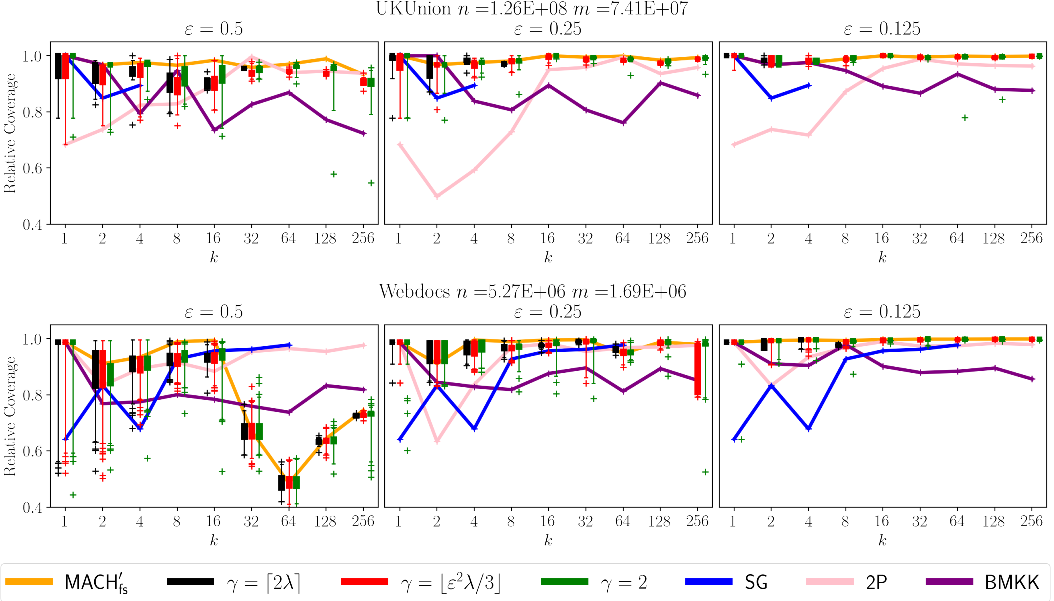


Figure 3 Coverage of the algorithms relative to greedy coverage. Since they are randomized, there are box plots of coverage produced by MACH'_γ. The box plots show the 1%, 25%, 75% and 99% quantiles, hence the points below and above the box plots are in the first and last 1%. For $\gamma = 2$ and $\gamma = \lfloor \varepsilon^2\lambda/3 \rfloor$, each boxplot gathers 200 data points on average whereas for $\gamma = \lceil 2\lambda \rceil$, each boxplot gathers 90 data points on average. Observe that the MACH′ methods return excellent coverage except for values of k around 64 on the dataset *Webdocs* when $\varepsilon = 1/2$.

6 Conclusions

In this paper, we accelerate the sublinear-space approach to solving Maximum Coverage. The algorithm MACH_* of McGregor and Vu is hampered by a high-independence hash function. We generalize their approach to produce MACH_γ, so that $\mathsf{MACH}_* = \mathsf{MACH}_{\lceil 2\lambda \rceil}$ and then avoid F_0-sketches to obtain MACH'_γ. The space consumption is in $\tilde{\mathcal{O}}(\varepsilon^{-2}k)$ and the approximation factor is $1 - 1/e - \delta(\varepsilon)$.

For reasonable values of ε (≤ 0.25), our algorithm, MACH'_γ, maintains the space efficiency and approximation quality of $\mathsf{MACH}_* = \mathsf{MACH}_{\lceil 2\lambda \rceil}$. In experiments, it is several orders of magnitude faster. In practice, we find MACH'_2 presents the best trade-off between space complexity, time complexity and approximation quality. Since MACH'_2 is so efficient, we can run it several times with fresh randomness. This approach is more effective than executing MACH'_γ with a high independence factor. Although we avoided F_0-sketching in MACH'_γ, they could help compare independent instances of the fast MACH'_2.

We obtained several key results by carefully analyzing upper bounds on algorithm failure probability. We expect this idea accelerates other lower-space streaming algorithms.

References

1. Shipra Agrawal, Mohammad Shadravan, and Cliff Stein. Submodular secretary problem with shortlists. In *10th ITCS*, pages 1:1–1:19, 2018. doi:10.4230/LIPIcs.ITCS.2019.1.
2. Sepehr Assadi. Tight space-approximation tradeoff for the multi-pass streaming set cover problem. In *36th ACM PODS*, pages 321–335, 2017. doi:10.1145/3034786.3056116.
3. Ashwinkumar Badanidiyuru, Baharan Mirzasoleiman, Amin Karbasi, and Andreas Krause. Streaming submodular maximization: massive data summarization on the fly. In *20th ACM SIGKDD*, pages 671–680, August 2014. doi:10.1145/2623330.2623637.
4. MohammadHossein Bateni, Hossein Esfandiari, and Vahab Mirrokni. Almost optimal streaming algorithms for coverage problems. In *29th ACM SPAA*, pages 13–23, 2017. doi:10.1145/3087556.3087585.
5. Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2):33–38, 2008. doi:10.1145/1480506.1480511.
6. Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *13th WWW*, pages 595–601, 2004. doi:10.1145/988672.988752.
7. Amit Chakrabarti and Anthony Wirth. Incidence geometries and the pass complexity of semi-streaming set cover. In *27th ACM-SIAM SODA*, pages 1365–1373, 2016. doi:10.1137/1.9781611974331.ch94.
8. G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using hamming norms (how to zero in). *IEEE Transactions on Knowledge and Data Engineering*, 15(3):529–540, 2003. doi:10.1109/tkde.2003.1198388.
9. Graham Cormode and Donatella Firmani. A unifying framework for ℓ_0-sampling algorithms. *Distributed and Parallel Databases*, 32(3):315–335, 2013. doi:10.1007/s10619-013-7131-9.
10. Graham Cormode, Howard Karloff, and Anthony Wirth. Set cover algorithms for very large datasets. In *19th ACM CIKM*, pages 479–488, 2010. doi:10.1145/1871437.1871501.
11. Uriel Feige. A threshold of $\ln n$ for approximating set cover. *J. ACM*, 45(4):634–652, 1998. doi:10.1145/285055.285059.
12. Moran Feldman, Ashkan Norouzi-Fard, Ola Svensson, and Rico Zenklusen. The one-way communication complexity of submodular maximization with applications to streaming and robustness. In *52nd ACM STOC*, pages 1363–1374, 2020. doi:10.1145/3357713.3384286.
13. Bart Goethals and Mohammed J Zaki. Fimi'03: Workshop on frequent itemset mining implementations. In *3rd IEEE Data Mining Workshop on Frequent Itemset Mining Implementations*, pages 1–13, 2003.

14 Tal Grossman and Avishai Wool. Computational experience with approximation algorithms for the set covering problem. *European Journal of Operational Research*, 101(1):81–92, 1997. doi:10.1016/s0377-2217(96)00161-0.

15 Piotr Indyk and Ali Vakilian. Tight trade-offs for the maximum k-coverage problem in the general streaming model. In *38th ACM PODS*, pages 200–217, 2019. doi:10.1145/3294052.3319691.

16 Ching Lih Lim, Alistair Moffat, and Anthony Wirth. Lazy and eager approaches for the set cover problem. In *37th ACSC*, pages 19–27, 2014. doi:10.5555/2667473.2667476.

17 Andrew McGregor and Hoa T. Vu. Better streaming algorithms for the maximum coverage problem. *Theory of Computing Systems*, 63(7):1595–1619, 2018. doi:10.1007/s00224-018-9878-x.

18 Ashkan Norouzi-Fard, Jakub Tarnawski, Slobodan Mitrovic, Amir Zandieh, Aidasadat Mousavifar, and Ola Svensson. Beyond 1/2-approximation for submodular maximization on massive data streams. In *35th ICML*, pages 3829–3838, 2018. URL: https://proceedings.mlr.press/v80/norouzi-fard18a.html.

19 Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3):1–50, 2012. doi:10.1145/2220357.2220361.

20 Barna Saha and Lise Getoor. On maximum coverage in the streaming model & application to multi-topic blog-watch. In *9th SDM*, pages 697–708, 2009. doi:10.1137/1.9781611972795.60.

21 Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff–Hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995. doi:10.1137/s089548019223872x.

22 Huiwen Yu and Dayu Yuan. Set coverage problems in a one-pass data stream. In *13th SDM*, pages 758–766, 2013. doi:10.1137/1.9781611972832.84.

A Definition of γ-independent hash functions family

▶ **Definition 10.** *The family of hash functions $\mathcal{H} \subset \{f : \mathcal{U} \to Y\}$ is γ-independent iff for every γ distinct keys $x_1, \ldots, x_\gamma \in \mathcal{U}$ and γ values $y_1, \ldots, y_\gamma \in Y$, if we draw f uniformly at random from $\mathcal{H}$, then the $f(x_i)$ are independent uniform random variables, and $\mathbb{P}\left[\cap_{i=1}^{\gamma}(f(x_i) = y_i)\right] = 1/|Y|^\gamma$.*

B Corollary 5 (proof)

▶ **Corollary 5.** *If $\gamma \le \lfloor \frac{c}{3} k \log m \rfloor$, with I, C, and C' defined accordingly, then:*

$$\mathbb{P}\left(\left| \, |C'| - p|C| \, \right| \ge \varepsilon v p \right) \le e^{-\lfloor \gamma/2 \rfloor} .$$

Proof. Consider I to be fixed, so the $\mathbb{1}_{i\in C}$ factor is just a constant, with no randomness.

$$\mathbb{E}(|C'|) = \sum_{i=1}^{n} \mathbb{1}_{i\in C}\mathbb{E}(\mathbb{1}_{h(i)=1}) = \sum_{i\in C} \mathbb{P}(h(i) = 1) = p|C|$$

Let $\delta = \varepsilon v/|C|$ and $\mu = \mathbb{E}(|C'|)$, then $\mathbb{P}\left(\left| \, |C'| - p|C| \, \right| \ge \varepsilon v p \right) = \mathbb{P}\left(\left| \, |C'| - p|C| \, \right| \ge \delta\mu \right)$. Now, recalling the definition (2) of λ, we verify the condition on the independence factor, γ:

$$\begin{aligned} \frac{c}{3} k \log m = \frac{\varepsilon^2}{3}\lambda = \frac{\varepsilon}{3}\delta\mu &\le e^{-1/3}\frac{\varepsilon}{2}\delta\mu && e^{-1/3} > 2/3 \\ &\le e^{-1/3}\min(1,\delta)\delta\mu && \delta \ge \varepsilon/2 \text{ and } \varepsilon/2 \le 1 , \end{aligned}$$

where $|C| \le \mathrm{OPT} \le 2v$ gives us the condition $\delta \ge \varepsilon/2$. The condition $\varepsilon/2 \le 1$ is arbitrary but recall that we want a $1 - 1/e - \varepsilon$ approximation so $\varepsilon < 1 - 1/e \le 0.7$. Therefore, $\gamma \le \lfloor \frac{c}{3} k \log m \rfloor \le \lfloor e^{-1/3} \min(\delta, \delta^2)\mu \rfloor$ and Theorem 4 gives the desired inequality. ◀

C Lemma 6 (proof)

▶ **Lemma 6.** *Let v be a guess in* MACH$_\gamma$ *and let C' be the final subsampled coverage associated with guess v. If*

1. $\big|\,|C'| - p|C|\,\big| < \varepsilon vp$; *and*
2. $v \geq OPT$; *and*
3. $(1-\varepsilon)(1-1/e-\varepsilon)\lambda \leq |C'|$;

then $|C| > (1 - 1/e - \delta(\varepsilon))OPT$.

Proof. Assuming condition 3, we have,

$$\begin{aligned}
|C'| - \varepsilon vp &\geq (1-\varepsilon)(1-1/e-\varepsilon)vp - \varepsilon vp & \\
|C|p &\geq (1-\varepsilon)(1-1/e-\varepsilon)vp - \varepsilon vp & \text{Condition 1}\\
|C| &\geq (1-\varepsilon)(1-1/e-\varepsilon)v - \varepsilon v & p > 0\\
|C| &\geq (1-1/e-\delta(\varepsilon))v & \text{where } \delta(x) = x(3-1/e-x)\\
|C| &\geq (1-1/e-\delta(\varepsilon))\,\mathrm{OPT} & \text{Condition 2.}
\end{aligned}$$ ◀

D Proposition 7 (proof)

▶ **Proposition 7.** *For $\gamma \geq \lfloor \frac{c}{3} k \log m \rfloor$,* MACH$'_\gamma$ *finds a $1 - 1/e - \delta(\varepsilon)$ approximation of the Maximum-k-Coverage problem with probability at least $1 - 2e/m^{ck/6}$.*

Proof. Let v_s be the selected guess accordingly to procedure 2 and v_* the right guess, i.e. $\mathrm{OPT}/2 \leq v_* \leq \mathrm{OPT}$. Also, we denote by I_s the solution associated with guess v_s.

- If $v_s = v_*$, we already saw that the $1 - 1/e - \delta(\varepsilon)$ approximation is guaranteed if the event $E_{I_s} = \big||C'_s| - p|C_s|\big| < \varepsilon v_s p$ is met.
- If $v_s > v_*$, then $v_s \geq \mathrm{OPT}$ because each guess is of the form $2^g||\mathcal{S}||_\infty$, so v_s must be at least twice as big as v_*. Therefore, Lemma 6 ensures the $1-1/e-\delta(\varepsilon)$ approximation if the event E_{I_s} is met, because procedure 2 always takes a guess for which $|C'_s| \geq (1-\varepsilon)(1-1/e-\varepsilon)\lambda$.

To conclude, MACH$'_\gamma$ finds a $1-1/e-\delta(\varepsilon)$ approximation of *Maximum-k-Coverage* if $v_s \geq v_*$ and E_{I_s}. Furthermore, a consequence of Corollary 9 in §2.3 [17] is that, for the right guess v_*, if $\big||C'_*| - p|C_*|\big| < \varepsilon v_* p$ then $|C'_*| \geq (1-\varepsilon)(1-1/e-\varepsilon)\lambda$, which makes the right guess a possible choice for procedure 2. Therefore, $E_{I_*} \Rightarrow v_s \geq v_*$. Consequently:

$$\begin{aligned}
\mathbb{P}\left(\{v_s \geq v_*\} \cap E_{I_s}\right) = 1 - \mathbb{P}\left(\{v_s < v_*\} \cup \overline{E_{I_s}}\right) &\geq 1 - \mathbb{P}\left(\overline{E_{I_*}} \cup \overline{E_{I_s}}\right)\\
&\geq 1 - \mathbb{P}\left(\overline{E_{I_*}}\right) - \mathbb{P}\left(\overline{E_{I_s}}\right) \geq 1 - 2\frac{e}{m^{ck/6}}
\end{aligned}$$ ◀

E Proposition 8 (proof)

▶ **Proposition 8.** *Let h be a 2-independent hash function, and I, C, C', be defined accordingly. We have $\mathbb{P}\left(\,\big|\,|C'| - p|C|\,\big| \geq \varepsilon vp\,\right) \leq 2/(ck \log m)$.*

Proof. As before, $|C'|$ can be represented as $|C'| = \sum_{i=1}^{n} X_i$, where $X_i = \mathbb{1}_{i \in C} \mathbb{1}_{h(i)=1}$. Letting $\mathbb{V}$ stand for *variance*, we have:

$$\begin{aligned}
\mathbb{V}(|C'|) &= \mathbb{E}\left(|C'|^2\right) - \mathbb{E}(|C'|)^2 = \mathbb{E}\left(\left(\sum_{i=1}^{n} X_i\right)^2\right) - p^2|C|^2 \\
&= \mathbb{E}\left(\sum_{i=1}^{n} X_i + \sum_{i \neq j} X_i X_j\right) - p^2|C|^2 && X_i^2 = X_i \\
&= \mathbb{E}(|C'|) + \sum_{i \neq j} \mathbb{E}(X_i X_j) - p^2|C|^2 \\
&= p|C| + \sum_{i \neq j} \mathbb{E}(X_i)\,\mathbb{E}(X_j) - p^2|C|^2 && \text{pairwise independence} \\
&= p|C| + \sum_{i \neq j} \mathbb{1}_{i,j \in C} \mathbb{E}\left(\mathbb{1}_{h(i)=1}\right) \mathbb{E}\left(\mathbb{1}_{h(j)=1}\right) - p^2|C|^2 \\
&= p|C| + p^2\left(|C|^2 - |C|\right) - p^2|C|^2 = p(1-p)|C|
\end{aligned}$$

Including this value for the variance in Chebyshev's inequality, we have:

$$\begin{aligned}
\mathbb{P}\left(\ \big|\ |C'| - p|C|\ \big| \geq \varepsilon v p\ \right) &\leq \frac{\mathbb{V}(|C'|)}{(\varepsilon v p)^2} = \frac{p(1-p)|C|}{\varepsilon^2 v^2 p^2} \\
&\leq \frac{p\,\mathrm{OPT}}{\varepsilon^2 v^2 p^2} = \frac{\mathrm{OPT}}{\varepsilon^2 v^2 p} \\
&\leq \frac{2v}{\varepsilon^2 v^2 p} && \mathrm{OPT}/2 \leq v \\
&= \frac{2}{\varepsilon^2 \lambda} && p = \lambda / v \\
&= \frac{2}{\varepsilon^2 c \varepsilon^{-2} k \log m} = \frac{2}{ck \log m}\,.
\end{aligned}$$

◀

F Experimentation

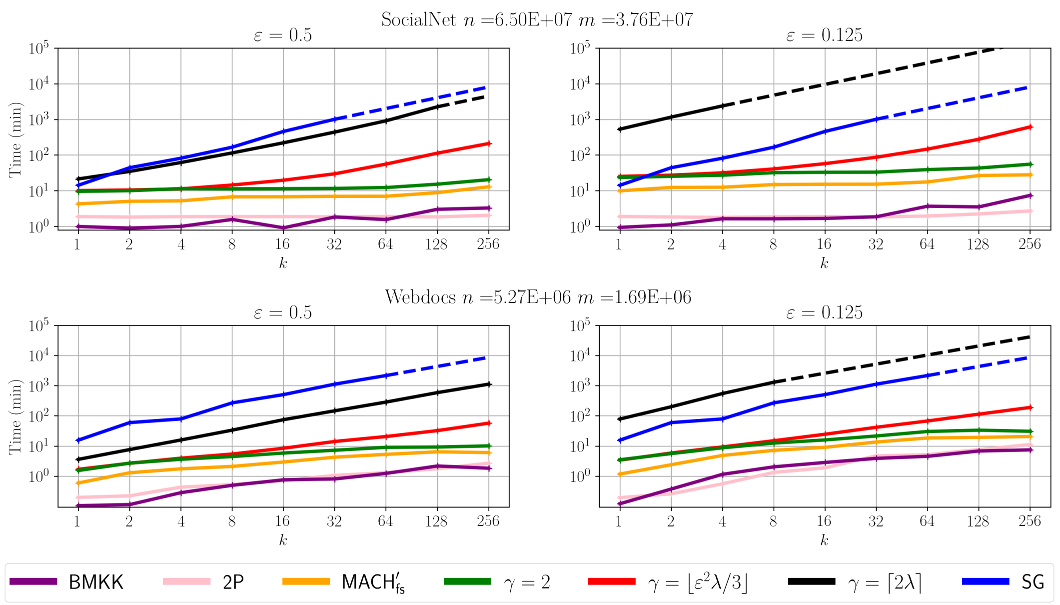


Figure 4 Running times of algorithms MACH'_γ, $\mathsf{MACH}'_{\mathsf{fs}}$, SG, and BMKK on datasets *SocialNet*, *UKUnion* and *Webdocs*.

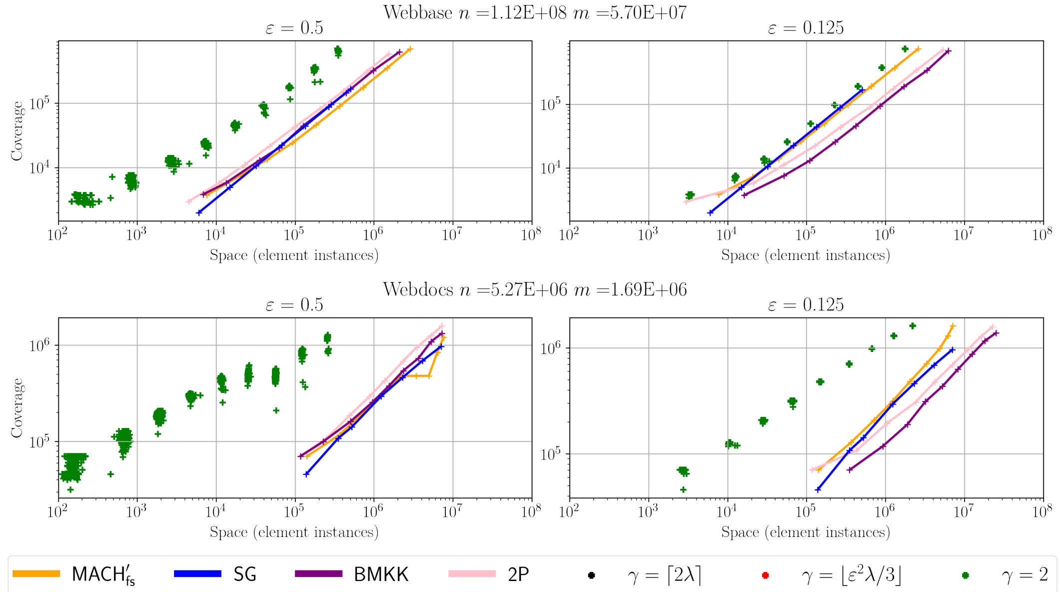


Figure 5 Coverage versus space, for $k \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$ on datasets *Webbase* and *Webdocs*. The anomalies of $\mathsf{MACH}'_{\mathsf{fs}}$ and MACH'_γ in *Webdocs* when $\varepsilon = 0.5$ coincide with the coverage quality drop in Figure 3.

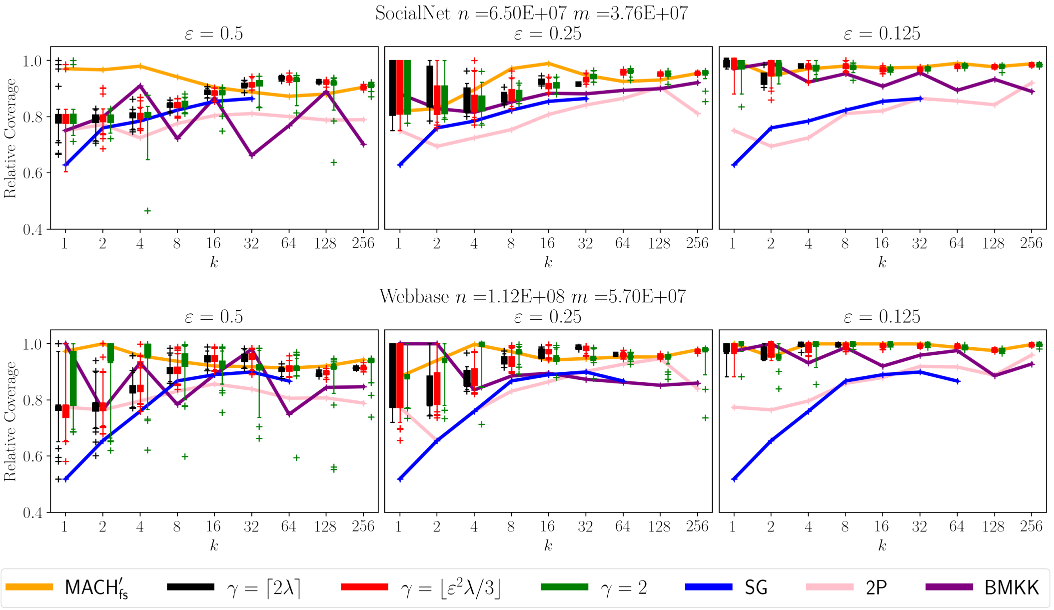


Figure 6 Box plot of coverage produced by MACH$'_\gamma$, SG and BMKK relative to the coverage produced by the greedy algorithm for the datasets *SocialNet* and *Webbase*.

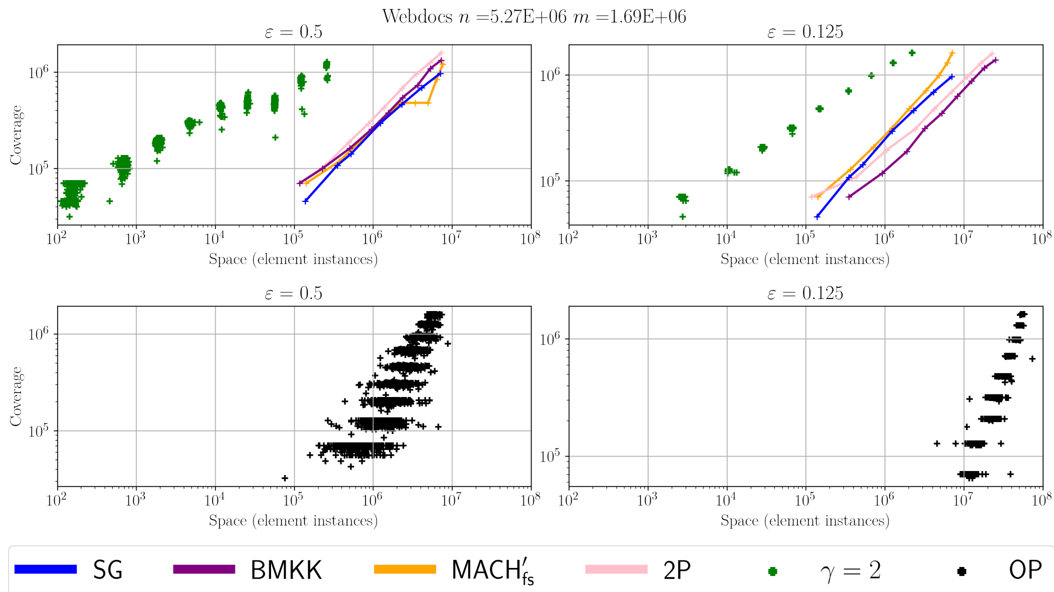


Figure 7 Coverage versus space, the $\tilde{O}(\varepsilon^{-2}m)$ space algorithm OP [17] consumes systematically more space than all the other $\tilde{O}(\varepsilon^{-d}n)$ space alternatives. This is because the $\tilde{O}(\varepsilon^{-d}n)$ space algorithms actually scale linearly with respect to the returned coverage size with a hidden constant close to one. On the other hand, the $\tilde{O}(\varepsilon^{-d}m)$ space algorithms, such as OP, have precisely a $\tilde{\Theta}(\varepsilon^{-d}m)$ space complexity with a much bigger hidden constant, storing a fraction of each set.